Sensemble: A Wireless Inertial Sensor System for Interactive Dance and Collective Motion Analysis

by

Ryan P. Aylward

B.S. Electrical and Computer Engineering,
University of Rochester, 2004

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

Masters of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

Author [illegible]
Program in Media Arts and Sciences
August 11, 2006

Certified by [illegible]
Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by [illegible]
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Sensemble: A Wireless Inertial Sensor System for Interactive Dance and Collective Motion Analysis

by

Ryan P. Aylward

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 11, 2006, in partial fulfillment of the
requirements for the degree of
Masters of Science in Media Arts and Sciences

Abstract

The motivation for this project is the recent opportunity to leverage low-power, high-bandwidth RF devices and compact inertial sensors to create a wearable, wireless, motion analysis system meeting the demands of many points of measurement and high data rates. This thesis outlines the implementation of such a system intended for interactive dance, in which sensor nodes are worn on the wrists and ankles of dancers in an ensemble.

Interactive dance is in some ways an ideal situation for pushing high performance requirements. Collecting data in a highly active environment of human motion demands a comfortable yet sturdy wearable design. Obtaining detailed information about the movement of the human body and the interaction of multiple human bodies demands many points of measurement and high resolution. Most importantly, using this information as a vehicle for interactive performance demands the real-time translation of data into an efficient feature set that a composer, designer, or choreographer can interpret.

Now that it is possible to extend expressive motion sensing to multiple points on multiple dancers, an interactive system is capable of responding not only to individual motions, but also to how an ensemble is working together. The primary goal in this work is to demonstrate that simple features describing this type of collective activity can be extracted from the system and interpreted real-time, in order to generate responsive music or other immediate feedback. To this end, relevant strategies for feature extraction and music generation were implemented and tested, using data from a small dance ensemble. The results presented in this thesis show promising opportunities for future development in the areas of dance and interactive performance. In the broader scope, the hope is to expand this system to other applications, such as analyzing the dynamics of team sports, physical therapy, biomotion measurement and analysis, or personal physical training. Preliminary testing in these areas is also discussed.

Thesis Supervisor: Joseph A. Paradiso
Title: Associate Professor of Media Arts and Sciences, Program in Media Arts and Sciences

Sensemble: A Wireless Inertial Sensor System for Interactive Dance and Collective Motion Analysis

by

Ryan P. Aylward

The following people served as readers for this thesis:

Thesis Reader [illegible]

Tod Machover
Professor of Media Arts and Sciences
Program in Media Arts and Sciences

Thesis Reader [illegible]

Gaetano Borriello
Professor of Computer Science and Engineering
University of Washington

Acknowledgements

To my advisor and readers Joseph Paradiso, Tod Machover, and Gaetano Borriello, without whose support, guidance, and expertise this thesis would not have been possible.

To France Telecom for their generous support and participation in this project, as well as to the Things That Think Consortium and the other sponsors of the MIT Media Laboratory for their support.

To my colleagues Mat Laibowitz, Mark Feldmeier, Josh Lifton, Gerardo Barroeta, Dan Lovell, Jon Gips, and Ari Benbasat, for providing the countless pieces of critical knowledge that have allowed this project to progress as far as it has, as well as the humor and camaraderie required to endure it.

To the Responsive Environments Group, an inspiring research community and a home away from home.

To Suelin Chen, Melissa "Buffy" Wells, and the OTHILA Dance Co., for offering their rehearsal time, insight, and dancing prowess.

To Eric Berkson, James Zachazewski, Thomas J. Gill, and the MGH Sports Medicine Department for their interest and enthusiasm in extending this project to high-performance athletics, and also to Jim Rowe, the Boston Red Sox, and all of the pertinent names I'm forgetting, for allowing us to come all the way to Spring Training to do so.

To Analog Devices, especially to John Geen, for providing sensors and advice for extending their capabilities.

To Mark Bocko, Cobi, Sandy Pentland, and Richard DeVaul, who should know that you played no small part in my finding an opportunity to study at the MIT Media Lab, and I am grateful for your help.

To Dev, Amos, Sarah, Lindsay, and Audrey, friends whose presence particularly reassures me that there will be more to my life than sensors.

To my family for their consistent love and support through the years, you especially will appreciate that I've built a g-force meter but it actually works this time.

And not least to Erika, whose patience, care, and understanding has helped me through an exceptionally tedious year, and whose power point aesthetic and skill with a wire stripper, crimper, battery pack, screwdriver, etc. has not gone unnoticed.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Sensemble is a new sensor platform developed for interactive dance or other real-time human motion sensing applications. The system is based around a flexible, high-speed network of small wireless sensor nodes equipped with very capable inertial measurement units (IMUs) and capacitive node-to-node proximity sensing. These sensor nodes can be worn at various locations on the user's body, typically at the wrists and ankles. Each device has the same set of capabilities, including its own wireless module, so they are completely encapsulated and rearrangeable within the system. The innovation explored in this work is the combination of high resolution, very low-latency measurement with a scalable, distributed network of compact sensors, to an extent which has been achieved in few systems to date. Additionally, a framework has been developed to process the large amounts of resulting data in real-time, with relevance to collective activity analysis and interactive feedback in a dance setting. Currently, 25 sensor nodes have been built and can run simultaneously on the network at high data rates.

1.1 Synopsis

This thesis provides a very detailed description of the design process and implementation, as well as early test applications, results, and directions for future development. The remainder of the *Introduction* is devoted to discussing the motivations behind this research and other approaches that have been attempted in the past. The *Hardware* chapter outlines each aspect of the hardware design, as well as the considerations and challenges associated with the choice of components and architecture. *Firmware and Data Flow* provides a similar summary of the communication structure and firmware running on the sensor nodes and basestation. The chapter entitled *Building Application Software* describes the basic process of developing host application software to collect and access data from a central computer. From here, *Feature Extraction* focuses on developing a theory for analyzing data, and presents basic results associated with generating useful features from dance motion as measured with inertial sensors. The next chapter, *Assessing Capacitive Sensor Performance*, presents additional results related to the performance of a capacitive sensing system which senses inter-node proximity. The *High Performance Adaption for Athletics* chapter outlines an excursion from dance in which the system was tested in a professional athletic training situation. The relevant modifications and preliminary results are all covered. This leads to *Test Application and Results for Dance*, in which the dancers return to study the possibility of using data from a performance to generate music in real time. This chapter addresses the strategy for translating data from the ensemble into musical sounds, additional issues regarding the specific implementation of a test application, and the results produced, as well as immediate directions for improvement. The final assessment of all aspects of system performance and suggestions for future work are made in *Evaluation and Conclusions.*

1.2 Motivation

The name 'Sensemble' evokes the feeling that the overlying goal is not just to track the movements of an arbitrary individual, but to sense the very subtle and expressive relationships within the collective movement of dancers in an ensemble. What qualities separate an ensemble from an untrained group of people moving cooperatively? What qualities lend a specific expressive mood to a dance? What role does the relationship between the movements of each individual and those of the rest of the group play in determining the character of a performance? If a computer system could address these concepts, then it would begin to process human movement in a more human fashion, and could react accordingly to generate feedback, for instance, responsive music. Of course, this hits on a host of very difficult problems at the core of so-called artificial intelligence, and the questions will not be answered here. However, to even pose such questions in a computational setting requires more advances in the sensing technology used to capture human movement as digital data. Many previous systems have explored the possibilities of different motion sensing arrangements for capturing expressive detail from one user. The high-performance devices presented here make their contribution by demonstrating real-time simultaneous measurement for up to 6 users, without sacrificing high resolution or multiple points of measurement per dancer.

With this in mind, the artistic implication of improving interactive dance applications is not the only reason for developing wireless sensor technology along the path taken in this work. Sensing dance is a situation which poses demanding technical challenges, such as the need to capture movement across a large and unrestricted space, the desire to take into account as many of the degrees of freedom of the body as possible, a design sturdy enough to handle strenuous physical activity without impeding this activity, sample rates which can cope with rapid motion, and high enough resolutions to capture subtlety of motion. In addition, the goal here is to collect, analyze, and translate this sensor data into music in real-time, meaning continuous transmission at high data rates is a necessity.

Of particular value is the fact that this set of challenges differs substantially from those faced in the more common sensor network applications. Typically, in these applications, functionality is pared down to an absolute minimum to create the cheapest and lowest power design possible. Much slower phenomena or transient events are measured, data is reduced within the network, and information trickles back slowly as a function of the power limitations. Faster radios are preferred in many cases because they can afford to spend more time turned off. By contrast, a system meeting the challenges of dance would also be useful in high performance, immediate feedback applications where the former sensor network design goals become a limitation. Possible examples could include augmenting musical ensembles, adding live media content to team sports broadcasts, professional athletic training, martial arts, interactive personal fitness monitoring, or physical therapy.

1.3 Related Work

1.3.1 Sensor Technology for Dance

In some ways, the history of artistic development in the last century has been the history of a stormy love-affair with rapidly developing technology. Among other benefits, technology offers the potential to merge forms of artistic and social expression in ways never before possible. On the other hand, the search for a satisfying union between human and machine capabilities continues, and artists arrive to explore every new innovation, often finding that technology still struggles to assuage the human imagination. Philosophy aside, it does not come as a surprise that technology and dance have a long history together. Dance offers a unique flexibility in that it is by nature open to the addition of music, video, lighting, narrative, or theatrics. The opportunity to automate and enhance the confluence of all these elements provides a natural avenue for the use of new technology. For example, as early as the 1930's, Leon Theremin adapted his well known electronic instrument to a full-body interface for dancers [1]. Even earlier, in the 1890's, Loie Fuller explored the dramatic potential of electric and other lighting technologies in her dance performances [2]. Today, the major advantages over earlier systems include the ability to make more precise measurements with a wider array of sensing strategies, the increased availability of processing power to accomplish more sophisticated interpretations of data, and a greatly enhanced flexibility in the area of media rendering.

As evident by the design presented in this thesis, one general strategy for designing dance interfaces involves placing sensors directly on the body, with a wireless communication link to transmit data to a central computer. Several interfaces of this sort have been developed to capture dance gestures over the last few decades. A number of the earliest of these placed specific emphasis on sensor devices built into shoes. For example, the Taptronics system designed in the 1980's for tap dance featured piezoelectric pickups at the toe and heel [3].

The Expressive Footwear shoes developed in 1997 by the Responsive Environments group at the MIT Media Lab [4] were capable of extracting much more detailed information from a dancer's feet — measurement capabilities included inertial sensors, pressure sensors along the sole, tap sensors, ultrasound range-finders, and a capacitive sensor for positioning over a floor-mounted electrode. All of the data generated by a pair of shoes could be received in real-time at 60Hz rates. Unfortunately, the shoe-based systems cannot measure upper body or arm motion, and have typically been designed with one dancer in mind. Because of bandwidth constraints, Expressive Footwear was never scaled beyond one pair of shoes.

Attempts to extend wearable dance sensors to other locations on the body have usually started with bendable sensors spanning primary joints, such as the elbows or knees. Architectures of this sort have been introduced at DIEM in Aarhus [5] and by Mark Coniglio of Trokia Ranch in New York [6]. The most extreme (and expensive) wearable joint-bend interfaces are outfits for full-body motion capture used in the computer graphics community, including exoskeletons or flexible fiber-optic angle-sensing technologies such as the ShapeWrap by Measurand [7]. Although these systems have become wireless, they employ a single radio in a beltpack or backpack, so the various sensors need to be tethered across the body to this central dispatcher. Such a "strapped-in" design can be very restrictive for a dancer, and the bulky infrastructure limits reliability in a performance setting. Additionally, many of these systems were developed for single subjects, and tend to be difficult to scale to an entire ensemble.

Magnetic tracking technology, such as the systems marketed by Polhemus [8] or Ascension [9], provides another alternative for motion capture that has been attempted with dancers on a few occasions [10]. Although these systems still require the user to wear sensor devices, they possess the ability to track absolute position in space, which is difficult with other wearable electronics. Unfortunately, precision magnetic tracking can be slow, especially with many points of measurement, and only works within a confined range around the transmitter. It is also expensive, and can be susceptible to tracking errors caused by metal in the surrounding environment. For these reasons, magnetic tracking is usually impractical in a performance situation with multiple dancers.

A more common approach to positional gesture tracking for dancers minimizes or eliminates body-worn hardware by exploiting computer vision — processing video from a camera or cameras watching the stage. This technique is now well established, and platforms like the Very Nervous System [11], EyesWeb [12], BigEye [13], and Jitter [14] are used by many composers and choreographers. The prevalence of optical tracking methods has even prompted some artists to develop their own video analysis tools, for example, [15, 16]. Real-time video analysis is processor intensive, and although the underlying technology and algorithms are steadily improving, computer vision is further limited by constraints on lighting and choreography. Robustness to occlusion and background noise remains problematic. As compared to inertial sensors, sample rates are severely limited in all but the most expensive video systems. Hence, obtaining multiple relevant features reliably from multiple dancers in real-time at the rates desired here can be difficult.

In constrained environments, there are certainly situations where computer vision or full-body sensor suits provide the most suitable means for highly accurate motion capture. However, when the goal is to obtain a general idea of gestural content as quickly as possible without compromising artistic expression, simpler techniques with fewer operating constraints can be considered. This is especially important for working with groups of people in a performance setting. Video tracking systems with relaxed measurement requirements have, in fact, been developed to accommodate this kind of general sensing of group interactions. For instance, Cinematrix was designed to capture the movement of an entire crowd holding colored reflective paddles [17]. Yet, in its extreme simplicity compared to high resolution motion capture, such an approach is much too coarse to describe dance movements in any detail, and the need for large reflectors on dancers is physically limiting. In addition, despite the struggle to adapt optical tracking methods to dance, acceleration and velocity measurements can be more useful than positional data from a gestural motion analysis standpoint. Accordingly, a system of wearable inertial sensor nodes with dedicated wireless connections was selected as the preferred strategy for this project. This approach has advantages over other wearable sensor techniques in that devices can be compact and do not have to be wired across the body, the system easily scales to a flexible number of dancers and number of measurement points per dancer, does not suffer from occlusion, and provides sensor data which is immediately relevant to features of human motion.

1.3.2 High-Performance Compact Wireless Sensor Platforms

The current hardware design has its roots in the Stack [18], a modular system, including full IMU card, developed at the MIT Media Lab several years ago as a smaller and more customizable alternative to the earlier Expressive Footwear design. However, the Stack was created at a time when the Expressive Footwear shoe was being moved to health monitoring applications [19, 20]. Because of this, the bandwidth requirements were not as stringent, and the radio used at the time was limited to data rates of only 115kbps, sufficient for two shoes on one user, but too low for a high resolution multi-user dance interface.

Although compact sensor clusters have since been developed at other institutes, many of them meet with the same limitations as the Stack. Very few systems have successfully combined low power and small size with the number of sensor channels high and data rates needed here. For instance, Motes [21] are quite established in the sensor network field, and they provide a configurable research platform for testing a variety of applications. However, they tend to support mainly peer-to-peer routing at modest data rates. Likewise, the Smart-Its platform and its descendants [22] are designed to work at data rates similar to the Stack. A system of sensor nodes specifically made for inertial motion tracking has been discussed in [23], but in this case a wireless solution for scaling the network effectively was left for future work.

More recently, Flety and collaborators at IRCAM [24] have built wireless sensor networks that use the WiFi 802.11 standard, and have used their WiSeBox system in a dance setting with multiple performers. The trouble with WiFi is that, although it provides very high data rates, it tends to be much too power hungry for efficient continuous operation with a small battery. Consequently, the WiSeBox architecture is based on a bulky central radio and processing unit, and the sensors themselves have to be wired across the body. Many other new systems favor sensor nodes with Bluetooth for achieving high data rates and ease of connectivity with consumer electronic devices. Some of these systems use inertial sensors to measure human motion, and have been designed with goals very similar to the ones outlined in this work. For example, researchers at ETH Zurich and CSN UMIT in Austria have collaborated on systems to analyze Butoh Dance [25] and Tai Chi [26]. Unfortunately, Bluetooth also has relatively high power requirements, and the size of the network is limited to only seven slave nodes per master. This means that Bluetooth places limitations on the scalability of a system for dance. In the case of [25], sensor nodes based on the ETH PadNet system [27] were again wired across the body to a central Bluetooth unit. Several dancers could potentially be accommodated with this structure, but the cumbersome wires are still present. The Tai Chi analysis presented in [26] uses IMUs and wireless modules from the commercially available XSens system [28]. Presumeably, there is an improvement over the custom made PadNet sensors, but XSens is still a modular Bluetooth system with sensor units that must be wired across the body to a larger radio unit.

A system with completely encapsulated sensor nodes and high data rates has been proposed in [29], including applications cited for dance and athletic motion analysis. However, the wireless link in this design is still Bluetooth, which means a new master would be needed for every seven nodes. Realistically, to expand this system to multiple users, each person would have to carry a Bluetooth master device with a higher speed uplink; as the authors suggest, a cell phone. Essentially the wires across the body have been replaced by power hungry Bluetooth channels, and the heavy central radio pack still exists. The solution proposed here is to dispense with the convenience of Bluetooth in favor of a custom protocol using a configurable high data rate, low-power radio. This approach has been taken before; for example, the very compact and low-power wireless accelerometer nodes designed by Emmanuel Tapia of the MIT Media Lab for ubiquitous health monitoring applications, which were capable of rates up to 1Mpbs [30]. By comparison, the applications of interest here require more sensor degrees of freedom and continuous operation, but the starting point is similar.

1.3.3 Collaborative Interfaces

The advantage to having enough sensing power to track multiple people simultaneously is an opportunity to study collaborative motion, and to design a system which responds to the relationships between people's activity. Processing data from sensors in a way that describes a high-level shared experience between devices can be difficult. Controlling an interface which requires cooperation can also be difficult, especially when the interface has difficultly perceiving cooperation in the way humans expect. The ability to respond to collaboration is desirable, however, as it is a fundamental basis of human interaction.

A large body of research in sociology and other fields has been devoted to bringing technology into this world of collaboration. In the context of this work, it is worth mentioning that a number of sensor platforms and analysis strategies have already been developed to study the possibility of collaboration between devices, and to help characterize collaboration between people. For instance, a project entitled "Are You With Me?" [31] used accelerometers to establish when a pair of devices was being carried by the same person. In [32] or [33], accelerometers were again used to detect when two devices were shaken at the same time or tapped together, respectively. In each of these cases, inertial sensors were used to determine a relationship via shared experience of movement, at which point a virtual association could be made between devices and responded to accordingly.

As mentioned previously, computer vision has been employed to sense collective movements on a larger scale with the Cinematrix system [17]. In a similar vein, inexpensive handheld 'shaker' sensors have been developed at the MIT Media Lab for characterizing large group movement, including an application for interactive dance, in which the activity patterns of an entire club influence the musical output of the DJ [34, 35]. Although the ability to scale dramatically enough to study crowd movement is interesting, the techniques used to do so are too coarse to capture the nuances of gestural interaction desired here. The sensor system developed in this thesis allows a collaborative interface to be formed with fewer people, but with much more expressive capability.

Collaboration, technology, and music, in the sense of multiple people sharing control over one musical output, frequently come together in spaces for shared composition online, such as [36, 37]. Increased connectivity between communities has created new ideas for how to generate music in a collaborative setting, along with interfaces that usually take the form of a type of forum or wiki, allowing users to modify visual musical score elements in a shared space [38, 39]. Several experimental music interfaces or installation pieces have also been developed to incorporate direct face-to-face collaboration, by using a convergence of sensor data and generating sound as a function of multi-user input [40]. However, the interaction is generally designed ad-hoc, without a concept of gestural vocabulary or the attempt to make conclusions related to meaningful characteristics of group interaction. Apart from the system described in [34], it seems that only a few attempts have been made to study collaborative performance interfaces from the perspective of detailed collective motion analysis. In one example, [41] discusses a sensor platform and topology to facilitate this type of system, but applications have not yet been realized. The developments made in this thesis barely scratch the surface of what could potentially be done with a high-resolution collaborative sensor interface for music, but it is the hope of this work that the capabilities of the system described here will inspire future development in this area.

Chapter 2

Hardware Design

2.1 Initial Requirements

Initial guidelines for the sensor design took into consideration restrictions and specifications on several levels, as motivated by the intent to work towards interactive dance. First, the topology was envisioned as a network of small sensor devices, each with its own wireless data link, to be worn on the wrists and ankles. These extremities were chosen as a compromise between ability to describe full body movements and the number of devices required per dancer. The small nodes in this scheme can easily be generalized or scaled to include measurement of the head, torso, or other locations on the body — this can be considered an added benefit.

Another prime stipulation was the quality and frequency of measurements coming from each device. The approach discussed in this work calls for full 3-axis acceleration and rotational velocity sensing, in other words, a full 6 degree-of-freedom inertial measurement unit (IMU), at each measurement point on the body. Based on the frequency content of active human motion [42] and the peak resolution of MEMS intertial sensors, sampling at 100Hz with 12 bits was also imposed to capture this data in sufficient detail. It was also considered beneficial to include options for other extended sensing modalities where resources were available. This led to the inclusion of a capacitive node-to-node proximity sensing system and flexible expansion port.

In designing the communications structure for the wireless network, speed was the first priority. Because the application involves real-time feedback in the form of music or other rapid response enabling the user to interact with the system, data must be received at a central computer quickly and simultaneously from all sensors. Hence, the system makes use of a star topology, in which a central basestation arbitrates and collects data from the entire network, as shown in Figure 2-1. The flow of information is direct, and the centralized structure makes it easy to synchronize sampling between all of the sensor nodes. Further, the intent was to scale to a small dance ensemble, with 20–40 nodes in the network all transmitting data at the rates suggested above. Using 20 nodes produce 144kbps of inertial data alone:

$$6 sensors/node \times 12 bits/sensor \times 20 nodes \times 100\,\text{Hz} \quad = \quad 144 kbps.$$

Factoring in a rough estimate of 60% loss due to overhead and protocol inefficiency, as well as four additional sensor signals that might be generated by the capacitive sensing system, the minimum data rate required in the channel is:

$$\frac{10 sensors/node \times 12 bits/sensor \times 20 nodes \times 100\,\text{Hz}}{1-0.6} \quad = \quad 600 kbps.$$

Clearly, the available bandwidth becomes a major issue.

The ultimate limiting factor, of course, is size. Sensors must not only be practical to wear, but also comfortable, light, and durable enough for strenuous dancing. Being wireless units, each node must be equipped with a battery, and this power source must fit into the description above, while providing enough energy for a performance of several hours. Although the power requirements here are not strenuous in the same sense as those for tiny wireless devices designed to operate for years on a single battery, the need for high bandwidth continuous operation in this application makes the selection of appropriate power sources limited.

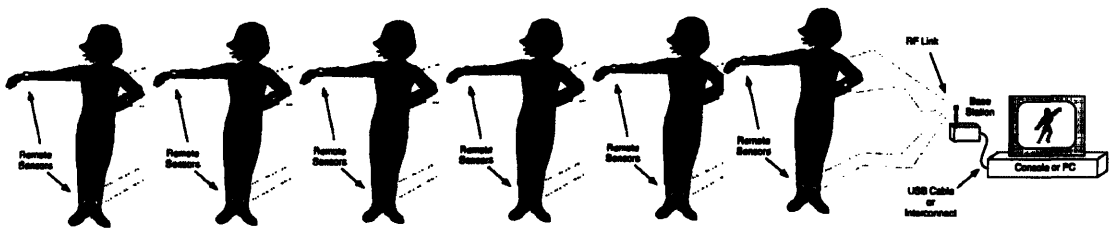


Figure 2-1: Basic network topology and deployment.

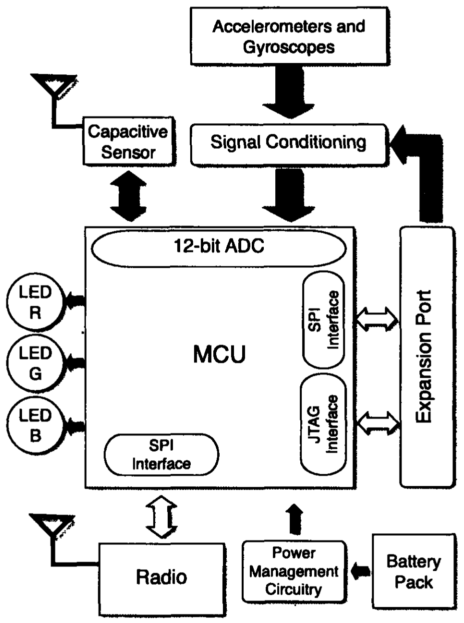


Figure 2-2: Sensor node architecture.

Figure 2-2 provides a pictorial overview of the sensor node architecture that grew out of these considerations. Each of the design elements will be described in detail in the sections to follow. Figure 2-3 shows the assembled hardware.

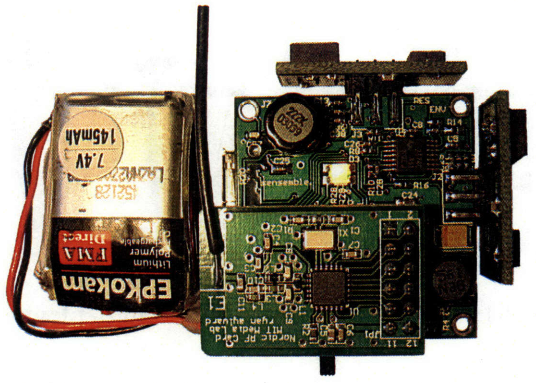


Figure 2-3: Assembled node board and battery pack.

2.2 Inertial Sensors and Signal Conditioning

Inertial sensing refers to the measurement of a moving object with either accelerometers or gyroscopes, which harness the properties of inertia in various ways to register changes in motion. Characterization of human movement through the dynamics of the body is the focus of this work, and hence inertial sensors are the basis of the hardware design. In general, inertial sensors can range from cheap and simple devices to detect a vibration [35] to expensive and elaborate instruments able to navigate an aircraft or missle [43, 44] (see Figure 2-4). The best middle ground in terms of size, cost, power consumption, and accuracy for continuous measurement on the body are sensors of the micro-electro-mechanical systems (MEMS) variety, which use micro-machined silicon structures to measure the inertial parameters. The performance of various MEMS accelerometers and rate gyroscopes have already been tested extensively in the work directly leading up to the design presented here [45, 46]. Although several new options have materialized recently, and sensors continue to improve, the current possibilities are similar to those discussed in the aforementioned work, and the design decisions draw heavily from these previous iterations.

Gyroscopes are more complicated than accelerometers and have imposed the most significant constraints on the sensor design. Table 2.1 shows the current availability of MEMS gyroscopes and accelerometers along with relevant specifications. In the case of rate gy-

	Part	Supply Voltage Range	Current per Axis	Number of Axes	Range	Sensitivity	Noise Floor	Bandwidth	Bits @ 50 Hz[a]	Turn-on Time (ms)	Size (mm)	Cost per axis (US$)
Accelerometers	Analog Devices ADXL203	3 – 6 V	0.35 mA	2	±1.7 g	$\frac{1000\,mV}{g}$	$\frac{110\mu g}{\sqrt{Hz}}$	2.5 kHz	10	5 ms[b]	5x5x2	$10
	Analog Devices ADXL210	3 – 5.25 V	.3 mA	2	±10 g	$\frac{200\,mV}{g}$	$\frac{200\mu g}{\sqrt{Hz}}$	6 kHz	12	5 ms[b]	5x5x2	$7.50
	Analog Devices ADXL320	2.4 – 6 V	.25 mA	2	±5 g	$\frac{174\,mV}{g}$	$\frac{250\mu g}{\sqrt{Hz}}$	2.5 kHz	11	5 ms[b]	4x4x1.45	$3
	Analog Devices ADXL330	2 – 3.6 V	.06 mA	3	±3 g	$\frac{300\,mV}{g}$	$\frac{280\mu g}{\sqrt{Hz}}$[c]	1.6 kHz[c]	10[c]	5 ms[d]	4x4x1.45	$3.33
	Silicon Designs 1210	5 V	7 mA	1	±5 g	$\frac{800\,mV}{g}$	$\frac{32\mu g}{\sqrt{Hz}}$	400 Hz	14	N/A	9x9x3	$120
	Silicon Designs 1221	5 V	8 mA	1	±2 g	$\frac{2000\,mV}{g}$	$\frac{5\mu g}{\sqrt{Hz}}$	400 Hz	15	N/A	9x9x3	$120
	ST Microelectronics LIS2L0xAL	2.4 – 5.25 V	0.43 mA	2	±2, ±6 g[e]	$\frac{1V}{g}$ @5 V	$\frac{30\mu g}{\sqrt{Hz}}$	2 kHz	12	N/A	5x5x1.6	$7.25
	ST Microelectronics LIS3L0xAL	2.4 – 3.6 V	0.32 mA	3	±2, ±6 g[e]	$\frac{660\,mV}{g}$@3.3 V	$\frac{50\mu g}{\sqrt{Hz}}$	1.5 kHz	12	N/A	5x5x1.6	$4.83
Gyroscopes	Murata ENC-03M	2.7 – 5.25 V	4.5 mA	1	±300°/sec	$\frac{.67\,mV}{°/sec}$	0.5°/sec[f] (50 Hz)	50 Hz	10	N/A	12.2x7x2.6	$80
	Analog Devices ADXRS150	4.75 – 5.25 V	6 mA	1	±150°/sec	$\frac{12.5\,mV}{°/sec}$	$\frac{.1°/sec}{\sqrt{Hz}}$	2 kHz	7	35ms	7x7x3.2	$35
	Analog Devices ADXRS300	4.75 – 5.25 V	6 mA	1	±300°/sec	$\frac{5\,mV}{°/sec}$	$\frac{.05°/sec}{\sqrt{Hz}}$	2 kHz	9	35ms	7x7x3.2	$35
	Analog Devices ADXRS401	4.75 – 5.25 V	6 mA	1	±75°/sec	$\frac{15\,mV}{°/sec}$	$\frac{.05°/sec}{\sqrt{Hz}}$	2 kHz	7	35ms	7x7x3.2	$35
	InvenSense IDG-300[g]	3 V	4.75 mA	2	±500°/sec	$\frac{2\,mV}{°/sec}$	$\frac{.014°/sec}{\sqrt{Hz}}$	140 Hz	12	200ms	6x6x1.4	N/A

[a] $\log_2(\frac{\text{Full Range}}{2.5\times\text{Noise floor}\times\sqrt{50}})$

[b] Depends on the external filtering capacitor placed at the output.

[c] Performance is slightly worse on the third (Z) axis.

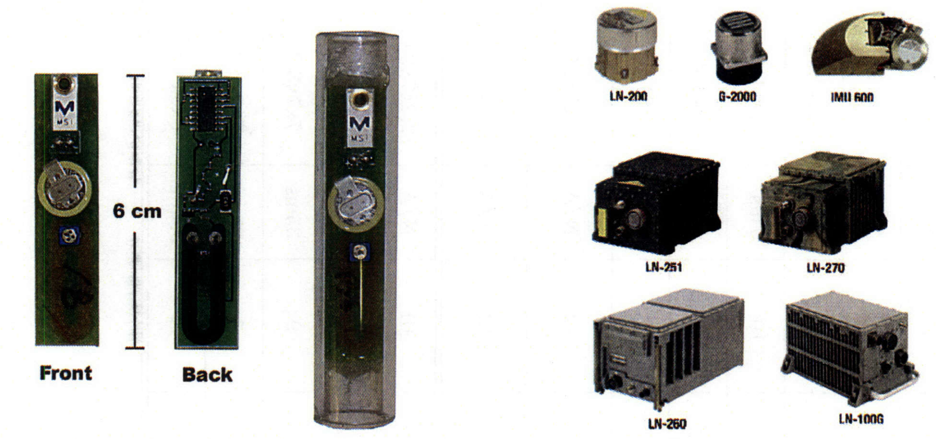
[d] Depends on the external filtering capacitor placed at the output.

[e] Values shown for the ±2g device.

[f] Not given on datasheet; value as measured by Benbasat [45].

[g] Not available when the hardware was being designed.

Table 2.1: Overview of some currently available inertial sensors.



(a) Ultra low-cost wireless motion sensor. (b) Heavy navigational IMUs.

Figure 2-4: Extreme examples of inertial sensing.

roscopes, the nominal ranges shown in degrees/sec are really quite small for rapid human movements. For instance, a quick 90-degree rotation of the wrist can easily be made in 0.25 seconds, amounting to an average rotational velocity of 360 degree/sec, already outside of the nominal range of most small gyros. The peak values of rotational velocity can be much higher than the average values, of course, meaning that the problem is not limited to such a deliberate rapid gesture. To maximize nominal range and minimize size, the ADXRS300, which had been selected in previous projects, was still the best choice at the time of the design. Luckily, the ADXRS300 also provides a simple way of extending its range up to a factor of four by setting an external resistor (R37 in Figure A-1). The extended range of roughly ± 1200 degree/sec was more than enough to capture dance activity without clipping the sensors. Methods are available to increase the range even more drastically for demanding applications such as professional athletics (see Section 7.3). Exploration of this area will be discussed in more detail in Chapter 7.

Because multiple axis gyros were not yet available when components were evaluated, the IMU was constructed with two orthogonal daughtercards to accommodate 3-axis sensing. The mounting procedure is shown in Figure 2-5, and discussed in more detail in Section 2.8. With these daughtercards already in place, it makes sense to use a pair of 2-axis accelerometers rather than being restricted to the limited selection of 3-axis accelerometers.

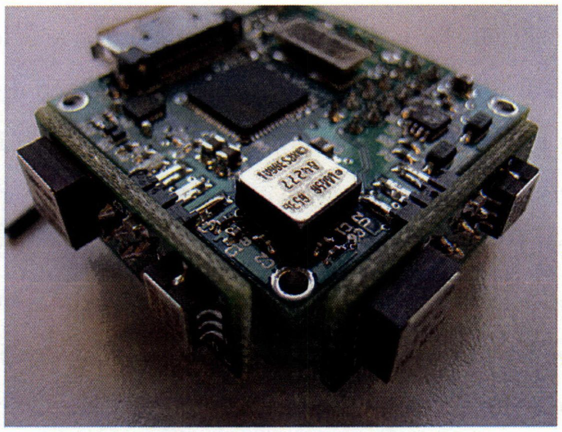


Figure 2-5: Orthogonal mounting of sensor daughtercards.

For certain applications, however, the new 3-axis ADXL330 would provide great savings in size, power, and cost. The discussion in [45] provides a justification for why it might be worthwhile to sacrifice this convenience for the sake of having multi-axis gyro signals available.

Initially, the ADXL203 accelerometer was chosen because of its high sensitivity and low noise. However, while the ADXL203 appears to be the best option for fine motion sensing at a reasonable price, it turns out that, as with the gyro, the range of ±1.7g is nowhere near enough to accommodate strenuous activity. Because of pin compatibility, the design dropped back to the older ADXL210E, which is less accurate but provides a full ±10g range. Unfortunately, ±10g is too large of a range for most human activity, and experimentation has suggested that a ±5g accelerometer such as the non-pin-compatible ADXL320 would be a more appropriate choice for dance. Future designs will support footprints for these newer components.

The power requirements of the ADXRS300 gyros necessitate that they run from a 5V supply. The ADXL210 accelerometers are more flexible, but for simplicity they also use 5V here, as they are mounted on daughtercards along with the gyroscopes. The digital circuitry, however, including analog to digital conversion, runs from a 3.3V supply. The necessary voltage conversion, as well as buffering and filtering, is accomplished by the analog signal conditioning stage (see Figure A-1). After the inertial sensors' own internal output filter stages, the signals are scaled down with inverting amplifiers and lowpass filtered with a first order roll-off at 22Hz. This limits aliasing during analog to digital conversion, while preserving most of the information one could expect to find at 100Hz sampling rates. The input and feedback resistor values are large to compensate for the high output impedance of the sensors (for the accelerometers, roughly $32\,\mathrm{k}\Omega$). Unfortunately, the sensor signals had to be attenuated to make the conversion from 5V to 3.3V ranges. Attenuation is only possible with an inverting amplifier; otherwise a high input impedance non-inverting configuration with smaller associated resistances could have been employed, potentially improving noise performance.

Six op-amps are required for the signal conditioning stage, one for each IMU axis. Rather than using three dual packages, two quads were used, leaving one free amplifier to buffer the reference voltage needed for voltage conversion, and one free amplifier to provide a spare analog signal path for future expansion, which will be further addressed below. To select an appropriate op-amp, the focus was on low noise, low power, and rail-to-rail output to take advantage of as much of the bit depth of the ADC as possible. The TLV 2474 was selected as a good all-around amplifier meeting these criteria at a low cost. Accounting for the entire signal path, the noise floor is low enough to get 10-bit precision from the inertial sensors.

2.3 Microcontroller

Each sensor node is equipped with a microcontroller (MCU) which functions as a local control center for collecting and processing data, arbitrating sensor behavior, maintaining communication with the RF module, and timing events. In this particular design, the focus was on requirements of speed for making effective use of the high bandwidth wireless network and associated tight timing requirements, availability of analog inputs and ADC resolution for handling several high quality sensor data streams, digital I/O capabilities for flexible control over other circuitry and peripheral devices, adequate RAM for offline computation, and internal flash memory for the option of logging data. The Texas Instruments MSP430 family of MCUs provides respectable solutions to these design criteria, with the added benefits of very low power operation intended for embedded devices. Ease of use also plays a role, and previous experience with the MSP430 in other projects, as well as familiarity with and access to the Rowley CrossWorks™ development environment [47], made this MCU family attractive. Specifically, the sensor nodes use the MSP430F148, an 8MHz 16-bit RISC MCU with 8 analog input channels, 12-bit ADC, two serial interfaces, two highly flexible 16 bit timers, 2kB of RAM, and 48kB of internal flash memory. The MSP430F149 is a pin compatible upgrade with 60kB of flash. Both MCUs consume only 3.4mA when idle in active mode, and less than 0.5μA in the deepest sleep mode. They are available in compact 64-pin leadless (QFN) packages, providing ample I/O within a 9x9 mm footprint. Programming the network can be accomplished node by node through a standard JTAG interface or wirelessly with a bootloader, although the latter was not implemented. The firmware running on the MCU in the current implementation and the allotment of MCU resources will be described gradually as the remaining design elements are defined below.

2.4 Capacitive Node-to-Node Proximity Sensing

Despite the ability of MEMS accelerometers and gyroscopes to perform well in detailed gesture tracking applications, their signals are still not clean enough to provide positional tracking in space. Because the computation of position from velocity or acceleration requires a double integration, noise and bias rapidly accumulate in the result, a phenomenon known as drift. To recover from drift, navigational systems need to be calibrated periodically with an outside reference. In the case of small and relatively noisy MEMS sensors, this recalibration would have to occur on the order of every 5 seconds [45]. Even then, finding a reliable calibration reference is problematic. Since the focus here is on relative qualities of movement, activity features, and subjective comparisons between gestures, it is mostly possible to avoid the slippery slope of calibration, absolute measurement, and positional navigation. However, in some cases it would be satisfying to make a simple statement about the location of one sensor node with respect to another. For instance, are the wearers arms far apart or close together? It would be impossible to answer this question with inertial sensors alone. To help extract this type of information, a capacitive sensor system was added to the design that supports rough measurements of distance between pairs of nodes in the network.

Capacitive sensing, alternatively known as electric field sensing, may come in a variety of forms, but the term typically implies a measurement based on the relationship between electrode spacing and the capacitance across two electrodes [48]. For an ideal plate capacitor, with electrode spacing d much smaller than electrode area A, and dielectric constant ϵ, capacitance C is inversely proportional to d as shown:

$$C = \frac{\epsilon A}{d}$$

In human interface applications, the electrodes may be irregular conductors worn on the body, or the body itself may act as a not-so conductive ground electrode, and the distances of interest are comparatively large [49, 50]. Because of this, the capacitance drops off much more quickly with d. In addition, the capacitive coupling in such a setting usually occurs through the air, with its low dielectric constant and sensitivity to environmental factors such as humidity. Hence, the challenge is to appropriately measure small and notoriously noisy changes in capacitance.

For this application, the goal is to measure the distance between two sensor nodes with conductive electrodes attached, ideally without the influence of the body and its surroundings, which are effectively grounded. The best way to accomplish this is to transmit a high amplitude carrier signal from one electrode and receive and demodulate the signal on the other, to determine a received carrier amplitude. The amplitude of the receive signal is expected to vary with capacitance because the capacitive coupling channel looks like a variable cutoff highpass filter in the appropriate frequency range of 50–100kHz. This arrangement is similar to transmit mode sensing as discussed in [49, 50].

In the case of two isolated wireless sensor nodes, there are two problems to address with this transmit-receive capacitive sensing arrangement. First, as there is no direct electrical path between the two devices, there is no common reference, resulting in a decreased signal-to-noise ratio and the potential for unpredictable behavior. The solution is to ground the devices to the body, allowing the sensor nodes worn by the same user to share a reference through the skin. Although the body is not an ideal conductor, this practice has been documented [49], and indeed greatly improves capacitive sensor performance as discussed in Chapter 6. Of course, directly grounding the body highlights the fact that the capacitive sensor will never actually be free from the influence of the body and its surroundings. As a ground plane, the body will readily shield the electrodes if the path between them becomes blocked by it, resulting in reduced signal. Similarly, the existence of nearby floating conductors can actually increase the coupling between the electrodes, resulting in an improved signal. This behavior can cause confusion if, for some reason, one attempts to assess the performance of a capacitive sensor by placing electrodes near the metal ruler in my office, for instance.

The second problem, also due to the electrical isolation of the wireless devices, is that the receiver has no information about the phase of the transmitted signal, which it would ideally know, in order to synchronously demodulate the carrier and measure its amplitude properly, while rejecting uncorrelated noise. The simplest way to get around the need for phase coherence is to sample in quadrature, or at exactly four times the transmit carrier frequency. Assuming the carrier frequency is known, quadrature sampling produces four values for every period, known as the in-phase and quadrature components, which can be combined to reconstruct any bandpass signal without the need to know the initial phase [51]. To make the capacitance measurement, only the magnitude of the carrier is needed, which can be computed for each period as:

$$V = \sqrt{(S_0 - S_{180})^2 + (S_{90} - S_{270})^2}$$

This process of quadrature sampling and reconstruction is essentially demodulating a carrier signal down to baseband. Then, the magnitude of the complex envelope around DC is measured to compute the amplitude estimate. Supposing there is a small error between the actual transmitted frequency and the expected carrier frequency, it will be reflected as

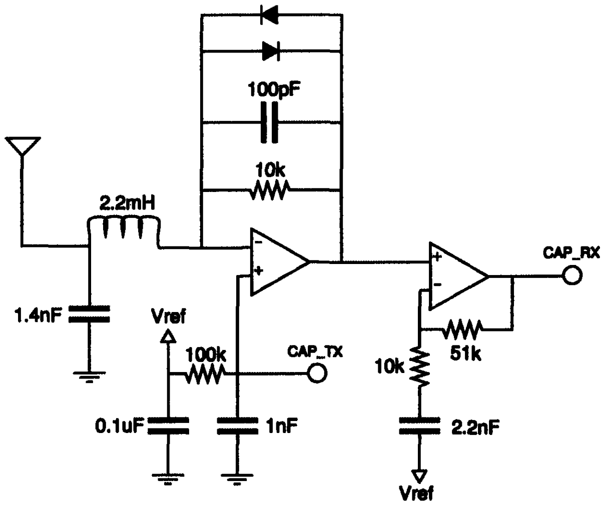


Figure 2-6: Capacitive transceiver circuit.

low frequency noise or beats. Typically, the amplitude estimate is computed and averaged over several periods of the carrier to reduce this effect. The use of quadrature sampling for electrically isolated capacitive sensing devices has been extensively discussed in previous work at the MIT Media Laboratory [52]. In particular, Smith addresses ways to handle the error associated with simplifying the calculations required during demodulation, for cases where processing power is limited. In the work presented here, sufficient processing power is more readily available, and making the full calculations is favorable. Performing the square root operation is not strictly necessary, but in order to compress the result into an appropriate range of 12 bits, it was carried out in this case with a significant loss in processing time, as made more apparent in Section 3.6. Optimization of the capacitive detection algorithm will be left for future development.

A capacitive measurement can only be made for one pair of nodes at a time. During this transaction, one node must play the role of the receiver and the other must play the role of the transmitter. The circuitry, also inspired by previous work detailed in [52], is such that the two roles can be swapped easily by the microcontroller, and each node can make use of the same amplifier stage and electrode for both transmit and receive modes (see Figures 2-6,A-1).

In transmit mode, the microcontroller generates a square wave on `CAP_TX` at the desired transmit frequency, which is then buffered in the amplifier stage, and drives a tuned series LC resonator to generate a high amplitude sinusoid at the electrode. In this implementation, 40V peak-to-peak signals can be generated, although even higher amplitude signals were reported with the setup in [52]. Because the energy is stored within the LC circuit, these high amplitude signals can be generated with relatively little current draw.

In receive mode, `CAP_TX` is set to high impedance and a bias is applied at the non-inverting input to the amplifier stage. The first op-amp and LC circuit then act as an inverting amplifier with a high-Q bandpass filter tuned to the carrier frequency — this isolates and amplifies a carrier signal received at the electrode. A second non-inverting amplifier stage buffers the received signal and applies more gain before sending the received sinusoid (`CAP_RX`) to an ADC input. At this point, the microcontroller handles quadrature sampling and estimates the amplitude of the received signal.

The frequency of operation was determined by the ability of the microcontroller to sample and store values at precisely four times this frequency, in order to achieve quadrature sampling. It was also necessary to generate the driving signal for the transmitter by toggling a pin at precisely two times the frequency of operation. Physical limits on the ADC, clock speed, and number of instructions to be executed place an upper limit on the possible choices. At the same time, within the range of frequencies appropriate for measuring capacitive coupling (typically 10–100kHz), higher values are preferable because the strength of the coupling is linearly proportional to frequency [52].

With these restrictions in mind, 90.9kHz was chosen as the fastest feasible carrier frequency for the MSP430F14x. With the clock frequency set to the maximum rate of 8MHz, the 90.9kHz transmit drive signal can be generated by toggling a digital output pin every 44 clock cycles. The received signal must then be sampled every 22 clock cycles to capture the in-phase and quadrature components. During receive mode, the MCU stores an array of samples over several periods of the carrier frequency, so that the averaging and quadrature computations can be made after the time-sensitive sampling process. Algorithms to accomplish these tasks in firmware are shown in Listings 2.1, 2.2, 2.3 (see also Listing C.5).

They have been implemented in C, but include careful timing that is device specific and may also be unique to the compiler included in the Rowley CrossWorks [47] development environment.

```
int i=0;
int length;

P4DIR |= 0x40;          //CAP_TX mode

//toggle CAP_TX at 90.9kHz
//Number of toggles is twice the number of periods
length = (CAP_LENGTH<<1);

while(i<length){
  __delay_cycles(35);
  P4OUT ^= 0x40;      //CAP_TX signal
  i++;
  }

P4DIR &= ~0x40;       //CAP_RX mode
```

Listing 2.1: Capacitive transmit algorithm.

```
int temp=0;

P4DIR &= ~0x40;       //CAP_RX mode

//This captures samples in quadrature
//(4 samples per 90.9kHz cycle)
//for a duration of CAP_LENGTH/2 cycles

ADC12CTL0 |= ADC12SC;       //Initial ADC sample trigger
ADC12CTL0 &= ~ADC12SC;
__delay_cycles(7);

while(temp<CAP_LENGTH){
  ADC12CTL0 |= ADC12SC;     //Trigger A
  ADC12CTL0 &= ~ADC12SC;
  PAC_MAN0[temp]=ADC12MEM8; //Store sample from trigger B
  __delay_cycles(6);
  ADC12CTL0 |= ADC12SC;     //Trigger B
  ADC12CTL0 &= ~ADC12SC;
  PAC_MAN1[temp]=ADC12MEM9; //Store sample from trigger A
  temp++;
  }
```

Listing 2.2: Capacitive receive algorithm.

The full sampling rate for the received signal corresponds to 363.6kHz, much higher than what would typically be demanded from a microcontroller ADC. To ease the requirements, two ADC memory locations are used for the in-phase and quadrature components, respectively. In this way, the conversions can be computed as if two different signals are being sampled at half the rate, with the advantage that a sample intended for the second memory location can be triggered before the previous sample has finished being stored to the first memory location. To be sure that the MSP430 can handle such a rate, one must refer to the User's Guide [53] to determine the minimum time required to capture a sample and the minimum time required to convert the sample. In the relevant mode of operation, a sample

```
int temp;
long signed int I=0;
long signed int Q=0;
long signed int diff;
unsigned int val;
float S;

if (source_idx < MAX_CAP_RX_NODES){
  //Accumulate samples in quadrature
  diff = PAC_MAN0[0]-PAC_MAN0[1];
  I = diff*diff;
  diff = PAC_MAN1[0]-PAC_MAN1[1];
  Q = diff*diff;

  for(temp=2;temp<CAP_LENGTH-1;temp++){
    diff = PAC_MAN0[temp]-PAC_MAN0[temp+1];
    I += diff*diff;
    diff = PAC_MAN1[temp]-PAC_MAN1[temp+1];
    Q += diff*diff;
  }
  S = sqrtf((float)((I + Q)>>CAP_PWROF2));
  val = (unsigned int)ceilf(S);

  //Hard limit to 12 byte range
  if(val < 0x0FFF){
    CapRX[source_idx] = val;
  }
  else{
    CapRX[source_idx] = 0x0FFF;
  }
}
```

Listing 2.3: Capacitive calculation algorithm.

is captured by setting a register flag high for the sample duration t_{samp}, after which the user must wait a minimum time $t_{convert}$ before reading the relevant ADC output register. In order to ensure that the conversion for an in-phase sample is complete before conversion begins on the next quadrature sample, there must also be a separation of at least $t_{convert}$ between sample triggers. Since a sample must be taken every 22 clock cycles to acheive the full 363.6kHz rate with an 8MHz clock, the first requirement is that:

$$t_{convert} < 22 \textit{ clock cycles.}$$

The conversion time $t_{convert}$ is specified to be a constant 13.5 clock cycles, so this condition is met. Although there are 22 clock cycles between each sample, this time cannot be taken up entirely by the sample capture duration t_{samp}. After each new sample is captured and prepared for conversion, code must execute to move the previous sample value to a storage array (see Listing 2.2). Therefore the maximum time allowed for sample capture is given by:

$$t_{samp} < 22 \textit{ clock cycles} - t_{code} \tag{2.1}$$

where t_{code} is the execution time required in clock cycles. At the same time, a lower limit on t_{samp} is determined by the time required to charge the input stage of the converter, as modeled by an internal capacitance, C_I, and an internal resistance, R_I. In turn, this charging time depends on the output impedance R_S of the device connected to the input of the ADC:

$$t_{samp} > (R_S + R_I) \times ln(2)^{13} \times C_I + 800ns$$

$$R_I = 2\,\text{k}\Omega, \quad C_I = 40pF$$

Now, assuming zero output impedance at the input, the smallest possible value of t_{samp} is found to be 1.52μs, or 12.16 clock cycles of the 8MHz clock. The smallest whole number of cycles available for sample capture, then, is 13. Since this only allows 9 cycles for code execution, it is already close to the maximum value imposed above in Equation 2.1. Finally, the upper limit on the driving output impedance R_S can be estimated to verify that an input signal can be sampled at this speed:

$$\begin{aligned} t_{samp} &= 13 \textit{ clock cycles} &&= 1.625\mu s \\ R_S &< \frac{t_{samp} - 800ns}{C_I \times ln(2)^{13}} - R_I &&= \frac{0.825\mu s}{360.44pF} - 2\,\text{k}\Omega \\ & && R_S < 289\,\Omega \end{aligned}$$

Although this clearly pushes the limits of the converter, an op-amp stage with a typical output impedance of 10 Ω or less is used to drive the analog input. In this case, the output impedance is certainly low enough to enable the desired sampling rate. Proper behavior was verified by comparing the amplitude of the input to the ADC as measured on an oscilloscope with the estimated value obtained through quadrature sampling and computation.

The circuit components for the capacitive transceiver were chosen to ensure as much gain as possible. The LC circuit must have a high Q to generate the high amplitude transmit signal, and in turn must be tuned as closely as possible to the driving frequency of 90.9kHz. To accommodate this, an inductor was chosen with low equivalent series resistance (ESR), and in some cases the capacitance was hand-tuned by stacking surface mount components in parallel for the best performance, as shown in Figure 2-7. Originally, variable capacitors were considered too large and were difficult to obtain for the small range of values needed. In future designs, they may be an effective addition, as the precision of standard components is low when faced with the problem of matching highly tuned resonators, and having to tune each device by stacking capacitors is impractical. The op-amp was chosen with the major requirement of high enough gain-bandwidth product to effectively amplify signals in the range of 100kHz. A dual op-amp was required for transceiver and secondary gain stages (see Figure A-1). Also of importance were low power, and rail-to-rail operation to maximize the amplitude of the square-wave driving signal applied to the resonator in transmit mode. The LT6221 was selected as a reasonable solution, with a 60MHz gain-bandwidth product providing the potential for 600x gain at 100kHz, rail-to-rail input and output, and only 1mA supply current required per amplifier. However, the LT6221 suffers from a high noise floor, which is exacerbated by the fact that, by design, the LC resonator provides huge gain at the frequency of interest in receive mode. The result is a receiver with a lower signal to noise ratio than what may have been possible with a low-noise, non-rail-to-rail amplifier.

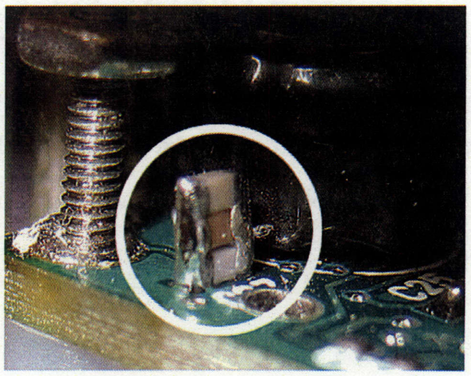


Figure 2-7: Stacking surface mount capacitors to tune the resonant circuit.

Another oversight present in the current design is the sharing of bias voltage `VREF` between the analog circuitry associated with the inertial sensors and the capacitive receiver circuitry (see Figure A-1). This bias voltage was originally intended to be half of the 3.3V digital supply `VLogic`, the correct bias point for the capacitive receiver, but because of the voltage conversion requirements built into the inertial sensor circuitry, `VREF` has actually been set to 2.06V. Therefore, the capacitive circuitry carries an unintended bias offset. The offset does not effect the amplitude measurement because quadrature computation is differential. However, when the received signal is strong, it will saturate the positive rail before saturating the negative rail, creating reduced sensitivity and increased noise just before the saturation point when the sensors are moved very close together. This error is trivial to correct, but will wait for future board revisions. A final performance evaluation of the capacitive sensor system is left for Chapter 6.

2.5 Extended Capabilities

2.5.1 Integrating Additional Sensors

On top of its core capabilities, the Sensemble sensor platform was designed with flexible options for future expandability, including the integration of additional sensors or peripheral devices such as external memory. A sturdy, compact 18-pin cable connector has been included, which acts as a full expansion port as well as the MCU programming interface, as shown in Figure 2-8. A cable connector was preferred over board-to-board connectors with the expectation that it would be less cumbersome to wire expansion devices across the body if necessary than to add more rigid mass to the sensor unit. The expansion port provides power lines at 3.3V, 5V, and ground, two digital I/O lines from the MCU, a free SPI port from the MCU, and a flexible analog sensor input which gives full access to the free signal conditioning path and ADC input on the main board (see Figure A-1).

Magnetometers

Several specific technologies omitted from the design come to mind as candidates for adding capability to the system, especially as a tool for interactive biomotion analysis. For instance, the difficulties of tracking position with inertial devices have been addressed, and this work suggests one way to dismiss positional tracking while still establishing some degree of spatial information, by using capacitive sensors. However, it has become popular to attempt true position tracking with MEMS IMUs by combining the inertial devices with three axis magnetometers. For instance, this is the case with the previously mentioned XSens system [28], the system described in [23], and the Motion Bands designed at the Nokia Research

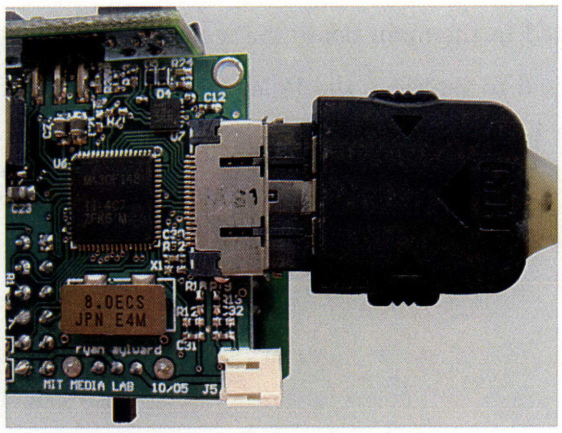


Figure 2-8: Expansion port and cable.

Center [29]. Magnetometers measure the Earths magnetic field to provide a continuous orientation reference, by which the initial angle of rotation can be recalibrated, and the gravity component of acceleration can be separated from other accelerations. Unfortunately, there are certain indoor environments, such as those cluttered with metal or near the floor, in which the magnetic readings become inaccurate. Handling these inaccuracies to create a high performance position tracking system was not the primary goal in this design, as accelerations and velocities alone can be used to characterize expressive movements. Hence, the use of magnetometers is left as a possibility with the expansion port, which could accommodate a single axis measurement using the free analog input, or a full three axes using a digital output, multiplexer, and analog input.

Heart Rate Monitoring

Another likely addition is heart-rate monitoring capability, to describe strenuous activity in a manner directly relevant to personal fitness. Many commercially available products already combine the power of accelerometer data with various types of personal physical training technology [54]. Experimental sensor systems developed for research at the MIT Media Lab have also demonstrated simple ways of integrating commercial heart monitoring tools with custom-built hardware. In particular, the hardware developed for [55] is able to measure heart rate reported from the Polar [56] heart monitor using their watch-sized receiver board connected to the main board via a single digital input pin. Using a similar setup, it would be trivial to interface the Polar monitor with the Sensemble sensor node through a free digital input. In fact, this possibility has been anticipated in that the expansion port includes a digital line labeled `Pulse`. The system has not yet been tested in conjunction with heart rate monitoring, however.

Tactile Inputs

Finally, there may be merit in adding tactile inputs to the sensor network, such as pressure sensors or buttons on some of the nodes. Without tactile feedback, it is difficult to give the user satisfying control over possible interface elements such as selecting modes of operation. The expansion port on each node potentially supports one continuous pressure sensor through the analog input, and two buttons through the free digital inputs. Adding buttons and transmitting button presses is a simple proposition that does not fundamentally broaden the concept presented here, and is left up to application specific needs in the course of future development.

2.5.2 Integrating Peripheral Devices

Other enhanced sensor arrangements can be accommodated, not limited to the addition of single analog or digital signal sources. Provided that the added device supports serial peripheral interface (SPI) communication, the expansion port allows the possibility of attaching a new full-scale sensor node with its own suite of capabilities and processing power, or a large external memory module for logging data. This is accomplished through the expansion port's SPI interface, which can be used in three or four wire modes, can run at speeds greater than 1Mbps, and allows the Sensemble node to be run as either master or slave. Making such dramatic extensions to the abilities of the sensor node is, of course, contingent on the power budget and may require the addition of a second power source or modification of the power regulation circuitry.

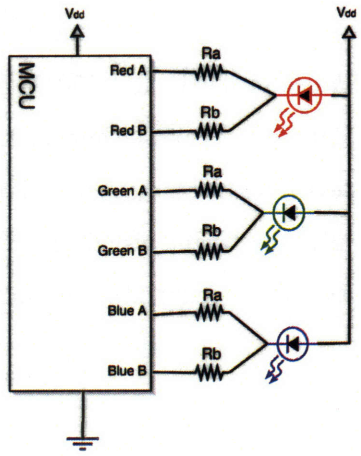


Figure 2-9: Resistor network for controlling the RGB LED.

2.5.3 Visual Feedback

Aesthetic considerations led to the addition of LEDs to the main board for visual appeal and feedback. Originally, three separate surface mount LEDs were used, but to save space, increase visibility, and provide for more effective color combination, a single bright tri-color (RGB) surface mount LED with low current requirements was chosen in the final design. The Fairchild QTLP650D was selected as one of the brightest and most well balanced of such devices at low current levels. Typically, an RGB LED can be driven through three MCU pins and associated current-limiting resistors. However, as will become apparent in Chapter 3, the timing for sampling data and running the communication protocol is so tight in this application that there is no room to handle the traditional pulse width modulation (PWM) used to mix red, green, and blue levels. Therefore, the LED is driven from six MCU pins through a resistor network providing three possible resistance values for each color, as illustrated in Figure 2-9 below. For instance, when *RedA* and *RedB* are pulled down together, the red LED is driven through the parallel combination of the resistors *Ra* and *Rb*. When only one driving pin is used, the other is set to high impedance to close off its current path. When both pins are pulled up, the LED is off. This arrangement provides four possible brightness levels for each LED; in other words, 64 possible colors.

The LED can be given a small duty cycle based around the existing firmware timing structure. However, it still creates a significant peak current draw, up to 25mA total for bright white with resistors Ra and Rb set to $430\,\Omega$ and $150\,\Omega$, respectively. There are several acceptable reasons to accommodate a fancy power hungry LED. The glowing sensors could potentially become an output device with the motion of dancers creating patterns of color on a dark stage. More practically, the bright visual markers have been indispensable for synchronizing collected data with video documentation during the trial runs.

2.6 Data Radio

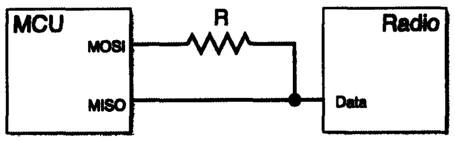
The wireless radio module on each sensor node is the foundation of this design concept. Giving each node its own high-speed wireless connection allows it to be truly self contained and physically independent of other nodes. In this way, the network is scalable and configurable, within the limits of available bandwidth and feasible battery power sources. Although this idea is an old one, it has recently gained momentum with the availability of increasingly fast low-power radio modules.

Much of the research in wireless sensor networks focuses on the limitations of power, as the concern is often long-term monitoring. Here, the goal is different in that the concern is with high resolution real-time monitoring on a short time scale, and therefore the primary focus is on limitations of bandwidth. Mainly for this reason, relying on the capabilities of hardware available from other institutions, such as Motes, to develop the Sensemble interface was insufficient. Similarly, standard communications protocols provide little help. Zigbee is a limited data rate protocol for typical low-power sensor network applications [57]. Bluetooth and WiFi provide high data rates, but were not developed with small sensor devices in mind, and are too cumbersome in terms of both software restrictions and power requirements.

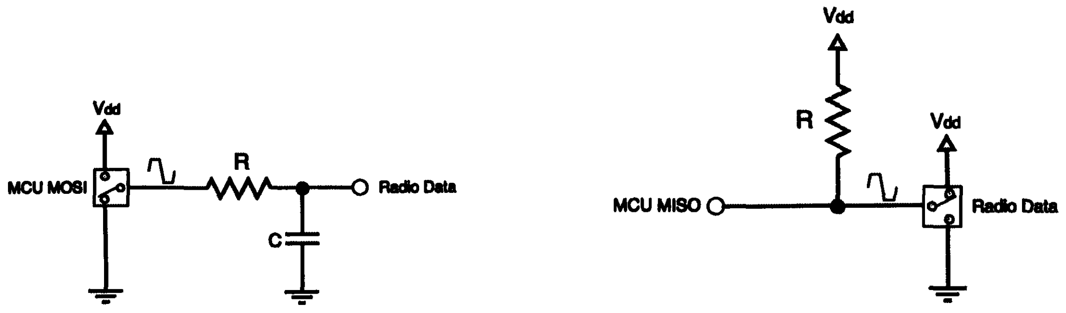
The preferred strategy was to build a custom radio system with the fastest low-power transceiver available. At the time, this was the Nordic Semiconductor nRF2401A, a 2.4GHz device designed for data rates up to 1Mbps. The nRF2401A can be controlled through an SPI-like interface, supports simultaneous reception on two channels, and handles addressing and cyclic redundancy check (CRC) computations internally. It consumes only 13mA in transmit mode, 20mA in receive mode, and 12μA in standby, while providing a reliable range of approximately 50 feet, depending on the surroundings. The nRF2401A has some disadvantages, including the inability to monitor a carrier-sense signal for manually determining when a channel is busy. Fortunately, this feature is not critical here, as the communications protocol will be low-latency and time synchronized such that dropped packets are preferable to uncertain delays. For ultra-compact designs, Nordic also offers the nRF24E1, essentially the same 1Mbps RF transceiver combined with an 8051 microcontroller and 10-bit ADC. Combining the radio and MCU into one package offers many advantages for small low-power systems, but here the flexibility and increased processing power afforded by a dedicated and more capable MCU was favored here.

The radio module was designed on a small daughtercard physically separate from the main node board. This choice was favored for two reasons, signal integrity and interchangeability (see Figures A-3,A-11,A-12). The sensitive nature of RF circuitry is such that it is preferable to isolate the radio from the rest of the system to minimize interference. In the case of the Nordic module, the datasheet also provides detailed layout guidelines, including matching network and ground plane arrangements, which are easily reproduced as a distinct unit. As far as interchangeability is concerned, the nRF2401A has already been superceded by the potentially more flexible 1Mbps Chipcon CC2400 and the new 2Mbps Nordic nRF24L01 during the course of the development of this project. Judging by this trend, capabilities of low-power radios will continue to improve rapidly for the foreseeable future. By encapsulating the radio module on a replaceable daughtercard, the sensor node can adapt easily to higher network speeds as the possibilities grow.

Communication between the main board and RF daughtercard is achieved through a three-wire SPI interface and several other digital control lines. Although it was considered important to provide a full SPI interface for the sake of future radio upgrades supporting SPI, the nRF2401A does not use true SPI, as suggested above. By far the most cumbersome departure from SPI is the fact that the Nordic radio has only one data line serving both incoming and outgoing data. Normally, there are two separate data lines for incoming and



(a) Merging SPI data lines.



(b) Equivalent circuit during MCU transmit.

(c) Equivalent circuit during MCU receive.

Figure 2-10: Modified SPI interface for Nordic radio.

outgoing data, one driven by the master *(MOSI)* and one driven by the slave *(MISO)*, as well as a synchronizing clock signal. In this way, the structure of the Nordic interface does not resemble SPI. However, unlike standard bi-directional communication interfaces such as I2C, which rely on the transmission of request and acknowledgment data packets to arbitrate the channel as a bus, the Nordic interface uses a separate control line to determine the directionality the data line. Thus, the actual stream of bits transferred to and from the radio on the data line together with the associated signal on the clock line resembles SPI more than any other type of transfer. Building a custom interface to match this arrangement to the MCU requires the potentially messy implementation of bit shifting and clock generation algorithms in firmware. Instead, the convenience of using built-in SPI hardware to generate the necessary data and clock signals is retained with a slight provision. At the expense of communication speed and power, the two MCU data lines can be logically merged into the single radio data line by bridging them with a resistor.

Figure 2-10 shows the approach taken, as well as equivalent circuits for data transmitted to the radio and data received. It can be seen that when the MCU drives the data line during the transmission cycle, the signal path is loaded by a lowpass filter with a time constant linearly dependent on the resistance R and the parasitic input pin capacitance C. The upper limit on baud rate during transmission is therefore inversely proportional to R.

At the same time, the receive configuration enforces a lower limit on R. Because of the structure of SPI hardware, the master device controls all transactions by driving the master out line as well as the clock. In other words, the master transmits an arbitrary byte in order to receive a data byte from the slave. Since the master out line is always being driven, a contention is created during the receive cycle, when the Nordic radio attempts to drive its data line at the same time. This is the situation shown in Figure 2-10(c), where a direct current path from power to ground occurs through the resistance R whenever the master tries to transmit a different bit value than it is receiving. In order to control the situation as much as possible, the MCU (the master in this case) is configured to transmit a series of ones when it wants to receive, so that master out is always pulled up to logic voltage, as implied in the diagram. In this way, when the direct current path opens, the MCU will always source the wasted current, not the radio, whose I/O pins may be less capable. The amount of current lost when receiving data is linearly proportional to R. Assuming for simplicity that 1s and 0s are equally likely in the received data, there is an average lost current of

$$I_{avg} = \frac{V_{Logic}}{2R}$$

while data is being received.

For the 10 kΩ resistor chosen in this design, the calculation predicts a lost current of roughly 0.2mA, which is suitably low. At the same time, experimentation suggests that the maximum possible baud rate for reliable transmission with this resistance is roughly 400kbps. This means data traveling upstream from the sensors to the transmission buffer in the node radio module is limited to less then half the rate of the potential 1Mbps network speed. However, the nodes can all perform this transmission step simultaneously, and the effective data rate at this stage is Nx400kbps, where N is the number of nodes. Because of this, 400kpbs is an acceptable baud rate for communication between the Nordic radio and MCU. Of course, the real bottleneck occurs later in the wireless channel and at the basestation receiver, where it becomes clear in Section 3.3 that the bandwidth in practice is still limited to less than the full data rate of 1Mbps.

Luckily, the question of how to adapt this limited hardware modification to increased bandwidth requirements in the future is no longer relevant, as the newest Nordic and Chipcon high bandwidth data radios all provide standard SPI interfaces. When the time comes, the bridging resistor added above can be removed from the board, and a jumper can be placed which restores traditional SPI wiring to the daughtercard, as implied in Figure A-1. 1Mbps SPI baud rates are certainly possible with the MSP430, but the datasheet suggests the baud rate can be set as high as 4Mbps. At such a high rate, trace capacitance may limit reliability, but the implication is that SPI communication rates between the MCU and radio daughtercard are unlikely to become a limiting factor with these sensor devices, even as the radio module is upgraded.

2.7 Power Circuitry

The major design choices for sensing and communication were made with only basic concerns towards minimizing power. Once these decisions were made, the power budget was more closely considered for the purposes of designing the appropriate regulation circuitry and choosing the best power source. Table 2.2 shows the breakdown of the major electrical components and their typical power requirements in the intended application. Traditionally, it is beneficial to pay close attention to the amount of time devices are on, and to reduce power consumption by duty cycling. Several factors conspired to limit the amount of temporal power management considered in this design. The largest power drain comes from the gyroscopes. The sensors will be sampled at 100Hz, or every 10ms, yet the gyro has a long wake up time, 35ms, which prevents it from being duty cycled between samples at this rate. By contrast, the accelerometers are very low power, and it was considered unnecessary to duty cycle them. Originally, because the inertial sensors were not duty cycled, the TLV2474 amplifiers making up their signal conditioning circuitry were set to run continuously as well. However, it can be seen that the sheer number of op-amps being used contributes to a significant power draw that could potentially be reduced by duty cycling in future designs. The TLV2474 has a wake up time of 8.3μs, which could allow an analog channel to settle and be sampled in 100μs or less. With a 10ms sampling period, this permits a 1% duty cycle, or a 99% reduction in the power consumed by the amplifier circuitry.

Other power hungry components were typically restricted by the communication protocol requirements. For instance, the radio is the third biggest potential power drain, with the highest peak current requirements of any device on the board other than the LED. During normal operation, the RF communication protocol strictly limits the amount of time the radio is in transmit or receive modes, and consequently the average power required is only 9.3mW. However, outside of normal operation, notably when not in range of the basestation, the RF protocol makes the radio to wait in receive mode indefinitely. In this state it consumes its peak power of 66mW. Thus, the radio is not strictly duty cycled, but its power consumption is managed when possible as arbitrated by the basestation. The LED plays a less critical role, but could potentially rival the gyros in terms of power draw if left on full brightness continuously. Much like the radio, control over the LED is bound to the communication cycle, as timing resources on the MCU are too limited to drive its duty cycle independently. In order to make the LED visible, it ends up being turned on continuously for most of the cycle, and its current draw is controlled through a resistor network as metioned above in Section 2.5.3. However, in this case it is simple to reduce peak power requirements by using a very dim setting or turning off completely when the communications link is lost, or whenever the radio requires its full share of current draw. Essentially, the operation of the LED is arbitrary, and its use can be moderated by the designer. The power requirements shown in Table 2.2 correspond to a typical situation in which the LED shines yellow during transmission of data and drops down to dim blue when the radio switches to receive mode to listen for the next basestation broadcast.

Based on typical operating conditions in this application, the average power requirement is estimated to be 211mW (see Table 2.2). Peak current draws amount to approximately 24mA from the 5V supply and 52mA from the 3.3V supply, equating to 292mW peak power consumption. As the system is intended for a performance, game, or training session lasting

Device	Function	Voltage	Current Draw Per Device or Channel	Number of Devices or Channels	Power Cycle (% ON)	Peak Power Consumed	Average Power Consumed
Analog Devices ADXRS300	Gyro	5V	6mA	3	100%	90mW	90mW
Analog Devices ADXL210	Accel	5V	0.6mA	2	100%	6mW	6mW
Nordic nRF2401	Radio	3.3V	Rx: 20mA	1	7.6%	66mW	9.3mW
			Tx: 13mA		10%	33mW	
			Standby: 12μA		82.4%	0.04mW	
Texas Instruments TLV2474	Op Amp	5V	0.6mA (per channel)	8	100%	24mW	24mW
Linear Technology LTC6221	Op Amp	3.3V	1mA (per channel)	2	100%	6.6mW	6.6mW
Texas Instruments MSP430F148	MCU	3.3V	3.8mA	1	70%	12.5mW	8.8mW
Texas Instruments MSP430F148 ADC	ADC	3.3V	1.3mA	1	100%	4.3mW	4.3mW
Fairchild QTLP650D	RGB LED	3.3V	Red: 14mA max	1	92%	46.2mW	42.5mW
			Green: 6mA max		92%	19.8mW	18.2mW
			Blue: 5mA max		8%	16.5mW	1.3mW
					Total:	292mW	211mW

Table 2.2: Power budget.

several hours, a suitable power source would have a energy capacity of at least 1000mWh and the ability to supply a peak power of 300mW. Since any small battery will be discharged within hours under this heavy usage, only rechargeable batteries were considered to avoid generating a small mountain of waste. Some AAA-size NiCd and NiMH cells approach the current capacity required, but are typically heavy and possess an inconvenient form factor for wrist or ankle mount devices. Lithium coin cell batteries would be more convenient, yet the highest capacity coin cells can't even come close to supplying the peak current draw required here. The lightest, most compact, and most powerful technology available is the lithium polymer (LiPo) cell. Two tiny 145mAh LiPo cells can be combined in series to provide 1073mWh of energy (at 7.4V), in a pack measuring only 3x2x1cm and weighing only 9 grams. A longer, slimmer 3.7V 350mAh cell, measuring 5.2x3.3x0.3cm and weighing 10 grams, can provide 1295mWh. Both battery packs can discharge at 20 times their rated current capacity, and so can easily handle the peak power requirements of the system, while lasting for 4–5 hours during typical use. The flat, rectangular form factor is well suited to a small wearable device.

Compared to more conventional power sources, LiPo batteries do have disadvantages. They are expensive, for instance, the 7.4V 145mAh pack currently costs $15, about 5 times as much as a comparable amount of energy in NiMH cells. They have strict operating conditions concerning charge and discharge rates, violation of which could destroy the battery and possibly cause serious personal injury. Ensuring these proper charging and discharging techiniques requires a charging circuit set up specifically for a certain LiPo cell arrangement, and a voltage regulator equipped with low-battery sensing, both of which may increase the cost of the electronics in a system. Despite this, LiPo was considered the best choice for pushing the boundaries of power versus size and weight as much as possible in this design.

Under the heavy power requirements imposed here, even the best batteries available begin to rival the size of the sensor device itself. Because the system is worn on the body during strenuous physical activity, it was decided early on that volume should be distributed more evenly by decoupling the battery pack from the rest of the electronics. Combining the two as a rigid unit would have added either too much extra height or too much inflexible contact area for a comfortable experience. Allowing the battery to be freely accessible also makes it much easier to recharge and replace when necessary. Since the sensor nodes are designed to be affixed to the wrists and ankles, a strap provides the natural structural element by which to wire a battery pack to the main board, in keeping with this idea. The short, thick 7.4V 145mAh LiPo pack is more easily attached to a flexible strap than any arrangement of longer 350mAh cells. Hence, the former option was considered the power source of choice. Decoupling the battery has another advantage, however, in that any number of different batteries could be used if desired, with little bearing on the rest of the hardware design. Providing for this flexibility was a major consideration in the choice of power management circuitry.

Conveniently, as the favored power source outputs 7.4V, both supply rails on the sensor node (5V and 3.3V) can be generated with step down regulators. Rather than use two wide input range regulators in parallel, the 3.3V supply is generated by a simple 5V to 3.3V low dropout regulator running in series off the 5V regulator (see Figure A-1). Dips in the logic supply voltage have been known to cause a perpetual reset state in MSP430 microcontrollers. Therefore, it was important to find a 3.3V regulator with a low-voltage error output, in order to properly reset the MCU in the event of a momentary power loss. Based on the peak current requirements assessed earlier, the 3.3V regulator must also handle at least 53mA. The Microchip TC1073, rated at 100mA, fits the requirements for only $0.90 (see Appendix B).

The 5V regulator is a more complicated proposition, as it must properly handle LiPo batteries. When discharging a single 3.7V LiPo cell, the voltage must never drop far below 3V, or the cell will permanently lose its ability to hold charge. To accommodate different LiPo arrangements, the regulator must have a low-battery sense input and shutdown with a variable voltage threshold, so that the circuit will be immediately shut down when voltage drops to the critical level. The step down design restricts battery selection to choices providing at least 5V, but to be able to operate with battery packs supplying more than 7.4V, the 5V regulator should also accept a wide input voltage range. Additionally, it must supply current to the 5V analog circuitry as well as to the 3.3V regulator input, a total peak draw of approximately 60mA. The Linear Technology LTC1474 was chosen to meet these requirements with the added benefit of high efficiency — according to the datasheet, 85–90% with a 7.4V input and 60mA current draw. It also allows inputs up to 18V, and could therefore be used with an 11.1V 3-series LiPo pack, or even a standard 9V battery. The low battery threshold can be easily set by a resistor to adapt to these different power sources. Rated at 300mA, the LTC1474 can comfortably supply the peak currents demanded by the design. However, the performance and flexibility of the LTC1474 comes at a price. The unit costs $7.50, and requires an extremely large external capacitor (100μF) and inductor (560μH) on the 5V output which claim board space and add additional cost (see Appendix B). In the future, the design could be greatly simplified by settling on one battery pack and dispensing with flexibility in the choice power sources. For example, the 7.4V 145mAh LiPo pack proved adequate in all situations explored by this work.

2.8 Structural Design

As a wearable device for strenuous activity, the sensor node has to be compact, sturdy, and lightweight. Since the intent was to strap sensor devices to the wrists and ankles of the wearer, the natural goal was to fit the form factor to that of a large wristwatch. The major physical components to accommodate within these guidelines are the main circuit board, RF daughtercard, two sensor daughtercards, battery pack, and strap.

First, the main board was laid out to occupy a 3.4cm square, which was considered to be an appropriate size for a rigid mass secured to the flat area of the wrist, and is slightly larger than a typical wristwatch. The main board houses the MCU and 8MHz crystal, one gyroscope, all analog signal conditioning, power regulation, and capacitive sensing circuitry, power connector, expansion port connector, and daughtercard connectors. The layout was designed on a four-layer circuit board, which includes an internal ground plane for signal integrity and an internal power layer to route 3.3V supply, 5V supply, and a reference voltage (see Figures A-6,A-7,A-8). The main board also includes three threadable screw mount holes for size #2 screws to secure it reliably to external packaging. These mounting

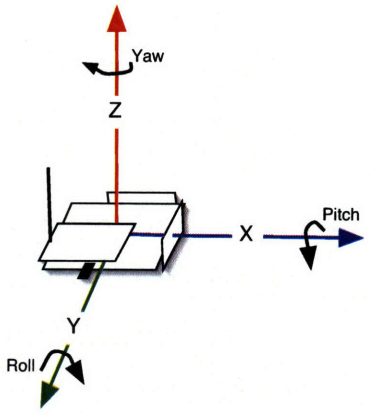


Figure 2-11: Axis conventions for inertial sensor data.

holes also serve as electrical paths to place ground in contact with the body and to connect the capacitive transceiver to an external electrode. Wherever possible, 0402 surface mount capacitors and resistors were used to make up for limited space, as evident in the bill of materials (see Appendix B). Still, much of the space on the external layers is taken up by traces, and adding additional internal signal layers would be the most dramatic way to reduce board area in the future. This option was simply considered too costly for prototyping.

The two sensor daughtercards are identical, each housing a two-axis accelerometer, gyroscope, and associated capacitors (see Figures A-9,A-10). The layout is on a small two layer circuit board measuring 2.5 by 1cm. The daughtercards must be mounted orthogonally with respect to the main board and to each other, in order to provide the three axes of inertial sensing (as well as one redundant accelerometer axis). This arrangement is achieved by using standard 0.001-inch 3x2-pin headers, which straddle two edges of the main board, as shown above in Figure 2-5. After the inertial sensor signals are inverted by the signal conditioning circuitry, the MCU receives data corresponding to the axis convention shown in Figure 2-11. Although the structural design prevents these axes from following a standard cross-product convention, the required inversions can easily be made in software.

The RF daughtercard designed for the Nordic nRF2401A radio is a simple layout adapted directly from the datasheet and Gerber files provided by Nordic. It is mounted on a raised

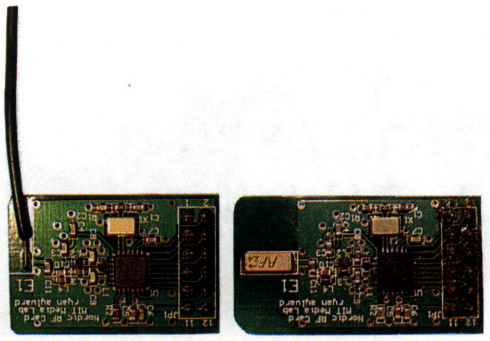
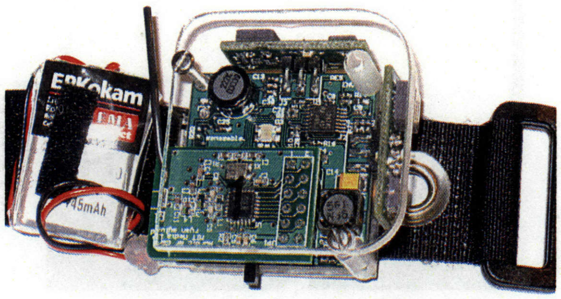


Figure 2-12: RF daughtercards comparing wire antenna to chip antenna.

header and lays across the lower half of the main board. The layout shown in Figure A-11 provides for a 2.4GHz chip antenna that was originally considered to be more durable than a simple quarter-wave wire segment. However, the chip antenna performed poorly and was eventually replaced by a wire segment soldered directly to the surface mount pad. Beyond greatly improved performance, the addition of the wire antenna also reduced board area. Originally, the chip antenna hung over the side of the main board to avoid being shielded by the ground plane beneath it. The wire antenna does not require a rigid mounting plane, and therefore the RF card could be trimmed flush with the main board (see Figure 2-12). In terms of durability, the simple wire is tried and true, and extremely cheap to replace. Chip antennas are no longer recommended for this type of design.

The bare sensor node complete with daughtercards measures 4x4x1.5cm and weighs only 10g (see Figure 2-3). It must now be merged with a strap that will secure the device to the body, as well as house the external battery pack. As mentioned previously, separating the battery pack from the sensor node provides flexibility and distributes the bulk of the electronics more evenly. In the ideal situation, a slim enclosure could be designed that secures the sensor and battery pack while grounding the body and routing the capacitive transceiver to an electrode built into a custom-made strap. This electrode could be designed using highly conductive fabric such as Bekiweave [58], for instance, or with flexible copper foil inside the strap. The prototype structure actually implemented is much simpler, relying on a sandwich of acrylic plates and an off-the-shelf Velcro strap, as shown in Figure 2-13. The


(a) Fully assembled sensor node.

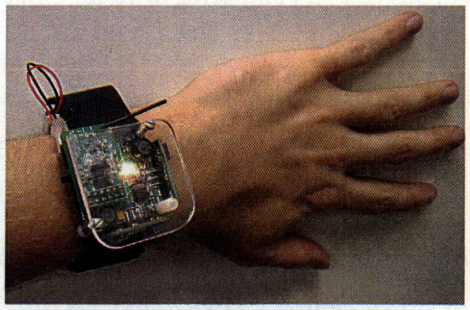

(b) Sensor node pictured worn on the wrist.

Figure 2-13: Finished prototypes.

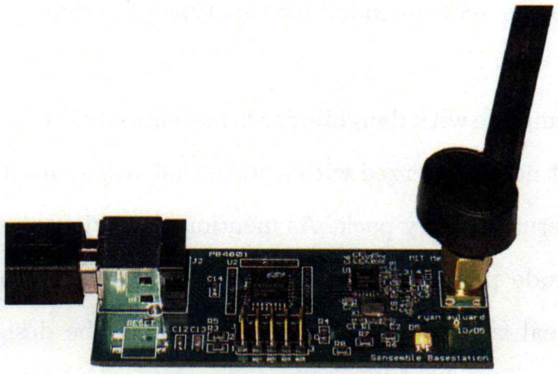


Figure 2-14: USB basestation board.

battery pack is secured to the strap with Velcro, which seems to be effective for most dance situations. In order to test the capacitive sensor, the body was grounded by wiring to a metal grommet on the strap, and simulation bracelet-sized electrodes were made with strips of copper foil (see Chapter 6). In the prototype enclosure, the node measures 4.3x4.3x2cm and weighs 40g including the battery. The use of thick acrylic and standard size spacers here dramatically increases the volume of the device, but a thoroughly designed enclosure could improve this. More importantly, the prototypes built for this project are sturdy and have proven their roadworthiness.

2.9 Basestation Design

The duty of the basestation is to control and synchronize the network, collect data from all of the nodes, and relay this information via USB to a central computer for processing. The design elements are simple, involving the Nordic nRF2401A radio module, an MCU with built in USB capabilities, power supply, status LED, and programming interface. Luckily, all of these components can operate for under 100mA, with attention paid to proper MCU and LED operation. Because of this, the basestation can be powered over USB as a standard device, greatly simplifying the design (see Figure 2-14).

In the wireless communications protocol, discussed in greater detail in Chapter 3, the basestation is required to transmit broadcasts at the sampling rate to drive the network. Because of this, the microcontroller must work quickly to pass data from the radio module to the USB interface without incurring delays that would disturb the timing. Since the speed of USB is so high (12Mbps at 'full' speed), most microcontrollers with built in USB capabilities are fast enough to meet this challenge. In this design, the Silicon Laboratories C8051F320 has been selected, a 25MHz 8051 MCU with built in USB controller, internal voltage regulator for delivering USB bus power, internal precision oscillator, and flexible digital I/O ports. With a supply current of 10mA when active at full clock speed, the F320 consumes more power than the MSP430, but is appropriate for a USB powered device. As before, part of the reason for selecting this MCU was previous development experience. Also, Silicon Laboratories provides the proprietary USBXpress API, a framework which makes it very easy to develop basic USB applications for Windows using the F320. However, the API places limitations on the type of USB transfers possible, a restriction which turned out to be more problematic than expected. Additionally, using the API makes it especially painful to communicate with the USB device from other host operating systems, such as MacOSX. These issues will be addressed further when host application software is discussed in Chapter 4.

Since the basestation is a single unit with a simple layout, it is easily replaceable. Because of this, the radio module was included on the main board, rather than using a daughtercard. The radio layout is otherwise identical, except the luxury of space allows for a larger antenna mounted on a threaded SMA connector (see Figures 2-14, A-13). As before, there are problems interfacing the Nordic radio with a standard SPI port. However, the F320 MCU provides the very fortunate ability to select which pins will be used for SPI communication on the fly. During transmission of data from the MCU to the radio, master out and master in can be shorted together to the radio data line with no added resistor. When receiving data, master out will attempt to drive the line, but it can simply be reassigned to a new pin which doesn't interfere with the radio. With no added resistance, the lowpass filtering effect that limited SPI baud rates on the nodes is no longer present. This change is critical, as the basestation receiver is a bottleneck, where the full 1Mbps data rate out of the radio will be called upon.

Chapter 3

Firmware and Data Flow

3.1 Communications Overview

Data generated by the sensor nodes must be transmitted to the central computer as quickly as possible to allow real-time processing to occur. In the case of music generation, delays of greater than 100ms are not only clearly audible, but can be disruptive to performers. Because of this, transmission latencies of greater than 30ms begin to leave insufficient time for processing. Additionally, the samples across all of the sensors in the network should be time synchronized and collected at a stable rate, to ensure that digital signal processing assumptions hold. As suggested previously, the best arrangement in this situation is a star topology, where a basestation controls all communication and timing on the network (see Figure 2-1). The Nordic radio is certainly capable of adapting to more complex scenarios, such as multi-hop networks where data flows between several nodes on its way to a destination. However, the typical reasons to do this — increased range and ad-hoc deployment of sensors — are not particularly relevant here. In this application, range is limited to a stage or other defined region, and the arrangement of the sensors can be predetermined.

In the protocol proposed here, which has been largely derived from an earlier design [46], timing and control of the network is based on broadcast messages sent from the basestation to all of the nodes at the sample rate of 100Hz. This imposes very strict synchronization,

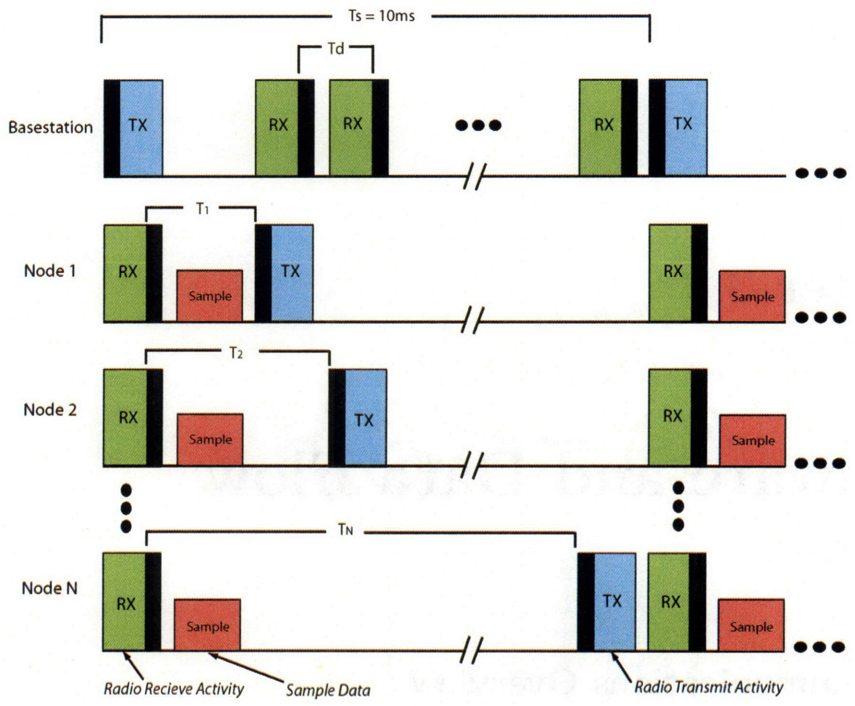


Figure 3-1: Basic illustration of the TDMA communications cycle.

and allows the nodes to share the communications channel effectively with a simple time division multiple access (TDMA) scheme. At the beginning of the cycle, each node listens for the broadcast signal. Upon receiving the signal, the sensors are sampled, and the data is transmitted back to the basestation after a preprogrammed time interval, as determined by each node's hard-coded ID number. Once the transmission has been sent, the node knows roughly when to begin listening again in preparation for the next broadcast. The basic procedure is illustrated in Figure 3-1. Besides low latency, the advantage of this protocol is that samples across the entire network will be taken essentially at the same time and precisely at the sample rate determined by the basestation. On the other hand, the integrity of the data depends on the ability of the basestation to make its broadcasts reliably. If the basestation stalls, the nodes will not begin to buffer samples in expectation that the communications link will be recovered. This type of recovery scheme could be added without too much struggle, but will be left for future implementations.

3.2 Radio Configuration

Although the Nordic radio can be configured at any point during operation, the process is somewhat time consuming, and for the purposes of this system can really only be done once, before sampling is initiated. Unfortunately, the possibilities are somewhat limited for a given configuration on the nRF2401A. In addition to selecting the carrier frequency, data rate, and other operating parameters, radio configuration assigns an address and a payload length for data transfers. Whenever a transmission is made, the address of the destination receiver must be specified. Thus, because the basestation needs to transmit broadcasts to the entire network, all of the node RF addresses must be the same. If the basestation then wants to send a message to a specific node, the request will have to be processed in firmware at every node receiver. Another limitation is the payload length, which must be the same for both transmit and receive paths. In this case, since the nodes must transmit a large data packet downstream to the basestation, the basestation is required to make its broadcast signal the same length, most likely resulting in wasted bits flowing upstream to the nodes.

In this implementation, the maximum required packet length from the nodes was considered to be 16 bytes, which fits a header byte indicating the node ID number or status, 6 inertial sensor values at 12 bits each, and 4 additional 12 bit values which might be taken up by capacitive sensor measurements or other information. The structure of a node packet is shown below in Table 3.1. Similarly, a 16 byte broadcast and control packet was designed for the basestation, with only 8 bytes potentially used, as illustrated in Table 3.2. The first byte in the broadcast packet signals an operating mode, allowing the system to wait in an idle state, for instance, before collecting data in sample mode. Other modes were implemented for additional functionality described later in Chapter 7. The next two bytes are used for an optional timecode that can be stored on the node in the case of logging data, an option which is also further explored in Chapter 7. The fourth byte is used to set the intended destination of a special message, in order to allow control messages to be passed to individual nodes. A message intended for all nodes is indicated by a hex value 0xFF in the fourth byte position. The remaining four data bytes are reserved for other messages,

Header 1B	Payload 15B						
Node ID	Inertial Data 9B						Other Data 6B
0x01–0x19	AccX	AccY	AccZ	Gyr Pitch	Gyr Roll	Gyr Yaw	Capacitive Measurements
0xF5	Node Idle, Payload Empty						

Table 3.1: Node packet structure.

Mode	Header	Timestamp	Target ID	Message Code	Message			Unused
Sample	0xAF	0x0000–0xFFFF	All Nodes 0xFF	Set LEDs 0x01	R 0x00–0x04	G 0x00–0x04	B 0x00–0x04	
			No Message 0x00					
Idle	0xFF		One Node Node ID 0x01–0x19					

Table 3.2: Basestation broadcast packet structure.

for instance, setting the color of the LED.

During wireless communication in either direction, the radio handles packets with a 16 byte (128 bit) payload, as well as a preamble, address, and CRC. For the most reliable operation in the presence of noise, the maximum lengths of 40 bits for the address and 16 bits for the CRC were used. As the preamble is set at 8 bits, this equates to a total packet length of 192 bits. The Nordic radio employed here is configured to use ShockBurst™ mode, which allows a data rate of 1Mbps, but requires a setup time of roughly 195μs to transmit a packet. Therefore, the effective data throughput for this configuration is only about 330kbps, or a 67% loss to overhead. In practice, the setup times during node transmission can be overlapped to pack the transmissions as closely together as possible, provided that the receiver can run CRC checking and clock data out fast enough. This can help to recover some of the loss, but more inefficiencies crop up in processing time for received messages at both ends of the RF channel.

3.3 Node Operation

This section provides a more detailed look at the firmware running on each sensor node and its relationship to the communications cycle. For reference, Figure 3-2 provides an illustration of the operations that must be carried out on several nodes, in the context of the basic RF communications timeline depicted above in Figure 3-1.

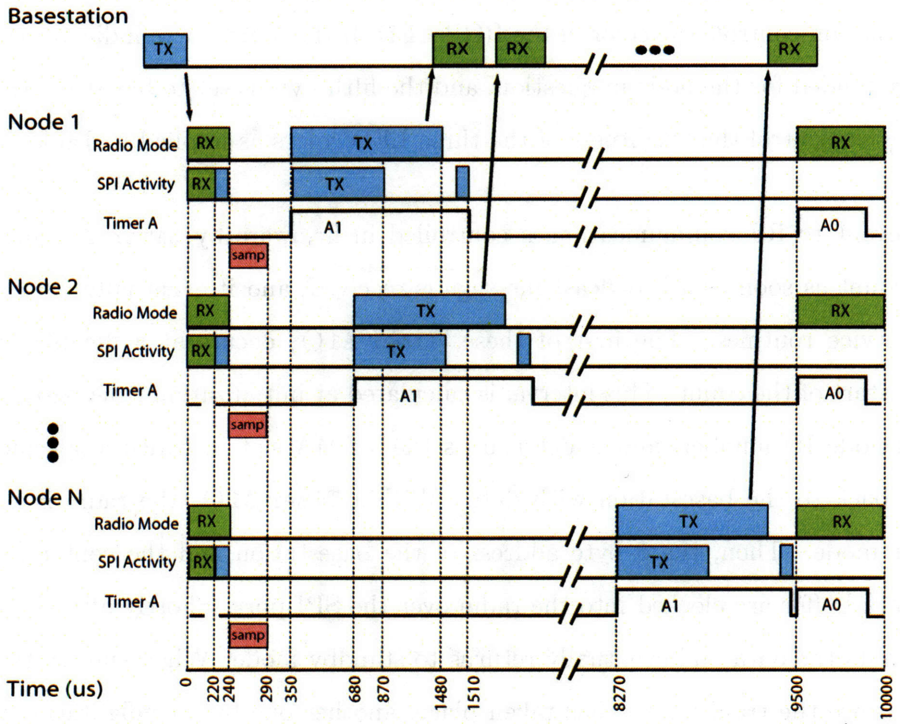


Figure 3-2: Detail of network timing cycle.

Initially, the node firmware is configured to hold the radio in receive mode until it receives a broadcast message. Once a signal from the basestation has been received, an interrupt triggers `nRF_data_waiting()` (see Listing C.5). At this point, the node reads the first 8 bytes of the message out of the radio and stores them in an array. Luckily, all 16 bytes of the received payload do not have to be clocked out of the radio, as this requires time on the SPI port, and the last 8 bytes of the broadcast message are never used. The SPI transaction with 8 bytes takes about 220μs. The mode byte is checked immediately after this interval to determine if it is time to enter sample mode. If so, all of the inertial sensors are sampled in a rapid sequence, and the first byte of the outgoing data packet is set to the node ID number to prepare for transmission. If the node has been told to wait in idle mode, the first byte of the outgoing data packet is set to hex 0xF5 to alert the basestation that the node has entered the expected state. Once either of these options is complete, the node sends a single configuration byte to the radio to prepare it for transmit mode, after which the radio waits in standby. Finally, the remainder of the broadcast received from the basestation is read for any special instructions. In the current implementation, the only special instruction controls the color of the RGB LED. If the fourth byte indicates that the message is intended for the node in question, and the fifth byte is set to hex 0x01, then the last three bytes control the brightness of the three LED colors as implied in Table 3.2.

Timing relevant to RF communication is controlled in firmware by `timer A`. This timer begins its count as soon as a broadcast message is received, and it eventually triggers two interrupt service routines. The first of these, `Timer_A1()`, occurs at a flexible interval beyond the start of the count. This interval is calculated at initialization time based on the hard-coded node ID number, and it determines the TDMA slot in which a specific node will report back to the basestation with data. Within `Timer_A1()`, the radio is first set to transmit mode. Then, the 5 byte address of the basestation and the contents of the outgoing data buffer are clocked into the radio over the SPI port. Shortly afterwards, the radio transmits the data and eventually returns to standby mode. When enough time has passed to ensure the transmission has taken place, another one byte configuration packet is sent to the radio to prepare it to return to receive mode the next time it wakes up from standby. This set of transactions takes approximately 1.16ms. The second interrupt routine, `Timer_A0()`, plays a brief but critical role. This interrupt is always set to trigger 500μs before the node expects to receive the next broadcast message, and when it is handled, the radio returns from standby mode to receive mode. This short window is crucial to limit the amount of time the radio is allowed to drain power waiting in receive mode, but is long enough to catch the next broadcast even in the case of significant clock drift.

During most of the 1.16ms block of transmit activity on one node, operations can be performed simultaneously on other nodes. In fact, it is absolutely critical that this overlap occurs, in order to pack the TDMA slots together as tightly as possible. The length of the TDMA slot, shown as T_d in Figure 3-1, corresponds to the delay between initiations of interrupt handler `Timer_A1()` on neighboring nodes. For the most part, the smallest delay factor possible here was determined empirically by observing received signals at the basestation, and was set to be $330\mu s$. However, the first node does not begin its transmission until $350\mu s$ into the communications cycle, to allow for necessary processing time.

Based on the $330\mu s$ interval required per node, $350\mu s$ setup time at the beginning of the communications cycle, $500\mu s$ buffer time at the end of the communications cycle, and 16 byte data packets, 25 nodes are able to share the channel simultaneously while sampling at 100Hz. If the entire 16 byte packet is considered data, the effective throughput of the sensor network is found to be:

$$25nodes \times 128\frac{bits}{node} \times 100Hz \quad = \quad 320kbps.$$

Assuming a 1Mbps symbol rate, this is a 68% loss of efficiency, as compared to the 67% loss inherent in the single radio transmission discussed above. Most likely, the improvement made by packing transmissions as closely together as possible was outweighed by the amount of time in the communications cycle spent not transmitting data. Although the loss in data rate appears drastic, this communications scheme is typically an improvement over what could be achieved with standard protocols and has the advantage of low latency, with a maximum one sample delay to the basestation.

3.4 Basestation Operation

The duty of the basestation is to control the network, and to pass data seamlessly between the wireless system and the USB connection to the host computer. Using a timer interrupt and the service routine `Timer0_ISR()` (see Listing C.13), this basestation generates regular broadcast messages at the desired sample rate, as described above. Each broadcast has an incremental 2-byte timestamp value associated with it, which could be used to locate gaps in the data up to 10 minutes long (at 100Hz broadcast rates). Once a broadcast has been made, assuming that the devices are operating in sample mode, the basestation must be prepared to store a barrage of data as the nodes send back their responses in the TDMA sequence. During this data collection phase, there will be little time to perform other operations. Yet, at some point during the communications cycle, the data collected must be transferred to the USB pipe so that it can be clocked out to the host. The only time this can happen is during the processing phase between a broadcast and the resulting response from the first node. Thus, the timer routine `Timer0_ISR()` is also used to drive USB communication.

Currently, the firmware running on the basestation uses the USBXpress™API, available for Silicon Laboratories MCUs. This set of functions makes it easy to implement USB communication, but with certain limitations. For instance, only bulk transfer mode is allowed. Although the packet size and data rate limitations of bulk mode do not pose a problem in this design, bulk transfers can only be initiated by the host, and take a low priority to other OS events. In a situation where data must be transfered at regular intervals determined by the device, interrupt transfer mode, which allows the device to place transfer requests, would be preferable. In order to use bulk mode successfully in this design, the host must place requests for data frequently enough to prevent buffer overflow, and the basestation firmware must have knowledge of these requests, so that it does not move data to the USB pipe before a request has been made [1].

The request for data is first processed in the USB interrupt handler, `USB_API_TEST_ISR()` (see Listing C.13). For some reason, an interrupt cannot be linked directly to the host's request for data, but only to the receipt of a USB packet. Because of this, the request for data is encapsulated in a 6 byte control message sent by the host at the desired USB packet rate — at least once every two periods of the RF communication cycle (see Table 3.3). Similar to the basestation broadcast message, the host control message can be used to set modes of operation on the basestation or nodes. In fact, most of the host control

[1]This may not be the case when the USBXpress API driver is used in Windows, which appears to allow bulk transfers to act like interrupt transfers, through a buffering mechanism that was not fully investigated. However, a hack using open source drivers in Mac OSX requires this, refer to Chapter 4.

message becomes the broadcast message used by the basestation until a new control message is received. In terms of software design, this is not a great level of abstraction, as the host programmer needs to know protocol specific to node firmware. In this case, however, it is simply the quickest way to convey information from the host to the nodes.

Request	Header	ID Number	Message Code	Message		
Sample Mode	0xAF	All Nodes 0xFF	Set LEDs 0x01	R 0x00–0x04	G 0x00–0x04	B 0x00–0x04
		No Message 0x00				
Idle Mode	0xFF	One Node Node ID 0x01–0x19				
Local	Message for basestation not passed to nodes.					
None	No new message.					

Table 3.3: Host control packet structure.

A host control packet can be received at any time during the communications cycle, since the host is not synchronized to the basestation in any way. Because of this, processing is kept minimal within the USB interrupt handler. If the control packet indicates a request for data, a flag is set, and the broadcast packet is set up to indicate sample mode. At the beginning of the next RF communications cycle, initiated by `Timer0_ISR()`, the basestation makes the broadcast, checks the USB flag, and finally calls `Block_Write()` to fill the USB pipe with previously stored data (see Listing C.13). Only then does the host receive the requested data packet. On the other hand, if the control packet does not indicate a request for data, the flag and broadcast packet can be set so that no data is returned from the nodes, and no call to `Block_Write()` is made.

Once the USB transfer to the host has been initiated, the basestation has time to handle the new batch of incoming sensor data triggered by the latest broadcast message. Each packet received triggers a call to the `RF_data_waiting()` routine (see Listing C.13). During each 330μs TDMA slot, the data is clocked out of the radio over SPI at 1Mbps and is stored as quickly as possible into a large (900 bytes max) buffer. This buffer will eventually be flushed to the USB pipe for transmission to the host, which may not occur every broadcast period. Because of this, it is made to store multiple samples from all of the sensors, and uses the headers and timestamps illustrated in Table 3.4 so that the data can be parsed efficiently at the host. Still, 900 bytes allows just 2 full samples to be stored for a 25-node network using 16 byte data packets, hence the requirement that USB packets should be requested at least once every two periods of the sample rate.

Packet Header	Node Header	Times-tamp	Length	Node ID	Payload	Node Header	Times-tamp	...
0x1F	0xAC	0x00–0xFF	typically 0x10	0x01–0x19	X	0xAC	0x00–0xFF	...

Table 3.4: USB data packet structure.

3.5 Host Requirements

One of the major concerns at the host is sample latency, and how it will effect the processing requirements for generating real-time feedback. Here, sample latency refers to the delay between the sample capture time and the response, not between the request for samples and the reponse. Still, the negotiation process at each stage of communication adds some latency to the protocol. The RF transfer to the basestation has an unavoidable worst-case delay of almost one sample period, or 10ms at the sample rate of 100Hz. In addition to this, the USB transfer to the host requires that data wait for as long as 1.35ms for a USB cycle period of under 10ms, or 11.35ms for a cycle period of under 20ms. Accounting for a small USB transmission latency as well, total delays of up to 22ms must be expected. To prevent buffer overflow at the basestation, the host must send requests and attempt to read the USB buffer at least every 20ms. Because the typical latency is longer than this cycle period, there will most likely be several attempts by the host to read data before it appears in the host buffer. In the worst case scenario that the requested data takes three read attempts to be collected by the host, and the host cycle is set to 20ms, another nearly 20ms of delay is possible. In practice, the host sends requests as quickly as possible, typically close to every 15ms, and the overall sample latency is closer to 30ms.

The degree of latency from sample to observation here places two significant stresses on the host application. First, with the goal of 100ms maximum total latency, there are only 70ms left for analysis and processing of the data. Second, if the software does not continually make USB read requests at least every 20ms, a buffer overflow may occur on the basestation, and several samples might be lost. This condition could easily occur while the OS is doing heavy processing and is asked to perform a GUI task, for instance, since the USB bulk transfers are very low priority[2]. A basestation firmware design allowing interrupt transfer mode would solve both problems, by allowing the host to respond immediately to new data, and by increasing the USB device handling priority. It would also promote cross-platform compatibility by getting away from the proprietary USBXpress API which can only be used officially with Windows. In this case, limited development time was the only attraction towards using the API in the firmware design, and host performance does in fact suffer as a result. The various limitations imposed on the host software will be addressed in more detail in Chapter 4.

3.6 Taking Measurements with the Capactive Sensors

So far, the discussion of sampling data has been limited to the inertial sensors. To say that a sample is taken on any of the standard sensor channels is a simple statement, as the ADC can handle this quickly and with minimal impact in firmware. However, to capture a data point with the capacitive node-to-node proximity sensor is much more complicated, requiring many samples taken for several cycles of the received waveform, and computation to extract the amplitude estimate from these samples (see Section 2.4). It is also important to schedule capacitive sampling in such a way that receive nodes sample their received signal precisely when a transmit node is transmitting, and only one node may transmit at a given time. At the carrier frequency of 90.9kHz, 30 cycles take up 330μs, the length of an entire TDMA slot. Thus, just obtaining the measurement is time consuming for both transmit and receive nodes. On top of this, the calculation for the magnitude estimate (Equation

[2]Again, this does not appear to be the case with the proper Windows driver, which buffers the USB data reliably even if the host application stalls, see Chapter 4.

2.4), which involves multiplication, accumulation, and a square root, is the heaviest of any computation required on the sensor node.

Given N nodes, there are $N(N-1)$ possible measurements that can be taken with the capacitive sensor system. Of these, half are unique node-to-node spacings, resulting in 300 unique measurements for a 25-node network. It would never be very worthwhile to transmit all of these measurements, especially since meaningful readings are limited to sets of nodes being worn by the same user or in very close proximity, as illustrated in Chapter 6. It also may be unnecessary to transmit capacitive data at the same 100Hz rate as the other sensors, because it is a noisy and usually slowly varying signal. Therefore, it is possible to design a strategy for a specific application that uses capacitive sensing sparingly but effectively, to limit the amount of resources required. For instance, capacitive measurements could be taken between wrists only, and the measurement could be alternated each sample period so that 5 pairs of wrists could be measured at 20Hz each rather than one pair at 100Hz. In the test arrangement designed here, however, the goal was to push the system and utilize the extra bytes provided in the data packet (see Table 3.1), without complicating the communications protocol. Towards this goal, as many capacitive measurements were made at the full rate as could feasibly fit into the existing timing structure.

The basic strategy was to transmit a pulse at the capacitive carrier frequency from a transmit node, and time several receive nodes to sample during the stable area of the pulse. Then, the next set of transmitters and receivers could be selected, and so on. Because of the TDMA scheme, at least one node is busy at almost any given time. Therefore, not all nodes are free to receive a transmission, and it seemed reasonable to select just a small family of receivers to handle each transmitter. This also makes sense because a node can only transmit up to 4 12-bit capacitive measurements per data packet. For simplicity, the pattern of transmitters and receivers was assumed to repeat on every broadcast cycle, thereby continually capturing the signals at the full rate. It also was most convenient to base the capacitive measurement cycle around the TDMA communications cycle, which has higher priority.

Unfortunately, it was difficult to realize this arrangement effectively, mainly because the

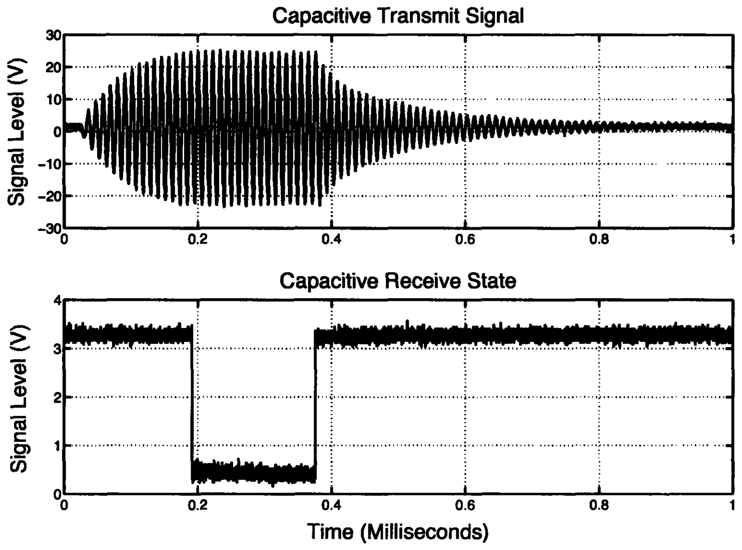


Figure 3-3: Oscilloscope trace showing alignment of capacitive transmit and receive phases on a pair of nodes.

transmit pulses ended up being much longer than a 330μs TDMA slot. The highly tuned LC resonator that allows the transmitter to generate high voltage signals also results in long ring-up and ring-down times, which limit the effectiveness of a short pulse. In order to get at least 10 steady-state cycles of the carrier signal to sample at the receiver, a 32-cycle transmit pulse was used, with the timing as shown in Figure 3-3. It is important to note that a capacitive measurement cannot be received accurately on any node until the transmit signal for the previous measurement has decayed. The length of the decay time shown here suggests that there must be at least 750μs between the beginning of one measurement period and the receive stage of the next measurement period. Since the receive stage is initiated 16 cycles into the transmit pulse, measurements must be separated by at least:

$$750\mu s - \frac{16cycles}{90.9kHz} = 574\mu s.$$

Because the measurements must fit within the communications cycle, a transmission pulse can only be generated at 660μs intervals, or every other TDMA slot. This means that at most, only half of the nodes will be able to act as transmitters. In future designs, the situation could be greatly improved by switching a load into the circuit to dampen the oscillator at the end of the transmit pulse. The need for such a provision was not anticipated early enough to be added to the current hardware design.

Another difficulty is the time needed to compute the magnitude estimate. The calculation required to process one capacitive sensor value with the algorithm in `cap_rx_calculate()` takes about 850μs, spanning three TDMA intervals (see Listing C.5). Also, there is only one set of memory buffers for quadrature samples, so calculations must be performed between every receive cycle. The sampling routine, `cap_rx_listen()`, which performs rapid quadrature sampling over 16 periods of the transmit frequency, typically takes 184μs to execute. Together, the entire receive process to pick up one capacitive measurement lasts for over 1ms. Bound to the TDMA cycle, the limit for samples collected on a single node is therefore just one every four TDMA intervals, or one every 1320μs. Because of the long computation times, capacitive receive measurements must also be scheduled carefully so as not to interfere with the timing of the RF protocol.

In the protocol implemented here, each node primarily concentrates on its RF transmission, to be sure that the timing of its reply within the TDMA cycle is accurate. Once RF communication has been initiated, the node runs a capacitive measurement sequence for as long as possible before it needs to prepare for the next RF broadcast. The timing diagram in Figure 3-4 illustrates the relationship of the capacitive sequence to elements of the communications cycle for two nodes. During the idle time when each node waits for its RF transmission to be processed, there is enough space to transmit a capacitive pulse with the `cap_tx_pulse()` routine, and to configure `timer B`, which then controls subsequent capacitive sampling (see Listing C.5). Only the nodes with odd ID numbers actually transmit pulses, however, as a 660μs delay is required between transmitters. Receive routines are scheduled in such a way that each node samples up to four other nodes, each separated by the requisite four TDMA intervals. Specifically, a node with ID number M is capable of listening for the signal transmitted from nodes with the ID numbers $M+4$, $M+8$, $M+12$, and $M+16$ for M odd, or $M+3$, $M+7$, $M+11$, and $M+15$ for M even. However, as these ID numbers approach the end of the TDMA sequence, it becomes impossible to complete the measurement before entering the next communication cycle, and if this is found to be the case, `timer B` stops its

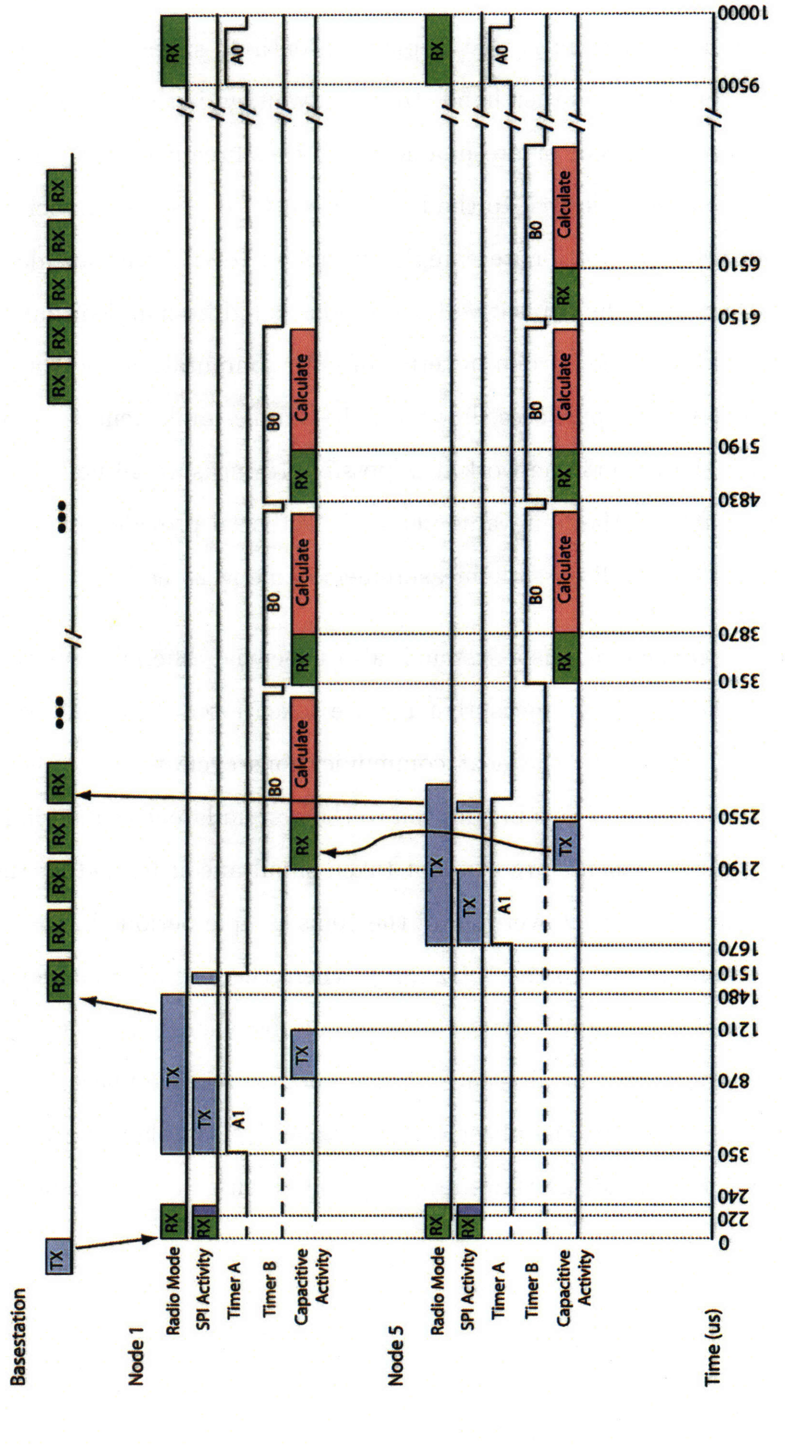


Figure 3-4: Detail of capacitive measurement scheme as related to the network timing cycle.

round of capacitive sampling. Then, the capacitive sensor data gathered so far is prepared for transmission to the basestation with the next outgoing packet.

The advantage of this system is that it plays an entirely subordinate role to the communications protocol. On the other hand, the disadvantage is that several nodes transmit a pulse that is never received, or are unable to fill the three capacitive measurements, and end up sending wasted bits back to the basestation. Of course, a number of improvements could be made with proportionate complexity added to the firmware design. One likely possibility is that all of the capacitive transactions could be handled, but at a much lower rate. These would ease the burden in terms of both computation and the quantity of data to transmit. For testing purposes, however, the simple implementation described here is adequate. With the 25-node network it is possible to measure 46 node-to-node couplings. This equates to 46% of the four value per node potential provided by the 16 byte packet structure, and 15.3% of all possible measurements among 25 nodes.

Latency with the capacitive sensor system is also somewhat different from that of the inertial sensors. Since the capacitive measurements are taken after data has been sent on to the basestation, they wait until the next communications cycle to be transferred. This can potentially cause a delay of 8ms in addition to the system latencies mentioned previously [3]. The capacitive measurements are also not taken simultaneously, and by their nature have phase offsets that can be well over half of the 10ms sample period. In this case, the signals are usually lowpass filtered to the 10Hz range where the increased latency and presence of phase offsets cause less trouble. However, in situations where capacitive measurements are taken more intermittently, or alternate between nodes, relative phase may come into play more dramatically. Luckily, the phase of each capacitive sample can always be inferred by the node ID and the broadcast timestamp associated with the inertial data it arrives with.

[3] 8ms is roughly the time between the first capacitive receive in a cycle and the sample capture event in the following cycle.

Chapter 4

Building Application Software

Application software for Sensemble developed down several paths in parallel in an attempt to satisfy different requirements at each stage of development. At various times, the goal was to log data from the network as a text file, to display visualizations of the data as it arrived, or to inject data directly into an environment providing audio and MIDI control for music generation. Each implementation was a rough attempt to meet these goals without too much concern with regards to advanced functionality.

Currently, Microsoft Windows™ is the only OS which supports the drivers and software libraries needed to properly interface with the USB basestation, since its firmware uses the Silicon Laboratories USBXpress API. This makes integration into Windows fairly simple and reliable. However, the MIDI control environment of choice, Max/MSP™, and other preferred music software, happened to be more readily available on a Mac. The prospect of designing a Windows-only USB device was also personally dissatisfying. Faced with indecision and a regrettably false sense of free time, basic working applications were developed on both platforms.

4.1 Windows

4.1.1 Using the USBXpress API

Application software developed in Windows can freely make use of the USBXpress API to establish USB communication with the basestation device, which is equipped with Silicon Laboratories' C8051F320 MCU. The API covers up many of the intricacies of configuring and enumerating USB devices, and provides a simple library of functions which make USB connectivity straightforward to implement in code. It also includes a set of templates for simple test applications built in Visual C++ or Visual Basic .Net. Of course, the decision to use USBXpress was not only motivated by convenience provided by Silicon Laboratories, but also support provided by colleagues who had previously developed applications using this framework. The software designed here for Windows is based on an earlier USB data-logging application built in Visual Basic. It primarily relies on only seven simple API functions to drive USB communication, as outlined in Table 4.1, and has been adapted for both data logging and real-time visualization.

Routine	Description
`SI_GetNumDevices()`	Returns the number of devices connected
`SI_GetProductString()`	Returns the serial number of a device
`SI_Open()`	Opens the device and returns its handle
`SI_Write()`	Writes data to the device buffer
`SI_Read()`	Reads data from the device buffer
`SI_FlushBuffers()`	Flushes host receive buffer and device transmit buffer
`SI_Close()`	Closes the device

Table 4.1: Host routines for basic USB functionality provided by the USBXpress API.

Before the application can run successfully, the basestation must be recognized by the system. In the Windows device manager, the device driver for the new unknown device should be manually selected and set to refer to siUSBXp.inf, the setup file provided by Silicon Laboratories. Then, the basestation should show up as a USBXpress device in the device menu.

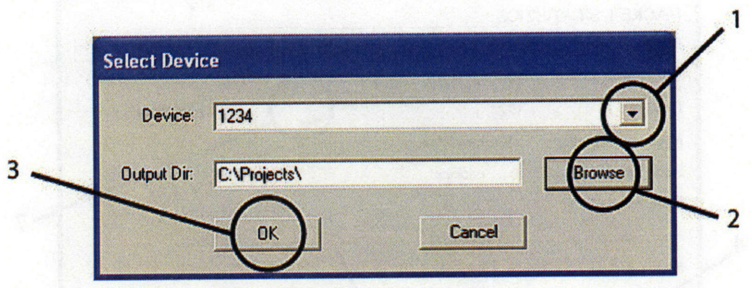
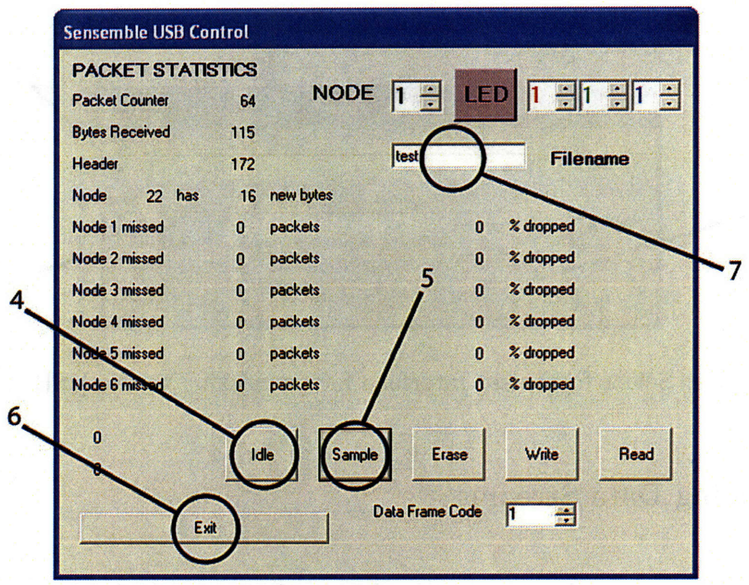


Figure 4-1: Select form user interface for connecting to the USB device.

4.1.2 Recording Data Streams

The most basic Windows application is one which collects data and records it as a text file. Execution begins with the select form shown in Figure 4-1 (see also Listing D.1), which scans for USB devices matching the description of USBXpress device. If any are found, their product serial numbers are returned. The basestation currently has serial number `1234`, which can be set in its firmware (see Listing C.13). The first pull down menu in the select form (1) allows the user to choose the correct device in case multiple USBXpress devices are attached. The 'Browse' button (2) allows the user to browse for a directory in which to store data logged from the device. Once a directory is chosen, selecting 'OK' (3) opens the main execution interface (see Figure 4-2) and initiates communication with the basestation (see Listing D.2).

The main interface is partially a debugging tool cluttered with output fields that still need to be organized or labeled. It also includes controls meant for the professional athletics application, which will be addressed in Chapter 7. For now, the important interface elements are the 'Idle' (4), 'Sample' (5), and 'Exit' (6) buttons. The meaning of these buttons is intuitive. When the application starts, it is in idle mode. First, it creates a new text file in the directory chosen at the select form, and opens a text stream to the file. By default, the file is named '`test.txt`', but a new filename can be created by changing the name in the 'Filename' (7) field. Once the text stream is open, further execution is based on the timer routine `Timer1_Tick()`, which tries to run every 8ms (it is often limited to 15ms by the OS). This routine continually executes a simple data collection parsing algorithm (see



```
PACKET STATISTICS
Packet Counter        64
Bytes Received       115
Header               172
Node       22  has    16  new bytes
Node 1 missed          0  packets        0  % dropped
Node 2 missed          0  packets        0  % dropped
Node 3 missed          0  packets        0  % dropped
Node 4 missed          0  packets        0  % dropped
Node 5 missed          0  packets        0  % dropped
Node 6 missed          0  packets        0  % dropped
```

NODE 1

LED 1 1 1

Filename test

Data Frame Code 1

Figure 4-2: Main user interface for logging sensor data to a text file.

Listing D.2). However, because it started in idle mode, the host passes 'Idle' packets to the basestation (see Table 3.2), and only 6-byte dummy packets (with node ID 0) are returned. These packets were meant for debugging purposes, and are ignored by the functional parts of the routine. As soon as the 'Sample' button is pushed, the host begins to feed the basestation with regular 'Sample' packets, and real data flows back from the nodes. At this point, the parsing engine does its work, generating text which is written to the output file. At any time, the 'Idle' button can be pressed to stop the flow of data again. Pushing 'Sample' then returns the system to sample mode, and also moves the previous text stream to a new output file with a number appended to the name in the 'Filename' field. Any time 'Sample' is pressed, the file number is incremented and a new file is generated, making it easy to create frames of data within a long recording session. Finally, pressing 'Exit' terminates the application, closing the last text stream and leaving the basestation and wireless network in idle mode.

During sample mode, the host potentially receives a data packet on every timer cycle. However, there is no synchronization between the host and the basestation, and the number of nodes in the network is unknown, so that packet is variable length. Because of its structure, a data packet also contains some packed 12-bit values that must be unpacked into 16-bit integers (see Tables 3.1 and 3.4). During its execution, the parsing engine writes to the text file byte by byte and saves the incoming data in the tab-delimited format shown below in Table 4.2. In addition to the ten sensor readings, basestation timestamps, and node ID labels, the host adds its own timestamp value, which corresponds to the time a sample arrived at the host. This feature was provided to help track delays which may not be reflected elsewhere, such as those which might have stalled the basestation or otherwise created a very long gap in the data. In fact, the host timestamp is the absolute system clock value in milliseconds, which means that even very deliberate, hour-long gaps between data frames can be accurately tracked to within several samples, and each data point is automatically tagged with a date and time. Since it is tab-delimited, the output text file can be imported easily into Matlab, Excel or similar tools for offline analysis. For example, Listing F.1 shows a Matlab script which reads the text file directly into a cell array structure, with each cell corresponding to a node present in the recorded data.

Node ID	Host Timestamp	Basestation Timestamp	AccX	AccY	AccZ	Gyr Pitch	Gyr Roll	Gyr Yaw	Cap 1	Cap 2	Cap 3	

Table 4.2: Tab delimited file format for recorded data.

4.1.3 Visualizing Data

Once data could be logged to a text file, the next step was to build an application to visualize data in real-time. This was simply done by adapting the structure of the previous application. Instead of writing lines of data to a file, the sensor values are continually stored to an array of ring buffers, which together hold the last 100 samples from each sensor on each node. At the end of each timer cycle, the buffers can be used to update strip chart graphics. This process was also implemented in Visual Basic, with graphics provided by the free ChartFX Lite™ package (see Listing D.3).

Shown in Figure 4-3, the user interface provides the same basic control with 'Idle' (8), 'Sample' (9), and 'Exit' (10) buttons. It also includes three charts, which from top to

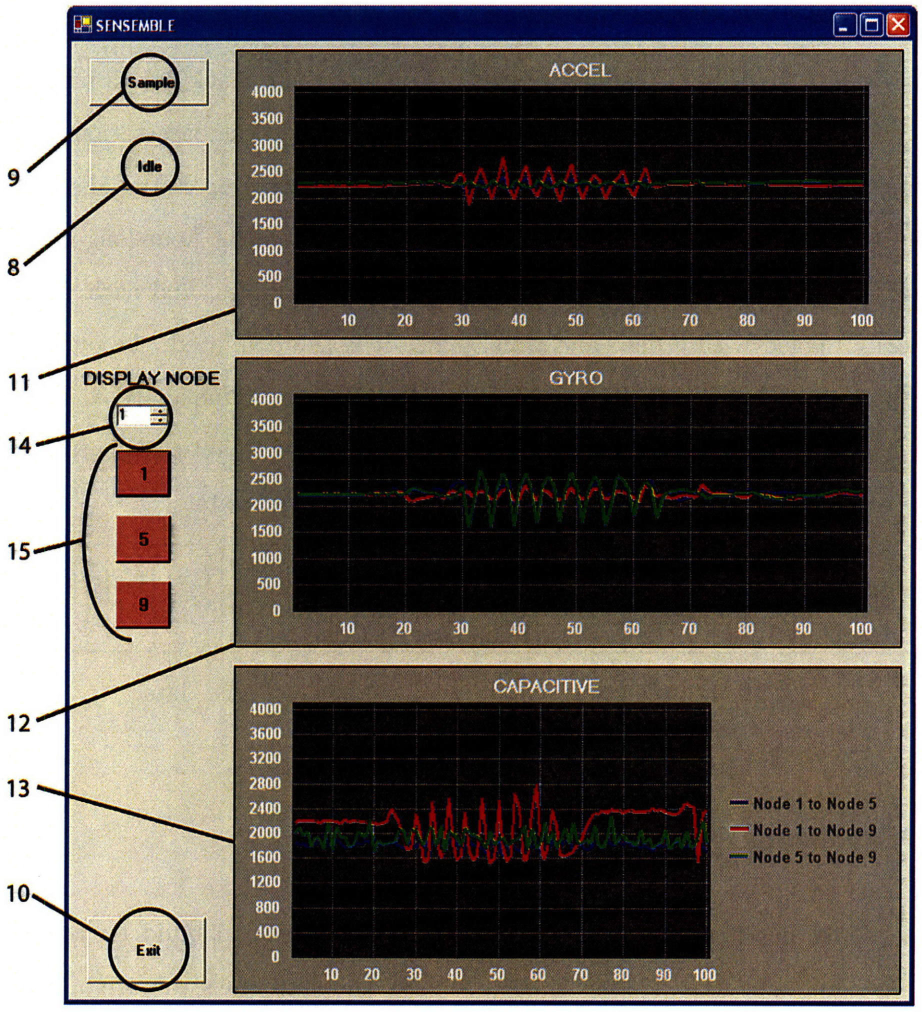


Figure 4-3: User interface for visualizing sensor data in real-time.

bottom display the three axes of accelerometer data (11), three axes of gyro data (12), and a subset of the capacitive sensor data (13). Because the capacitive sensor response is nonlinear, the display on the third chart is a log plot. To ensure that the charts are readable, only one node can display inertial data at a time. Any node from 1 to 25 can be selected by the menu field to the left of the chart displays (14). Alternatively, nodes 1, 5, or 9 can be selected with the quick access buttons below the menu (15). The reason 1, 5, and 9 are relevant is that they form a capacitive measurement family. That is to say, node 1 receives a capacitive signal from node 5 and node 9, and node 5 receives a signal from node 9, so all three edges between 1, 5, and 9 are measured. Since the strip chart visualizer was originally intended as a demonstration with just nodes 1, 5, and 9, the capacitive sensor chart only plots these three values, regardless of which node is selected to display accelerometer and gyro data. This is indicated by the legend to the right of the bottom-most chart.

4.2 Macintosh

4.2.1 Enabling USB Communication with the Basestation

In MacOSX™, the USBXpress libraries that made host application development so simple in Windows could not be used. However, similar functionality exists in an open source library known as libusb [59], which happens to be compatible with OSX. The libusb interface is quite similar to that of USBXpress, for instance, Table 4.3 displays a selection of available functions which parallel those in Table 4.1. If anything, libusb is intended to be more customizable, and is potentially a better solution. At the time the software described here was being designed, several colleagues had already implemented software to access the USB port in OSX using libusb. Some of their software even allows continuous data collection from custom built devices. However, in this case, the basestation firmware expects to communicate with proprietary USBXpress driver software on the host. Because of this, several modifications had to be made to adapt existing code structures to the needs of the project.

Routine	Description
`usb_init()`	Initializes libusb
`usb_find_busses()`	Finds USB busses on the system
`usb_find_devices()`	Finds all connected devices
`usb_set_configuration()`	Sets an active configuration
`usb_claim_interface()`	Claims an interface on the device
`usb_open()`	Opens the device and returns its handle
`usb_control_msg()`	Sends a control transfer to the device
`usb_bulk_write()`	Writes data to the device buffer via bulk transfer
`usb_bulk_read()`	Reads data from the device buffer via bulk tranfer
`usb_release_interface()`	Releases an interface on the device
`usb_close()`	Closes the device

Table 4.3: Host routines for basic USB functionality provided by libusb.

Luckily, connecting to the USBXpress device through libusb was simple. After initializing libusb with `usb_init()`, the first step is to open the correct device. Much in the same way as the select form in Windows searched for devices matching the USBXpress device description, here the application must search for a device matching the vendor ID, product ID, and interface ID provided by the programmer. This is accomplished with the `usb_find_busses()` and `usb_find_devices()` functions. Once the application has a handle to the correct device, it can be opened with `usb_open()`, and the interface can be claimed with `usb_claim_interface()`. On MacOSX, the active configuration also has to be set with `usb_set_configuration()` before claiming the interface. So far, this is no different from the standard procedure, and the basestation will respond as if it were opened correctly. However, requests to read or write data will cause an error.

In order to find out what was missing, a minimal amount of reverse engineering was required. In Windows, USB probing software was used to monitor the traffic to and from the basestation. Just after the standard control transfer for opening a device, an unidentified control packet was sent from the host with request type 64, request 2, value 2, index 0, and 0 bytes to transfer. With libusb, this packet can be sent manually using `usb_control_msg()`. If this is done immediately after claiming the interface, data will finally transfer successfully in both directions. One more modification is required, as releasing the interface and closing the connection in the standard way causes the basestation to hang. Just before usb_release_interface() and usb_close() are called, another manual control message must be sent, this time with the value field set to 4. The syntax for both messages is illustrated below in Listing 4.1.

```
//Opening the device
usb_control_msg(my_handle,64,2,2,0,0,0,100);

//Closing the device
usb_control_msg(my_handle,64,2,4,0,0,0,100);
```

Listing 4.1: Modified USB control messages for accessing a USBXpress device from libusb.

These modifications allow a USBXpress device to be accessed from MacOSX, but they do not succeed in totally simulating behavior under Windows. For instance, the basestation firmware stalls if the host is not ready to read data at the same time as it is being sent. This is the main reason for requiring the basestation to wait for a signal from the host before transmitting its data, as described in Chapter 3. Still, in MacOSX these troublesome stalls occur any time many processes are vying for CPU, and the exact circumstances of the error in firmware have not been determined. With the proper driver in Windows, it turns out that the basestation can initiate USB writes any time it wants. Originally, it was assumed that this data was actually moved to the 1kB USB buffer on the basestation MCU, later being retrieved with read request from the host. As such, a buffer overflow could easily occur. However, the USB probe revealed that apparently any data written from the basestation was automatically transfered to a much larger buffer in host memory. This happened without intervention from application software, so the system could even begin sampling without any application running. It was not entirely clear how the USBXpress API was able to accomplish this, although it may rely on interrupt USB transfers while professing to use bulk transfers. One other unidentified USB control message is passed upon opening the device, and it is speculated that this command might include an address in host memory for buffering. This message could never be simulated successfully using libusb, however. Further development will be required to streamline operation under OSX, but at this point making changes in the basestation firmware and avoiding the use of USBXpress completely might be preferable to further reverse engineering.

4.2.2 Recording Data Streams with Python

The first host application developed for OSX was an adaptation of an existing Python script for recording USB data to a text file (see Listing D.4). Access to libusb functions in Python is provided through another open source framework known as PyUSB [60]. With PyUSB installed, USB functionality becomes available after importing the usb module, which operates with methods that look quite similar to the libusb routines outlined in Table 4.3. As PyUSB and libusb are currently in the early stages of development, new versions are likely to require changes in the implementation presented here. For reference, the latest working setup uses libusb 0.1.10a and PyUSB 0.3.1, running on MacOS 10.3.9.

The execution of the script in Listing D.4 basically follows the same procedure as data logging in Windows, with the addition of the modified process for opening and closing the device as described above. Unlike the software in previous examples, this script is called from the command line, with options providing the output filename and mode of operation:

```
$> sudo python SensembleUSB.py [ log ] [ filename.txt ]
```

For now, the only operating mode is 'log', and the functionality is very basic. Sampling begins immediately, and rather than using a timed loop, requests for data and processing on receive packets occur in a `while` loop which simply runs as often as possible. The output file is text, with the same tab delimited format outlined in Table 4.2 above. In this case, however, the host timestamp logged in the first column is the time from the beginning of execution, not the system clock time. To stop sampling, `ctrl+c` terminates the `while` loop and the USB connection is closed appropriately, leaving the basestation in idle mode.

4.2.3 Access to Data Streams in Max/MSP

The main reason to pursue MacOS compatibility was the opportunity to merge sensor data with a suite of familiar MIDI and audio processing tools, making it easier to build off of previous work in the area of responsive sound. Thus, the focus shifted towards driving the USB basestation directly from within a music control environment, in this case, Max/MSP.

Max/MSP is a graphical environment in which functional blocks, or *objects*, are connected together to form a *patch* which operates on a stream of data (see examples in Appendix E). This graphical flowchart structure is particularly cumbersome for large array processing or conditional statements, which would be simple in a sequential language. However, if as much processing as possible can be encapsulated in object boxes, it becomes an intuitive way to manipulate continuous data streams. Most importantly, Max/MSP was made for music controllers, and it provides an intuitive way to harness MIDI and audio functionality.

Similar environments exist, most notably PureData (PD), which is freely available, and has largely the same capabilities. In this case, the choice was made partly based on experience and previous work, some of which called upon Max/MSP objects with no parallel in PD. For instance, the `rewire~` object allows Max/MSP to become a ReWire™ master with full control over other audio applications running on the system, such as Reason, Live, or Logic. In this project, the software synthesizers in Reason were called upon frequently for quick access to a rich sound palette.

Real Time Access

Several objects are available for transferring a stream of raw data from an external device directly into Max/MSP. Of course, none of these have the specific properties required to access the basestation, and in general there are few alternatives for any custom USB hardware. The `serial` object is too slow for USB rates, and control interface readers such as `hi` require that devices comply with the human interface device (HID) specification, which would add huge amounts of complexity to the firmware. Luckily, it is possible to program custom objects, or *externals*, in C, and detailed tutorials with example source code are made available in the Max/MSP SDK. As such, it is straightforward to encapsulate libusb functionality and the familiar USB polling and data parsing algorithms within a Max/MSP object that outputs real-time sensor data. The implementation presented here is an external object called `rawusb` (see Listing D.5).

One point not established above is that Max/MSP comprises two distinct packages — Max objects, which respond to single control messages at peak rates of 1kHz, and MSP objects, which continuously process data streams at the audio sample rate, typically 44.1kHz. Although the sensor data is a continuous stream, it arrives in discrete packets at rates no higher than 100Hz. Since there is no reason to generate output samples at audio rates, `rawusb` was designed as a Max external, and the output it generates is a stream of control values.

In order to develop an external object, basic conventions must be followed in the source code, including constructor and destructor methods to handle several instantiations of the object, syntax for data structures within the object, and specifics of the initialization process. These details are provided in the Max/MSP development tutorials and will not be described here. The more important aspects of the design are the algorithm and the interface which allows the customized object to be connected to others in a patch. Every object communicates to the outside world via *inlets* and *outlets*, which can be configured to handle various data structures. One such data structure is called a *message*, which is really any text string. Every object in Max must have at least one inlet which responds to messages, and typically this inlet is set to at least respond to a `'bang'`, which is the basic event message. Not surprisingly, inlets and outlets can pass single integers and floating point numbers, but of more interest here is the *list* structure, which in Max is something similar to an array. Since each data point returned from a node is an N-tuple of sensor values, the ability to process lists is very important.

In this case, input to `rawusb` was limited to the control functions necessary to start and stop sampling. Thus, the object was designed with a single inlet, which responds to the messages `'bang'` and `'stop'` (see Listing D.5). When a `'bang'` is received, the method `rawusb_bang()` is called, which starts the clock timer. During execution, the timer triggers `rawusb_tick()`, which contains the USB negotiation and data parsing algorithm, at intervals of 10ms. When the `'stop'` message is received, the `rawusb_stop()` method is called, the timer is halted, and the basestation returns to idle mode. If the request to stop has been processed successfully, a 'device ready' message is printed to the Max output window.

The details of USB operation are handled by libusb routines (Table 4.3), which are available as a C library, provided that the associated files are included in the build process. Traditional USB device open and close operations, as well as the workaround for connecting and disconnecting a USBXpress device successfully in MacOSX, are bound together in software with the `open_my_device()` and `close_my_device()` methods. A call to `close_my_device()` also sends the host 'idle' request message to ensure that the basestation and nodes are left in idle mode whenever the connection is terminated.

While data is being extracted during the timer routine, it can be parsed and organized by node ID, just as lines of text were generated in previous applications. In this case, the lines are not stored, but are clocked out sequentially as data arrives. The best way to do this in Max is to output lists, and therefore `rawusb` was designed with a single list outlet. The ordering of the list values is altered from the format in the received packets, however, to take advantage of Max-specific list processing behavior. Max places special emphasis on the first member of a list, which acts as a sort of tag for the rest of the structure. By placing the node number at the head of the list, the data points can be routed according to node very easily, for instance, with the `route` object. The output format and typical useage of `rawusb` in a patch is illustrated in Figure E-1.

Streaming from a Recorded File

At this point, applications have been presented which either record data from the sensor network or forward the data stream to Max/MSP, but there has not been a discussion about linking the two modes of operation, for example, recreating the data stream from a recorded file. This playback capability is especially useful for testing real-time feature extraction and interpretation algorithms repeatedly with the same data set, but in general it is valuable to be able to accurately reconstruct the time evolution of recorded movements in this way.

0,	id	[filename];
1,	[milliseconds].00	[data data data ...] ;
2,	[milliseconds].00	[data data data ...] ;
.		
.		
.		
N,	[milliseconds].00	[data data data ...] ;

Table 4.4: The `seq~` file format.

It was possible to accomplish this without actually developing new application software; it can be done with some preprocessing on the recorded text file and a simple Max/MSP patch. The format of the output data stream should be a series of lists with the same structure as the output of `rawusb` (see Figure E-1). The stream of output samples should not only be sequential, but should preserve the original timing of the recording. To store and access arbitrary control data in a time-synchronized sequence, the best existing solution is the MSP object `seq~` . Data is stored in a `seq~` as a text file with the format shown in Table 4.4, and must be clocked out with a counter signal running at the audio rate. When the counter reaches the millisecond timecode associated with an event, the data stored at that index is output.

The timestamps in the recorded sensor data text file can easily be converted to a millisecond timecode with floating point values. However, in the `seq~` format, every event line must carry a unique timecode. By contrast, the previously discussed text output format uses one row per node, and on each broadcast cycle data from many nodes has typically been recorded with the same timestamp. Thus, the information in the recorded text file has to be condensed again, into large packets with only one timestamp per packet. Then, the layout of the text file can be re-written to suit the `seq~` format.

The need to rearrange of timestamps in the recorded file provides an opportunity to compare the basestation and host time records for disagreements that could be corrected. For instance, because the host's data retrieval cycle tends to be slower than the basestation broadcast cycle, it is possible to record host packets containing two samples from the same node. Although these samples share a host timestamp because they appeared to arrive at the host on the same cycle, their true timing can be determined from the basestation timestamp, which should indicate which broadcast initiated each sample. When the data set is converted to **seq~** format and reconstructed as a stream, reporting two values for the same sensor at the same time could create a glitch. Instead, the host and basestation timestamps should be reconciled to produce a single corrected timescale. This correction process was implemented in Matlab, along with an additional feature to detect and fill gaps in the data with simple constant interpolation (see Listing F.1). Once the data from the original file is scanned into Matlab and processed, it can be written to a **seq~** format text file with another Matlab script, shown in Listing F.2, and then loaded into a **seq~** object in Max/MSP.

With the correctly formatted sequence file loaded, a clock signal must be generated to play back the recorded data at the correct rates. With the sequence running, the large blocks of data coming in at each sample can once again be broken down into smaller streams for each node, using the `route` object. Figure E-3 shows the basic patch structure used to load and play back the formatted text file.

This approach works well once the text file is loaded, but **seq~** is not meant to store huge files with tens of thousands of lines, which can be generated by the network in just a few minutes of recording. For this reason, loading the data before running the patch is extremely slow. In the future, a better solution might be to develop a custom object which is able to buffer data from a text file and process the sequence on a line by line basis.

Chapter 5

Feature Extraction

Now that the entire of process of designing the sensor system from hardware to application software has been established in detail, the discussion can turn to strategies for putting the data to use. Typically, this begins by generating features, parameters which are derived from a data stream for the purpose of isolating patterns. At a certain level, patterns combine to represent the qualitative statements one would like to conclude from the data. This chapter covers feature extraction — in particular, it highlights methods for the selection and interpretation of features with specific relevance to group movement. Most of the features discussed below were studied with dancers wearing the sensor system, in order to investigate what types of conclusions can be drawn in a dance setting. The results of this early experimentation, originally discussed in [61, 62], will also be presented in detail. Matlab scripts relevant to the analyses performed here are included in Appendix F.

5.1 Basic Inertial Features

In general, feature extraction starts with processing on single sensor signals to generate core statistical parameters such as mean and variance. This can be expanded in several directions to provide a foundation for interpreting inertial sensor systems. For instance, while gyroscopes are ideally zero-mean, accelerometers have a bias offset related to the direction of gravity. During periods of inactivity, when the orientation is stable, this bias offset becomes apparent. Therefore, a long-term running mean of the accelerometers should yield some degree of information about the average orientation of the device. Also, the derivative of the accelerometer signals can be calculated to determine jerk in three dimensions. Evidence suggests that the human body attempts to move in such a way that jerk is minimized [45]. Thus, it potentially becomes an important feature associated with the power or effort of movement. Similarly, variance is commonly used as an energy or motion activity measurement with wide applications as discussed in Section 5.2.2. Finally, the accelerometer or gyro axes can be collapsed by taking the magnitude, $\sqrt{x^2 + y^2 + z^2}$. Since the sensor axes are orthogonal, this magnitude represents a meaningful value, the length of the net acceleration or angular velocity vector in three-dimensional space. Any of these simple direct features can be used in various ways to study group movement.

5.2 Describing Collective Motion

The major advantage of having enough bandwidth to operate multiple sense points on multiple wearers simultaneously is the ability to obtain detailed information about correlated activity within a group. In the context of a dance ensemble, time and spatial correlations can be used to determine which dancers are moving together, mean tempo, which dancers are leading versus following, or perhaps which are responding to one another with complementary movements. Bulk parameters averaged across the entire ensemble can also be used to measure collective energy, net jerk, or the predominance of certain types of motion. Most of the analysis here focuses on simple features that can be used to characterize group dynamics with these goals in mind.

5.2.1 Measuring Temporal and Spatial Separation

The first task was to investigate features for quantifying both time separation and spatial similarity of gestures performed by multiple users. For initial evaluation purposes, the network was limited to three sensor nodes, each worn on the right wrist of three test subjects. In a pair of tests, subjects were asked to raise and lower their right hands, first simultaneously and then in sequence. In the last test, subject one raised and lowered a hand as before, subject two performed a qualitatively "similar" but distinct gesture, and subject three performed a qualitatively "completely different" gesture. To find the time separation between similar signals, cross-correlation is a natural choice. Here, the the similar measure of cross-covariance is favored, because of the bias present in the raw inertial data. Given two data segments of length N from different sources, the cross-covariance function of length $M = 2N - 1$ is defined as:

$$c[m] = \begin{cases} \sum_{n=0}^{2N-m-1} \left(x\left[n+m-N\right]-\bar{x}\right)\left(y[n]-\bar{y}\right) & \text{for } m \geq N, \\ \sum_{n=0}^{2N-m-1} \left(y\left[n+N-m\right]-\bar{y}\right)\left(x[n]-\bar{x}\right) & \text{for } m < N. \end{cases} \tag{5.1}$$

Hence, cross-covariance acts as crosscorrelation with the mean of the input signals removed. Early tests using this measure to explore correlated activity with wearable IMU platforms were presented in [46].

For each segment data in this test sequence, subject one was regarded as the reference and cross-covariance was calculated for the other subjects with respect to subject one. It can be seen in Figure 5-1 that the location of maximum cross-covariance provides an estimate for the time lag between similar gestures performed in succession. As might be expected, cross-covariance also appears to be useful for determining the time delay between disparate gestures, but with diminishing accuracy. The nature of cross-covariance as a signal-matching technique would suggest that the peak magnitude gives a measure of the strength of the correlation at the location of the peak. In the context of the inertial sensor data this translates loosely into a measure of spatial similarity between gestures. Indeed, Figure 5-2 illustrates that as the disparity between gestures increases, the height of the peak cross-covariance decreases. This is a satisfying result in that a cross-covariance calculation can be used to determine both the time and spatial correlation of group movements.

One problem with cross-covariance as a feature is that it requires a complete segment of data to calculate. The length of the segment also determines the maximum delay that can

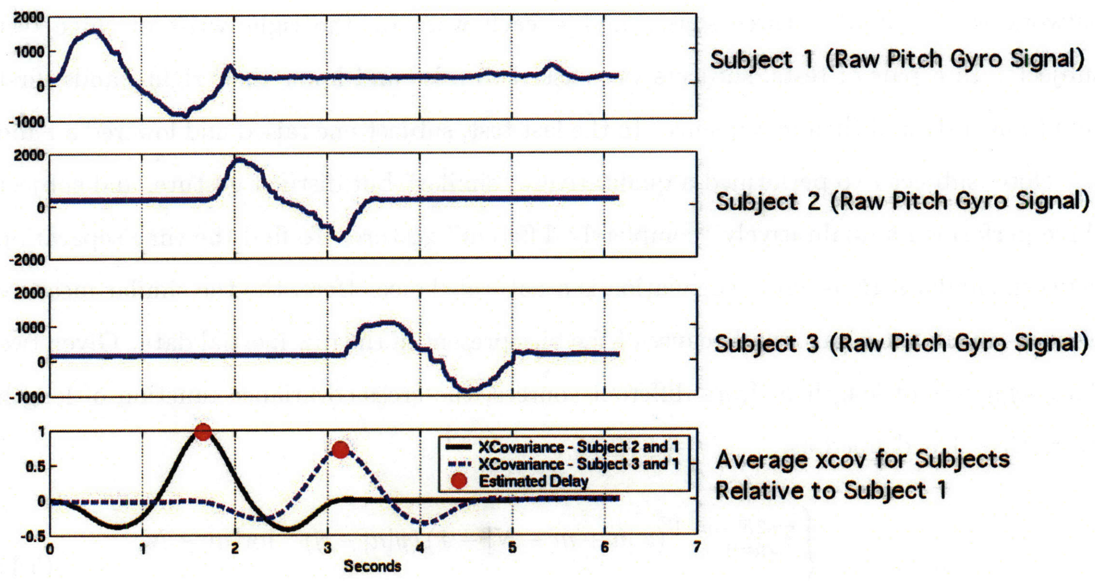


Figure 5-1: Raw data for hands raised and lowered in sequence and resulting average cross-covariance.

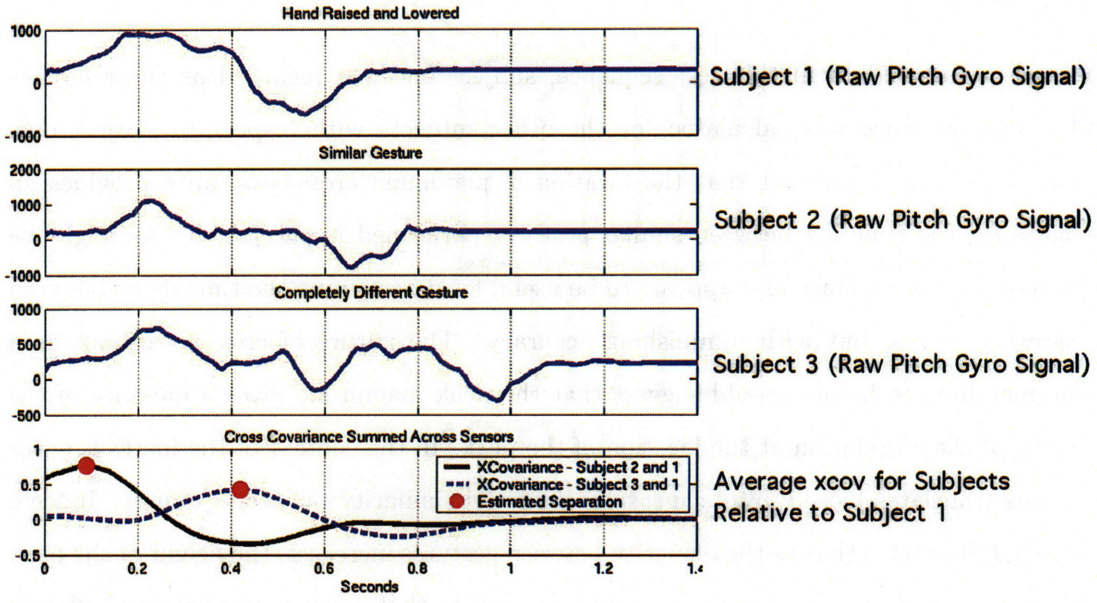


Figure 5-2: Raw data for gestures of decreasing similarity and resulting average cross-covariance.

be detected — the space between two events cannot be measured unless they have both occurred within the scope of the measurement. In a streaming situation, cross-covariance can be computed periodically on short windows of data. Window size is chosen to make a trade-off between latency and the maximum time separation that can be expressed. A window of length N samples handles time separations of $\pm N$, but also requires N samples before the calculation begins. Even then, the computation required to arrive at a result is intensive and scales with N, adding to the latency (see Section 8.2.1). Therefore, cross-covariance can never be extracted in real-time, but processing on windows less than a second long might still be useful for driving interactive content. For instance, when dancers synchronize to music or to a leader, the delays between their correlated motions are most likely less than a second, and the latency associated with cross-covariance might be manageable in this setting.

To test the effectiveness of a cross-covariance measure, six sensors were given to three dancers participating in a ballet lesson; each wore one on the right wrist and one on the right ankle. The class then performed an exercise involving a repeated sequence of leg swings executed in unison, to music. Although they were roughly in time with the music, the dancers were not necessarily looking at each other or at an instructor, creating a small but clearly visible delay in their motions[1]. Figure 5-3 shows a portion of the raw data collected from the leg of each dancer. Because there was very little arm motion associated with this exercise, only leg motion is discussed here. The area from about 35 to 65 seconds corresponds to the synchronized sequence of swings made with the right leg.

Figure 5-4(a) shows the result of windowed cross-covariance analysis on this data segment, computed using a window size of 1 second, step size of 0.25 seconds, and averaged across sensor axes. That is to say, at each interval of 0.25 seconds, data from the past second was considered, the cross-covariance vector was computed individually for each inertial sensor signal, and finally the individual vectors were averaged to produce the final result. Because the step size is small enough, individual leg swings and their synchronicity across the ensemble can be picked out. Note that the area of peak cross-covariance, shown in white,

[1]The dance rehearsal was documented on video for reference.

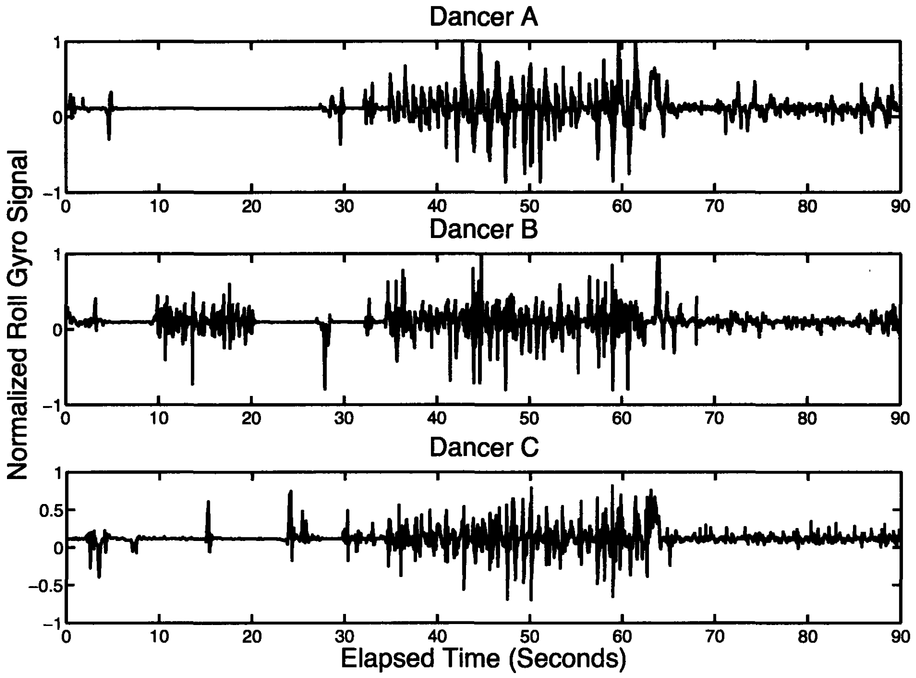
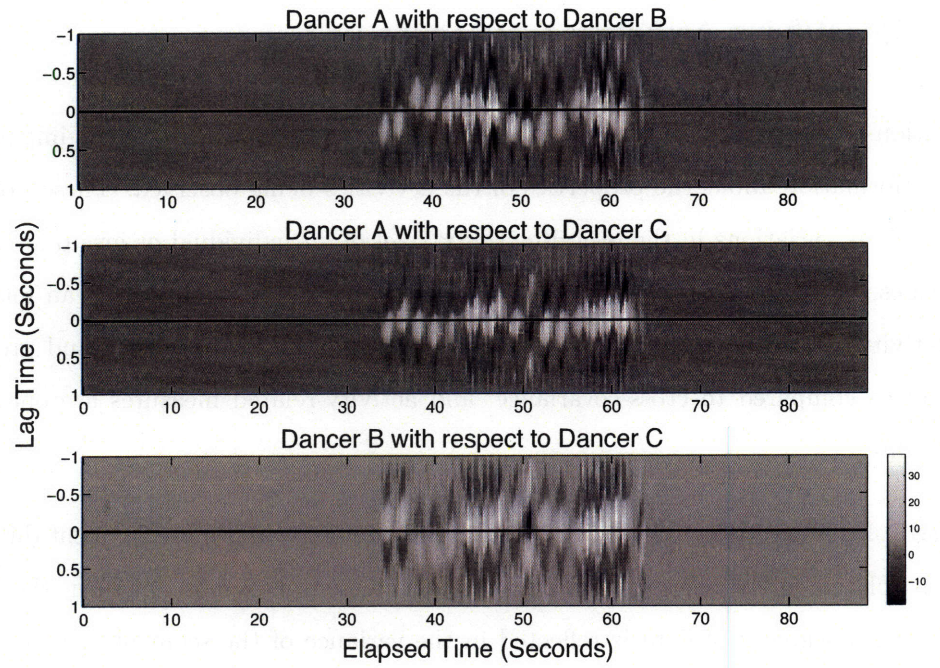


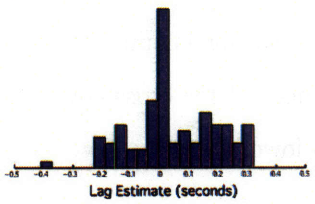
Figure 5-3: Selected raw data from the ankles of three ballet students performing a sequence of leg swings in unison.

tends to waver around the baseline as time progresses. This is consistent with the dancers slowly leading and lagging with respect to one another by small amounts. The histograms in Figures 5-4(b), 5-4(c), and 5-4(d) roughly show the extent of the peak drift for each of the three plots in Figure 5-4(a). It is clear from the relative stability of middle plot that Dancer A and Dancer C were closely synchronized for the duration of the exercise. However, the other two pairings were not as stable. For instance, in the top plot and corresponding histogram, Dancer A fluctuates from about 0.2 seconds ahead of Dancer B to about 0.3 seconds behind Dancer B. There is even more disparity in the relationship between dancers B and C. These fluctuations reflect accurately what is visible in the video. Interestingly, it turns out that Dancers A and C were facing each other during the exercise, while Dancer B had her back turned to the others.

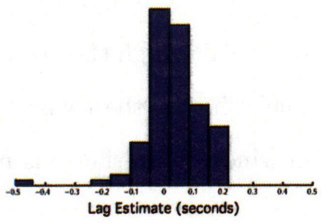
Another observation regarding Figure 5-4(a) is the fact that in some areas the covariance peaks are not as well defined as others. This is especially true in the bottom plot, relating



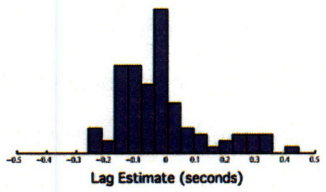
(a) Windowed cross-covariance (averaged across sensor values) between pairs of dancers, for the data segment presented in Figure 5-3.



(b) Xcov peak drift histogram for dancer A versus B.



(c) Xcov peak drift histogram for dancer A versus C.



(d) Xcov peak drift histogram for dancer B versus C.

Figure 5-4: Time and spatial correlation with three dancers performing leg swings in unison.

dancers B and C. Smaller peak values indicate less similarity in the motions being performed, and this data seems to agree with certain gestural differences visible in the video footage. Thus, although cross-covariance is not exactly a real-time feature, it is still clearly valuable for describing group relationships, both temporally and gesturally.

5.2.2 Quantifying Activity

In addition to extracting correlations between the activities of a group, it is important to obtain information about the properties of the activities being observed. These properties might include variations in the overall activity level of an individual or group at different time scales, principal axes of movement, or other features extracted during an interval of high activity. It will also be necessary to find features with lower latency and processing overhead as compared to cross-covariance, and activity-related measures are one way to achieve this.

Increased physical activity, as qualified by faster movements and more frequent directional shifts, is related to the energy present in the inertial sensor signals. In turn, the average energy over a segment of data is reflected in the variance of the segment. Therefore, one approach to activity measurement on a data stream involves computing windowed variance. Variance computed on a window of length N is shown in Equation 5.2. The computations require very little processing power, and although they rely on capturing a complete window of data, these segments can be much shorter than what was needed for cross-covariance. Hence, the latency associated with windowed variance is much lower.

$$\sigma^2 = \frac{1}{N}\sum_{n=0}^{N-1}(x[n]-\bar{x})^2 \tag{5.2}$$

Variance can be used here with various combinations of sensors, and can be processed in different ways, depending on the desired result. For instance, if the separation between gestures is long enough, the variance spikes created at the beginning and end of a movement can be used to delineate them. In other cases, it might be useful to use a median filter to obtain a slowly varying envelope on the running variance for certain sensors, in order to determine broader trends in activity level.

As an example of the latter, data was collected from the right wrist and ankle of a ballet student performing a sequence of motions in which slow kicks with the right foot transitioned into fast, tense kicks[2]. The full sequence was framed with a stylistic raising and lowering of the right arm at the beginning and end, respectively. Figure 5-5 shows a portion of the raw data from this segment along with four different activity envelopes obtained by filtering the windowed variance of both upper and lower body movement. Accelerometer activity here denotes the average variance envelope across the accelerometer axes, while rotational activity denotes the average across the gyro axes. One can clearly see a marked increase in activity as leg motion transitions to faster kicking. The role of the arm movement is apparent in the activity envelope as well.

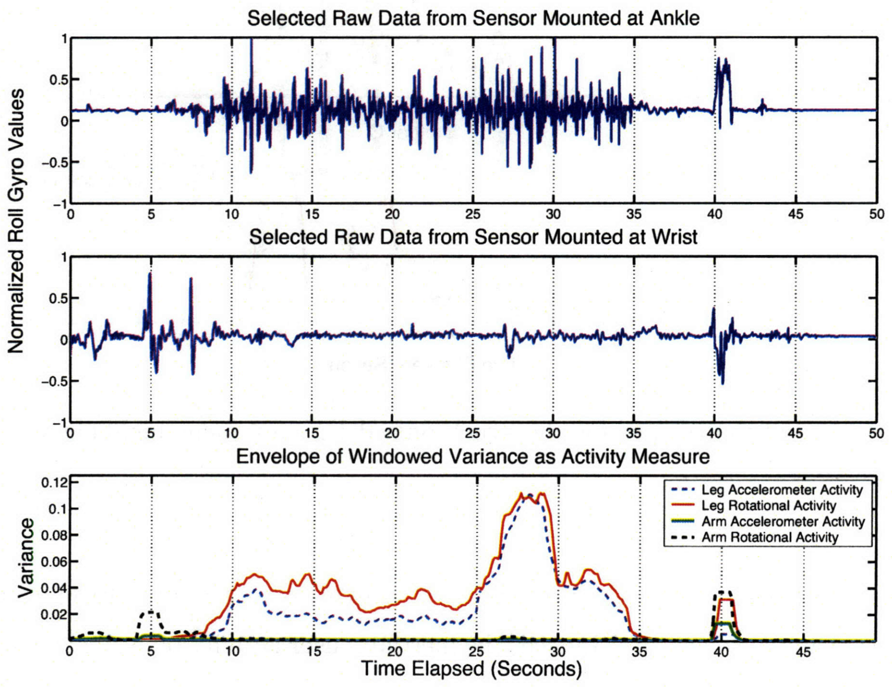


Figure 5-5: Selected data and resulting activity envelopes as dancer transitions from slow kicks to rapid tense kicks. Sequence of leg motions is framed by stylistic arm motion.

[2]In ballet terminology, the tense kicks are known as *petit battement*. Unfortunately, the terminology for the other movements was unspecified, and has not been sought out.

Similar conclusions can be drawn from Figure 5-6, which illustrates the activity envelopes of leg motion for each dancer during the period of correlated activity highlighted earlier in Figures 5-3 and 5-4(a). Two areas of peak activity across the ensemble appear around 50 and 60 seconds into the sample, corresponding to repeated leg swings over the full range of motion from front to back and back to front. The general trend of activity is increasing over the segment from 30 seconds to 60 seconds, as the instructor urges the dancers to make each leg swing "successively higher".

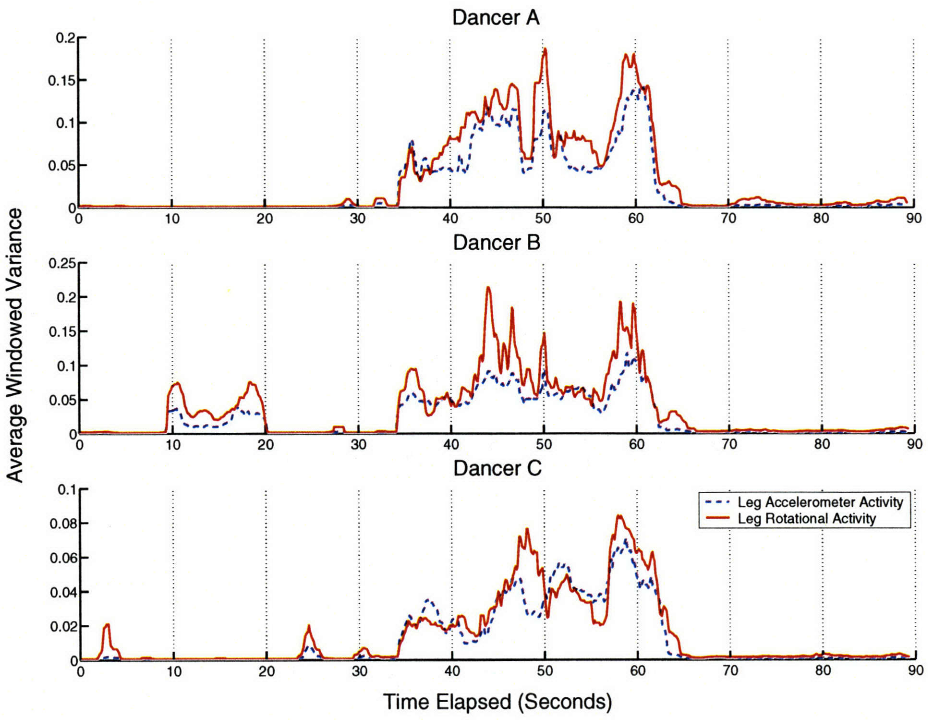


Figure 5-6: Activity envelopes for the synchronized leg movement highlighted in Figures 5-3 and 5-4(a).

Looking at Figures 5-5 and 5-6, it would seem as if there is no reason to distinguish between accelerometer and gyro activity. Indeed, the inertial sensor activity on a single node is often highly correlated, because human motion is unlikely to occur along only one axis. The accelerometers are also subject to gravity and centripetal acceleration, so rotations will be picked up strongly in some cases. It should be possible to use the gyro signals to help isolate translational acceleration from other types of movement picked up by the accelerometers.

However, if one wishes to identify a particular type of activity, it may be more important to compare motion along each sensor axis than comparing rotational versus translational motion, since a specific movement may be characterized by high variance in some directions, but not in others.

For example, Figure 5-7 demonstrates the results of a group of three people raising and lowering their right hands in unison, with sensor nodes worn on their right wrists. The bottommost plot indicates the variance on each sensor axis, averaged across all three subjects. Note that the average variance of the pitch gyro dominates, which supports one's intuition that the act of raising and lowering the hand involves mostly a rotation in pitch, and as such primarily generates activity on one axis.

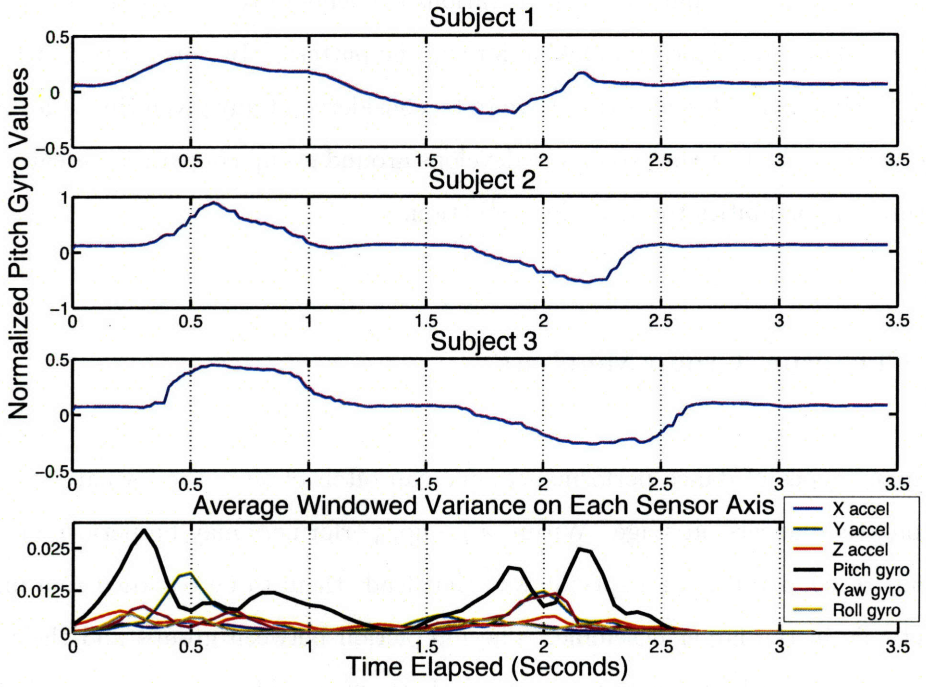


Figure 5-7: Right arm pitch gyro signals and windowed variance averaged across subjects for each sensor axis, as hands are raised and lowered in unison.

Based on this result, the variance envelopes from multiple sensor axes on one individual could be combined to form an 'activity profile' for distinguishing between certain types of movement. This profile could potentially remain effective even if it was highly simplified.

For instance, it could include only markers indicating which sensor signal has the highest variance at a given point in time and a confidence level indicating how much higher this variance is compared to the other sensors. This type of analysis can be extended to an entire ensemble, simply by averaging the appropriate activity envelopes across the group to create a global activity feature. As in Figure 5-7, global activity can be useful for determining the predominate axes of collective motion.

5.3 Group Structure as a Means for Feature Reduction

Eventually, the features extracted from sensor data will have to be processed in some way to produce a meaningful output, mapped to various elements of responsive feedback, or used as input to a detection algorithm looking for specific patterns. Because of this, it is helpful to reduce the number of features that have to be considered at any given time. In the dance ensemble setting, the fact that structure develops around group cooperation creates several very natural opportunities to make this reduction.

5.3.1 Clustering Group Members

During many types of dance performance, one can often observe the formation of groupings among the dancers on stage. Within a group, performers may be performing similar movements or cooperating in a specific way that leads them to be regarded as a unit with a distinct role in the larger ensemble. The interaction between groups and their coming and going can be used an expressive tool in a dance piece. The existence of an individual who does not fit into a group may also be significant. However, the importance of trying to detect this clustering within an ensemble using sensor data is not limited to quantifying choreographic content. If simple features can be used to form meaningful clusters, each grouping can be analyzed as one unit. This means that heavier analytical techniques can be performed on groups of dancers, rather than on every dancer individually.

Activity levels on various sensors could potentially be used as a simple way to define these clusters. For example, in Figure 5-3, activity is observed for Dancer B in the interval from 10 to 20 seconds that is not reflected in the movements of the other dancers. Referring to the video, this was confirmed to correspond to a few warm-up leg swings by Dancer B. By acknowledging that the activities of Dancer B are slightly different from those of the other dancers, a clustering algorithm might add evidence to the claim that Dancer B should be grouped differently from Dancers A and C. This grouping might be meaningful in light of the discussion above, in which A and C were found to be tightly synchronized in time, while B was slightly more of an outlier.

At the same time, the activity levels for all three dancers during the period of synchronization between A and C are presumably very similar, and it remains to be seen how a clustering algorithm based solely on activity envelopes would handle time sensitive groupings, such as leader versus followers. It is also worthwhile to mention that although activity envelopes cannot measure small time deviations, the cross-covariance analysis shown in Figure 5-4(a) is unable to compare the warm-up leg swings with the similar motion occurring later in time, because the time separation is larger than the one second cross-covariance window. Therefore, while clustering is very useful, it is not always clear what features would be best for defining the clusters.

5.3.2 Selectivity Based On Group Statistics

Depending on the circumstances, there is another option for feature reduction, which also makes use of comparisons between activity levels within the group. In this case, rather than going as far as clustering members of the group, the structure of the group is used as a basis for considering only what data appears to be statistically interesting. While sophisticated algorithms for this statistical analysis have not been developed with the system presented here, the concept can be put to the test in very simple terms.

For instance, the definition of 'statistically interesting' might include anything that appears very different from the predominant activities of the ensemble. In this case, one can compute a distance measure between each individual's activity profile and the mean activity profile of the entire group. If the activity profiles are just average variance envelopes, than the distance measure can be as simple as a series of squared differences. When this distance is large, an individual's movements lie outside the norm, and at this point a more detailed analysis can be performed.

Another possibility is that unique events are not considered important, and the analysis is meant to focus instead on the net characteristics of the entire ensemble. In this case, all of the features generated by individual performers can be merged appropriately to form a smaller set of average ensemble features.

One can also imagine a situation in which the covariance measurements discussed above are desired, but it is unclear who should be interpreted reasonably as a 'reference' for the rest of the group. By selecting the individual who's activity profile lies closest to the mean activity profile, there is a good chance that this individual will be in step with most of the others, and will result in cross-covariance measurements that reliably capture the state of the group. If all cross-covariance measurements can be made with respect to one individual, this cuts down on the number of computations that have to be made.

Chapter 6

Assessing Capacitive Sensor Performance

Feature extraction strategies for group movement have focused solely on the interpretation of inertial sensor data. However, the capacitive sensor system described in sections 2.4 and 3.6 was fully implemented in this design, and the data it provides may also be relevant to motion analysis. This chapter briefly details performance results obtained with the capacitive sensors during preliminary testing, and outlines the range of possibilities for future applications.

In making test measurements there were two major considerations to address, namely electrode design and grounding through the body. As discussed previously in section 2.4, larger electrodes were expected to result in a significant performance improvement, but they can quickly become cumbersome in the context of a wearable device. In this case, the assumption was that the area taken up by the strap affixing sensors to the ankles or wrists can easily be turned into a capacitive electrode, while anything larger is impractical. The test procedure described here compares this best-case electrode arrangement to a more compact arrangement. Additionally, allowing sensor nodes to share a ground reference through the body was expected to improve performance, and consequently tests were made both with and without the body grounded.

For testing purposes, the bracelet electrode design was simulated with a loop of copper foil and insulating tape, as shown in Figure 6-1. The smaller electrode used here was a patch of copper foil spanning the faceplate of the sensor node package, pictured in Figure 6-2. A ground path to the body was established by wiring one of the grounded screw mounts to a metal grommet in the strap. Measurements were then recorded over a range of electrode spacings for a series of electrode and grounding arrangements. In each case, both the transmit and receive node used the same arrangement.

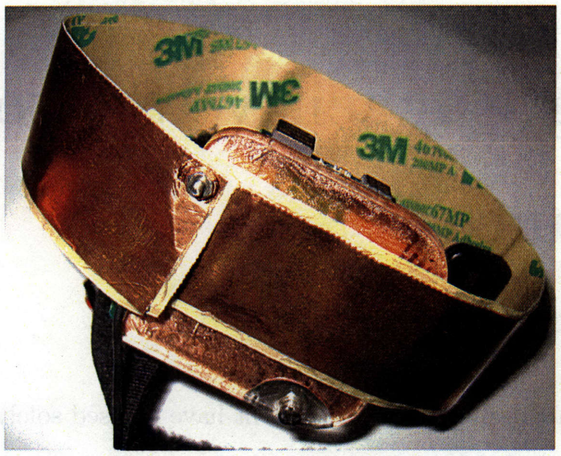


Figure 6-1: Bracelet electrode for capacitive sensor.

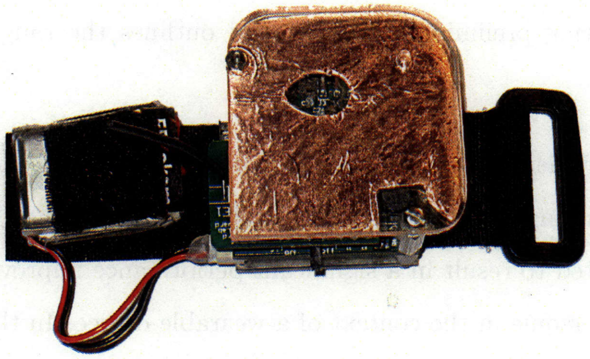


Figure 6-2: Faceplate electrode for capacitive sensor.

Figure 6-3 displays the typical response obtained, after lowpass filtering the data to about 15Hz to eliminate noise. It is clear from the results that grounding the body is critical for good performance. Even the grounded sensors with small electrodes perform better than the ungrounded sensors with large electrodes. Further, adding a shared ground to the system with large electrodes nearly doubles the useable range. On top of this, the large bracelet electrode has a clear advantage in that it is omnidirectional — the faceplate electrodes had to be pointed directly at one another to obtain the ranges shown here. Still, capacitive proximity sensing is a short range system, only working for distances within about 50cm. This is adequate for sensing over about half of the range of separation of the arms, and measurement becomes quite sensitive for movements within a 20cm range. Other range-finding solutions, such as infrared (IR) intensity measurement, are not much of an improvement in terms of range, and are more directional. However, within a range of 50cm the possibilities for engaging multiple performers on a stage are limited to close interpersonal interactions.

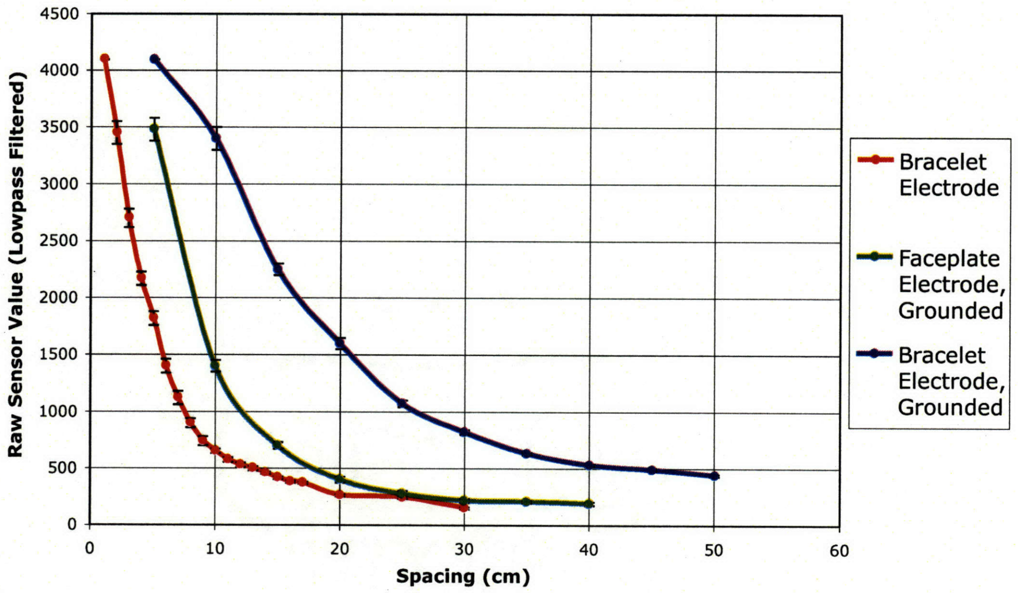


Figure 6-3: Typical capacitive sensor response with different electrode configurations.

One intriguing use of the capacitive sensors in a group context might be to detect physical contact between wearers. This has not been tested, but with the limited range, it seems unlikely that high values would be measured between sensors worn by two different individuals unless they were in physical contact, and therefore sharing a common ground. If the ground coupling between two individuals through the floor is not as strong as the coupling through skin, as is generally the case, one would also expect to see a sudden increase in gain when physical contact is made. This modality was explored in early work at the MIT Media Lab, which demonstrated using the body as a conductor through which data could be transported between people via a simple handshake [63]. The foreseeable difficulty in attempting these kinds of detections with the capacitive sensors is ambiguity, since the output can fluctuate with the presence of electrical conductors or noise in the surroundings, and indeed contact with other people or objects, just as easily as it fluctuates with changes in electrode spacing. In addition to this, there appears to significant variability in bias offset from sensor node to sensor node. This property tends to make accurate detection based on thresholds very difficult. At any rate, the capacitive system provides an interesting applicable auxiliary sensing modality, and more development is warranted to find how it can best be used to compliment the capabilities of the IMU sensors.

Chapter 7

High Performance Adaptation for Athletics

7.1 Motivation

Although interactive dance provided the impetus for designing the Sensemble system, its possible uses extend to any area in which high-quality human motion analysis may be applied, especially those requiring low latency feedback, high sample rates, or many points of measurement. Athletic sports comes to mind as one of the most promising areas for further development, in part because of demand from both high-performance professional markets and growing consumer markets. Wearable sensors for health monitoring or activity classification are becoming more and more prevalent, but few of these systems are intended to handle, much less focus on, the large accelerations and high speeds of athletic movement. Thus, a sensor platform designed for dancers, athletes themselves, is an attractive starting point.

Given the multi-user capabilities that have been stressed in this work, it would be natural to test the system in a team sport setting. Progress in this direction could generate some very interesting research in modeling strategy, communication, and competition. However, the benefit of sensor technology applied to training individual athletes is far more immediate, and within the scope of this project it was more practical to implement. Also, using one individual is not necessarily a limitation, as higher sampling rates can be achieved with the alloted bandwidth. Thus, single-subject training applications were pursued for evaluating the sensor system in the context of athletic sports.

The best chance to test the performance of the system in a demanding athletic training situation came when colleagues at the Massachusetts General Hospital Sports Medicine Department expressed interest in using the sensors to study professional baseball pitchers, in collaboration with the Boston Red Sox. Their interest in focusing on pitchers stems from a recent increase in injuries associated with the shoulder and elbow of the pitching arm, an increase which has not been limited to professional athletes [64, 65]. These joint injuries are typically the result of wear building up over time [66, 67, 68], but the risk of injury clearly escalates if a pitch is made with poor body mechanics or after passing the threshold of muscle fatigue [69, 70]. A more definitive risk assessment is difficult to make, in part because the mechanics of the arm during a baseball pitch are not fully understood. Especially for professional athletes, who routinely throw fastballs with release speeds approaching 100mph, the critical portion of the arm motion is simply too fast to be reliably measured with most techniques. The speeds generated by these players are at the limits of human ability, and being able to quantify these limits is crucial to understanding what makes the arm prone to injury. In particular, health practitioners are looking for the peak value of angular velocity at the shoulder (internal rotation), peak angular velocity of elbow extension, and peak acceleration at the wrist. These are the processes by which most of the force is directed to the ball during a pitch, and thereby are also associated with the greatest wear on the arm.

The biomechanics literature specifies vague ranges for these parameters. For instance, internal rotation of the shoulder peaks somewhere around 10,000 deg/sec [71, 72], elbow extension peaks between 2,500 deg/sec and 4,500 deg/sec [71, 73, 74, 75], and peak angular accelerations of the arm are quoted at anywhere between 300,000 deg/sec^2 and 500,00 deg/sec^2 [75]. The acceleration phase of the pitch, during which the peaks occur, is only

20–40 ms long. Typically, video motion capture systems have been used to take these measurements. However, even the state of the art system at the American Sports Medicine Institute (ASMI) is limited to 240Hz sampling. At this sampling rate, fewer than 10 frames of the critical phase of the pitch can be recorded. In addition, to obtain accelerations and velocities from the positional video tracking data, calculations have to be made which further reduce the accuracy of these 10 samples. Video systems also suffer from the fact they can only be used in constrained spaces and require significant infrastructure. Thus, video motion capture cannot move out into the field to measure players in a realistic situation. Wireless IMUs, on the other hand, can be sampled at much higher rates, provide acceleration and angular velocity measurements directly, and can theoretically be used anywhere within range of a basestation. Only one previous pitching study using an inertial sensor is widely known, and at the time, over 20 years ago, technology was too limited to achieve definitive results [76]. The availability of the high-speed, high-resolution, compact wireless IMUs developed here provides a great opportunity to explore new territory and to begin to address the limitations of video motion capture for high-performance biomechanics research.

7.2 Goals

Currently, one pilot study has been accomplished using the sensor system with professional pitchers, with procedures and results initially summarized in [77]. This was carried out at Red Sox spring training camp in Ft. Meyers, Florida, where a camera-based motion capture system (XOS Technologies) was also being tested. The goal of the study was to capture several pitches with the inertial sensor system, without any rigorous calibration, simply to see how the results compared to predictions from the literature and the performance of a state-of-the-art motion capture system. It was speculated that the results would not only agree with the video tracking system and previous predictions, but might exhibit the potential for more detailed measurement than what is possible at current video rates.

To make these measurements, sensors had to be securely mounted at several locations on the pitcher's body, for instance, the torso, the upper arm, the wrist, and the hand. At some of these points, the peak accelerations and rotational velocities were expected to be extremely high — over 80g ($784 m/s^2$) of acceleration at the hand and 10,000 deg/sec rotation at the shoulder. Thus, the sensors had to be modified to measure ranges as high as these. In order to capture the structure of peaks lasting only 20–40ms, the target sampling rate was also increased to 1kHz. The camera tracking system employed during this study captured only 180 frames per second, meaning that at 1kHz the inertial sensor system could capture between five and six times as many data points. In this case, however, it was not necessary to transmit the data at low latencies. Instead, the strategy was to log data to on-chip flash memory and allow it to trickle back at lower rates between pitches. Thus, it was necessary to make sure enough flash memory was available to record a full pitch. Finally, it was necessary to develop a way to synchronize the inertial data stream with the video data stream.

Figure 7-1: Baseball pitching subjects the arm to extreme accelerations.

7.3 Adapting the Hardware

The most important adaptation to make to the system was scaling from modest 10g, 300 deg/sec sensors up to devices capable of measuring over 80g and 10,000 deg/sec. Furthermore, these peak requirements are lower at some measurement points, the torso for instance, so it was considered appropriate to develop both mid-range and high-range sensor designs.

Several high-range accelerometers are available off the shelf. In this design, the ±70g ADXL78 and the ±120g ADXL193, both from Analog devices, were used. Although these sensors are essentially pin compatible, they only measure one axis. Because of this, a modified sensor daughtercard was designed for high-range applications (see Figures A-5, A-15), which allows one of the cards to house two orthogonal 1-axis devices. The situation regarding gyroscopes is somewhat more difficult, as the largest nominal range available is 300 *deg/sec*. As mentioned in Chapter 2, there is a simple way to increase the range of the gyro by about a factor of four, by placing an external resistor (R37 in Figure A-1). This alteration was desirable for dance, and had already been provided for in the basic design. However, for the pitching trials, the goal was to increase the range of the gyro by a factor of more than 33.

Fortunately, there is a modification capable of achieving this drastically increased range, of course at the expense of noise levels. The sensing element of the gyroscope is a polysilicon resonator which is driven into oscillation. Its deflections can then be related to the angular velocity of the device. The element is driven at a high voltage, in this case 12.5V, to achieve its sensitivity, and to accomplish this the ADXRS300 actually uses an internal voltage regulator. However, the output of this regulator is connected to one of the device I/O pins, providing an opportunity to override the driving voltage level. There is no formal analysis available from Analog Devices regarding the detailed behavior of the gyroscope when the driving voltage is altered, but the company was able to provide the theoretical relationship between voltage and sensitivity, as shown in Figure 7-2. As the driving voltage decreases, so does sensitivity, resulting in a proportional increase in range. Based on this curve, the device is capable of operating in some capacity with range increases as large as 50 times the nominal range, if the voltage supplied to the drive circuit is brought all the way down to 5V.

For the application described here, a range of about 1200 *deg/sec* can be obtained by setting the external resistor, after which an additional reduction in sensitivity by roughly a factor of ten seemed appropriate to ensure measurement out to 10,000 *deg/sec*. This corresponds

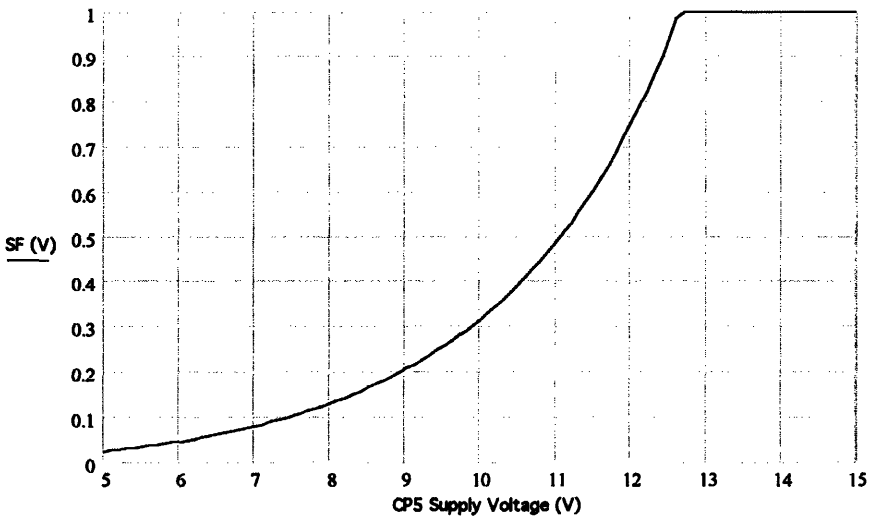


Figure 7-2: Relative scale factor of sensitivity versus driving voltage for the ADXRS300 gyroscope.

to reducing the voltage drive level to 7.5V. Luckily, an additional voltage regulator was not necessary — the current supplied by the internal converter was sufficient to drive a 7.5V zener diode placed from the regulator I/O pin to ground, as shown in Figure A-5. On the specially designed high-range sensor cards, a footprint was provided for the zener (see Figure A-15). The gyro already mounted on the main board was just as easily modified by placing a surface mount zener on top of existing capacitor C18 (see Figure A-1).

For the pilot study, the kind of calibration necessary for true absolute motion tracking was not attempted. For one thing, the datasheet indicates that modifications to the gyro output range should be accompanied by checking and resetting the output bias offset with another external resistor. Although this resistor was provided for in the high-range sensor daughtercard design (R2 or R3 in Figure A-5), the observed bias offset was not severe enough to impede the measurement range and was not adjusted. Additionally, the bias reading of the gyroscope is clearly visible in recorded data and can be removed after the fact. Nonlinearity in the modified gyros is possible, but this was also assumed to be negligible. However, to make any sort of useful measurement with the modified gyros, the actual range had to be verified, at least within a few hundred *deg/sec.* This was not a trivial task with minimal tools, as the potential peak range of 12,000 *deg/sec* after modifications corresponds to 2000 RPM.

The most readily available motor capable of reaching 2000 RPM in our laboratory was inside a Dremel handheld rotary tool. Conveniently, the Dremel bit could be replaced by a small aluminum plate which houses a sensor node securely by its three screw mount points, with liberal amounts of electrical tape to secure the battery. A rig was built to secure the tool horizontally, and a variable AC transformer was used to control the speed of the motor. In order to determine the speed of rotation, a marker was attached to the spinning platform so that it passed through a break-beam optical sensor taken from an old mouse with every rotation. The signal from the phototransistor was measured on an oscilloscope to determine the frequency of the interrupted beam. With this setup, using a 7.5V zener and setting the range adjustment resistor (R4 in Figure A-5) to 120 kΩ, the peak range was estimated to be roughly 1950 RPM, or 11,700 *deg/sec.*

The final set of hardware modifications focused on synchronization and memory. In order to synchronize with a video motion capture system, the RGB LED on one of the adapted nodes was exchanged for an infrared LED which was expected to be visible to the infrared cameras. Since data was to be logged to memory for this application instead of being immediately transmitted, the MCU was upgraded from the MSP430F148 to the MSP430F149, moving from 48kB to 60kB of internal flash memory. Given that the typical code running on the sensor nodes takes up about 4kB, this leaves 56kB of storage space. The six inertial sensor values, at 12 bits apiece, take up 9 bytes per sample. With an additional 1 byte timestamp and a sampling rate of 1kHz, this means that the node can store up to 5.6 seconds of data. This is not a luxurious amount of time, but was considered adequate for storing one pitch from just before the initial stance to the completion of the follow through.

7.4 Extending the Firmware

7.4.1 Communications Structure

The firmware running on each sensor node was mainly built around the communications protocol, specifically designed for a model in which a central basestation clocks the network at the sample rate, and receives data from the nodes every sample period. With a sample rate of 100Hz, there are 10ms to complete the negotiations required to collect each sample. In particular, partly because of the low SPI baud rate, it takes more than 1ms for each node to transfer a sample's worth of data to its radio module and initiate transmission, as discussed in Chapter 3. Using the same model at 1kHz, or ten times as fast, is not possible to perform the tasks required to stream data at the sample rate from even one node. The solution, as mentioned previously, was to store data temporarily in flash memory and retrieve it later.

Given this behavior, some of the low latency properties of the original communication structure are no longer relevant. Still, it was not necessary to design a completely new protocol for this preliminary study. During the sampling phase, the standard communications cycle can be adapted, by allowing each node to run on its own timer at 1kHz, while using the 100Hz broadcast pulse from the basestation to resynchronize the network after every ten samples. The clock skew on the nodes was found to be small enough at 1kHz that resynchronization could be performed in this way without noticeable jitter. When it is time to collect the stored data, nodes revert to the standard system, transmitting data packets in assigned TDMA slots at 100Hz.

The change of behavior between logging data and retrieving data suggests that states will have to be added to the system. In the previous structure, nodes could be in sample or idle modes. Here, there are two new modes — the 'Write' mode for storing data, and the 'Read' mode for transmitting it back. In addition, a number of states for reading and writing flash memory are encapsulated in each of these modes. As before, the basestation requests a state change in a broadcast packet, using the additional codes shown in Table 7.1. Since the state transitions are more complicated in the adapted system, an acknowledgment code is transmitted back to the basestation to make sure that all of the nodes reach the requested state. These acknowledgments replace the node header in a node data packet (Table 7.2, which is sent in accordance with the TDMA scheme discussed previously. The system of state transitions and responses will be described in further detail in the following section.

Mode	Header	Timestamp	Target ID	Message Code	Message			Unused
Sample	0xAF		All Nodes 0xFF					
			No Message 0x00	Set LEDs 0x01	R 0x00–0x04	G 0x00–0x04	B 0x00–0x04	
Idle	0xFF	0x0000–0xFFFF	One Node Node ID 0x01–0x19					
Write	0xBE		Start Writing 0xFF	Nodes do not respond to messages.				
Read	0xCD		Any					

Table 7.1: Basestation broadcast packet structure with additional options for logging and retrieving data.

Header 1B	Payload 15B						
Node ID	Inertial Data 9B						Other Data 6B
0x01–0x19	AccX	AccY	AccZ	Gyr Pitch	Gyr Roll	Gyr Yaw	Capacitive Measurements
0xF5	Finished Reading, Node Idle, Payload Empty						
0xFF	Flash Memory Full, Payload Empty						
0xF0	Flash Memory Ready (Erased), Payload Empty						

Table 7.2: Node packet structure with additional options for logging and retrieving data.

7.4.2 Logging and Retrieving Data from Flash

In write mode, whenever samples are recorded on the sensor node, they are logged immediately to flash memory, using the `write_flash_byte()` routine (see Listing C.3). During this transaction, it is important to ensure that memory has previously been erased, that no address will be repeatedly written to, and that program code will not be overwritten. In addition, memory cannot be erased by byte, but must be erased in 512 byte segments. To ensure this, the address to be written, stored in `FlashPtr`, always starts from the top of free memory and moves downward with each byte written, until it reaches the beginning of the first full segment following code memory. In this implementation, the address at the end of the last code memory segment must be noted by the programmer as `MAX_CODE_SPACE`. Once the flash pointer reaches this lower limit, it is reset, and further writes are not valid until the entire memory is erased.

Within the 10ms broadcast cycle, the sensor nodes must collect samples and write them to flash every millisecond, with enough time to spare at the end of the cycle to resynchronize to the next broadcast pulse. The difficulty with this procedure is that byte-by-byte writes to flash are fairly slow. Writing 10 bytes per sample at 1kHz, or 80kbps, approaches the limits of the current MCU. Flash can be written in blocks at higher rates, but this instruction must be executed from RAM, which is inconvenient to implement. Because of the timing limitations, once the node begins writing to flash, it cannot stop to read the messages it receives from the basestation, or to transmit its own messages. Instead, it automatically resynchronizes when any valid RF signal is received, and writes to flash without interruption until all of the free memory has been used up. New timer routines were required to run this 1kHz sampling procedure independently from the broadcast cycle, and this was provided for by eliminating capacitive sensor functionality and using `Timer_B` to control the logging of data. Another sacrifice was made, in that the current implementation does not store the 1 byte timestamp of each sample, but only stores the 9 bytes of inertial sensor data. This means more data can be stored, but was an obvious mistake, as discussed below in Section 7.7.

Reading data out of flash is less involved, since it is equivalent to reading any register in memory. Therefore, accessing packets of recorded data from flash and transmitting the data from several nodes could be handled easily by the existing TDMA scheme. On every broadcast cycle, the `read_flash_array()` routine is used to fill `DATA_PACKET` in place of `sample_data()` (see Listing C.6). The data packet can then be handled as in standard operation. However, firmware must ensure that packets are read out sequentially in the order they were written, and that once the last packet is read, flash memory must be erased

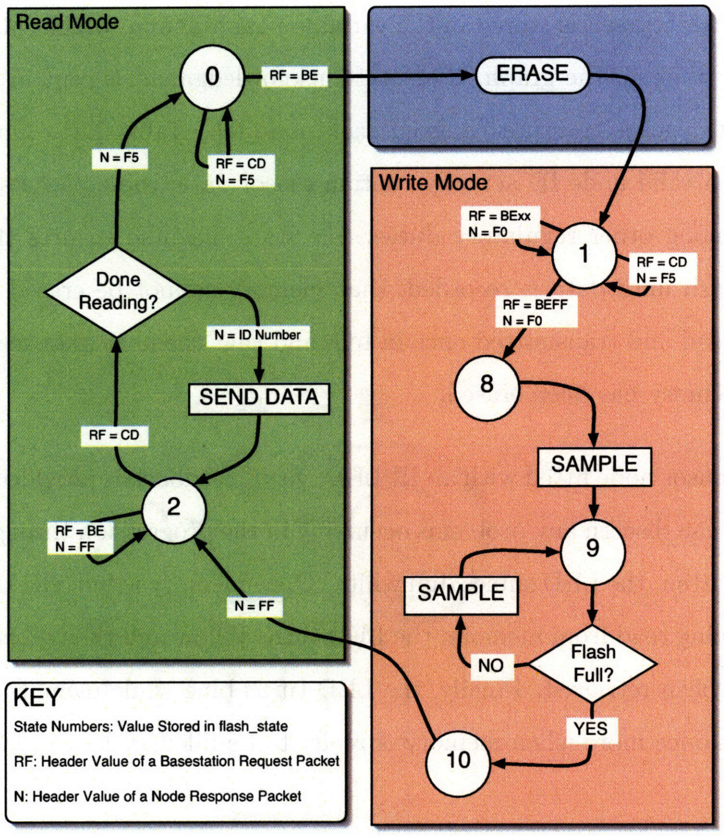


Figure 7-3: Reading and writing flash memory.

to prepare for the next write cycle. Erasing the entire data memory can take up to 1.4 seconds, during which time the node has no means of RF contact. It must therefore wait to resynchronize with the broadcast cycle until the operation is complete, and only then alerts the basestation that memory is ready.

As mentioned above, ensuring that flash is written, read, and erased with the appropriate behavior requires a series of states. Because of the time required to write and erase flash memory, nodes do not have the ability to rely on two-way RF communication from within some of these states. Thus, it is very important to send acknowledgments from the nodes with every state transition to make sure that the basestation is aware of the status of the network. The behavior of the adapted firmware running on the nodes is determined by the state machine illustrated in Figure 7-3. Typically, each transition is initiated by a state request stored in the header byte of the broadcast packet sent from the basestation (see Table 7.1). The state itself is stored in the variable `flash_state`, which takes on the values shown as state labels in the graph. The state acknowledgment is sent in the header byte of the next data packet transmitted from each node (see Table 7.2). Only data packets beginning with a valid node ID actually contain sensor data; these packets are sent during read mode. Among other regulatory duties, the state machine ensures that nothing can be read from flash until data is recorded, that memory cannot be erased until the stored data has been read and transmitted once in full, and that the next data segment cannot be written until memory has been erased.

Except on the sensor node fitted with an IR LED, visual feedback is provided from the nodes so that the user can discern any problems occurring in the progression through states. While data is being written, the LED shines dim white. Once memory is full, the LED turns green. While data is being read from memory, the LED turns yellow and then green again when all of the data has been retrieved. Finally, the LED turns blue while memory is being erased, and turns green once more when memory is ready to be filled with new data.

7.4.3 Basestation Control

Once again, the basestation acts as intermediary between the host computer and the sensor network. It must maintain its sequence of broadcasts at 100Hz to synchronize the network, and to forward state requests from the host to the nodes. In this case, it must also process a larger variety of data packets returned from the network and respond accordingly. In order to accomplish this without sacrificing speed, all of the received packets are initially assumed to contain sensor data and are immediately stored in the data buffer for USB transmission (see Listing C.13). Then, the header value is checked, and if the received packet was a state acknowledgment and not sensor data, the write index into the buffer is simply not incremented. In other words, any state acknowledgments in the data buffer will be overwritten.

One other modification was made to the basestation firmware, which provides another option

Request	Header	ID Number	Message Code	Message		
				R	G	B
Sample Mode	0xAF	All Nodes 0xFF	Set LEDs 0x01	0x00–0x04	0x00–0x04	0x00–0x04
		No Message 0x00				
Idle Mode	0xFF	One Node Node ID 0x01–0x19				
Write Flash	0xBE	Frame Number	Unused			
Read Flash	0xCD	Unused				
Local	Message for basestation not passed to nodes.					
None	No new message.					

Table 7.3: Host control packet structure.

for synchronizing the inertial system with external hardware such as a video tracking system. The idea was to generate a slowly varying signal at one of the digital outputs on the MCU, which could be initiated at the same time as sampling and recording data to flash. The signal would mark the exact temporal location of the recorded data segment, as well as encoding a number identifying the segment. It could be sent to an external system without fear of grounding issues by using an optocoupler, and finally incorporated into the external data stream using a free analog or digital input. In this design, the output pin for the synchronization signal is set by `SYNCHOUT` (see C.13). The segment ID is sent as the frame number from the host (Table 7.3), and stored in the variable `framenum`. When the system enters write mode, the output signal goes high, and over the next seven broadcast periods seven lowest bits of the segment ID are clocked out starting with the least significant bit. For up segment IDs up to 63 (six bits), this leaves the synchronization signal high for the remainder of the write cycle. When the first node replies to indicate that its flash is full, the signal drops back to zero. The digital output sequence is clocked by the 100Hz broadcast cycle, hence it can be read by any external system sampling at greater than 100Hz.

7.5 Host Application Software

The host application software for collecting data in conjunction with the baseball pitching study has already been introduced in Section 4.1.2. However, the additional features provided to handle data logged to flash were not discussed. Mainly, these consist of the 'Write' button, the 'Read' button, and the 'Erase' button. Pressing each one of these sends the associated control byte to the basestation for inclusion in the broadcast packet (see Table 7.3). If it is possible to perform the requested action, the button press will result in a corresponding change of state in the network. Provided the system is in the correct state, after pressing the read button the application will begin to receive data, which is stored in a text file using the same conventions as the real time sampling mode described in Section 4.1.2. During write mode, the frame number that will be appended to the name of the saved file is also used as the segment ID for the synchronization signal sent to external hardware. In most cases, this results in logged data segments with filenames that match up with the synchronization codes recorded on external systems. However, the user should be aware that during read mode, or any time when the nodes are waiting for a command as evident by green LEDs, the state progression shown in Figure 7-3 can be interrupted by going back to idle or sample modes. In the case of sample mode, the network begins to generate real-time data in the original configuration. This data is also logged to a file, and therefore may produce a break in the data retrieved from flash, or an offset between the filename and the true segment ID. An application to visualize logged data as it is retrieved has not yet been developed.

7.6 Experimental Procedure

The pitching study was performed in parallel with the XOS Technologies motion capture system at Red Sox spring training 2006 in Fort Myers, Florida. During the session, the inertial sensor system went through preliminary testing with a retired major league player and representative of XOS Technologies while the video system was being set up and calibrated. Then, a series of pitches, all fastballs, were recorded with an active minor league pitcher during his training regimen.

Mounting points for the inertial sensor nodes were selected to provide measurement at the most important locations for pitching motion — the torso, upper pitching arm, wrist, and hand (see Figure 7-4). Each of these locations experiences different peak accelerations, as estimated in [77], and therefore sensors were specified with different ideal ranges for each location. In the case of the upper arm and wrist, it was speculated that fine motions may be just as important to measure as high range motions. To achieve resolution across the scale, a pair of sensors with different capabilities were used at these points. The final system employed six sensor nodes, with detailed positioning information and sensor ranges outlined in Table 7.4.

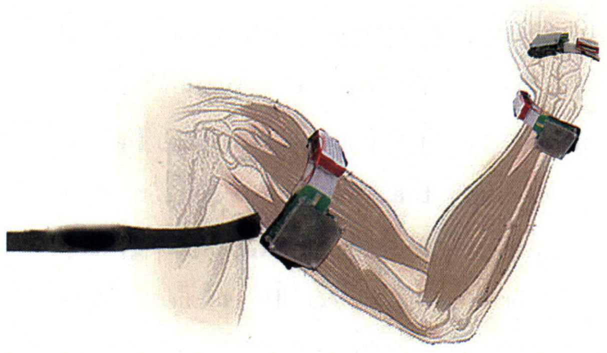
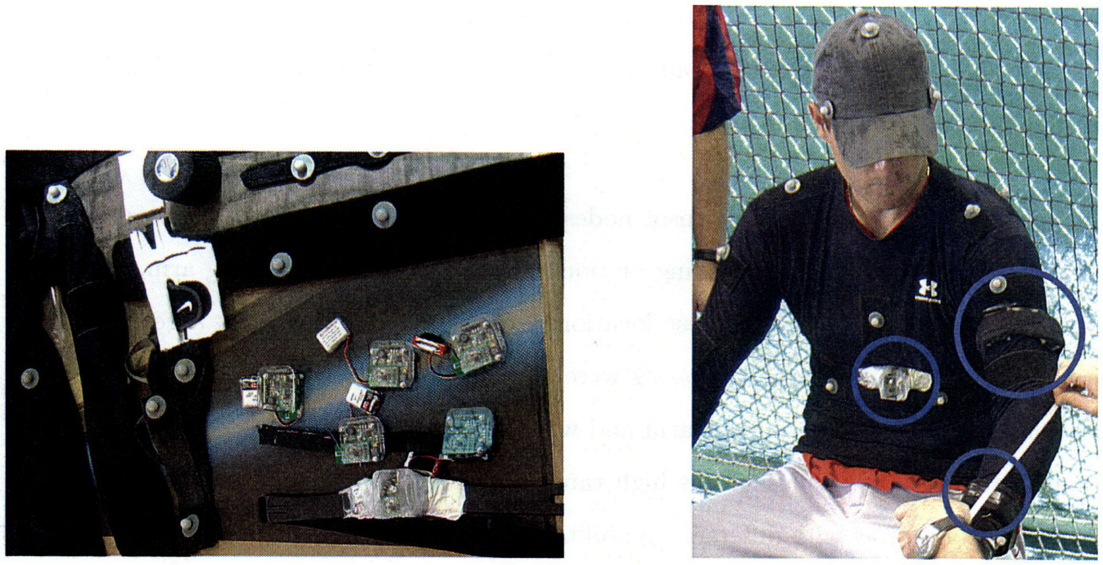


Figure 7-4: Basic placement of sensors on the pitching arm.

Sensor Position	Accelerometer Range (g)	Gyro Range (deg/sec)
Torso	±10	±1,200
Arm (Low Range)	±10	±1,200
Arm (High Range)	±70	±11,700
Wrist (Low Range)	±10	±1,200
Wrist (High Range)	±120	±11,700
Hand	±120	±11,700

Table 7.4: Sensor placements and approximate ranges for the baseball study.


(a) Six inertial sensor units alongside motion capture gear ready to go.

(b) Sensors being secured to the body of an athlete with straps and athletic tape.

Figure 7-5: Setup for baseball pitching study.

In order to mount the sensors to the pitcher's arm in a way that could withstand the forces anticipated, the standard velcro straps were supplemented by a heavy application of flexible sports tape (see Figure 7-5). Once the sensors were attached, their distance to primary joints was documented for later analysis. Although the on-body infrastructure for the inertial sensors was more bulky than the optical markers used for motion capture, it took about as much time to prepare, and the mounting strategy employed was probably the least uncomfortable method for the test subject. He claimed that although the mass of the sensors was undesirable, it did not interfere with pitching. Future tests can incorporate a number of immediate improvements to the size of the devices, for instance, a smaller battery can be used since the operating time is not as critical here as in a dance performance.

The test sequence began by a calibration of the video motion capture system, in which the subject assumed a starting position so that all of the optical markers could be found and initialized. XOS Technologies could not provide a direct signal input to their system, so the digital synchronization code from the basestation could not be used. However, the modified node with the IR LED could be seen with the cameras and did not confuse the marker system when it was turned on after calibration. The node generating the IR synchronization signal was mounted on the torso, and was programmed to flash at the beginning of the data collection cycle. Therefore, to capture the synchronization point and begin collecting data, the subject turned to face a camera and sampling on the inertial system was initiated by pressing the write button in the host application. From this point, roughly six seconds of storage space was available to capture data. If correctly timed, this was enough to capture the wind-up, pitch, and follow-through reliably. Once a pitch had been recorded at the nodes, the data was retrieved by initiating read mode. After all of the data had arrived at the host and the flash memory on the nodes had been erased successfully, then the system was ready to capture a new pitch. This process was used to capture four complete pitches from the minor league baseball player.

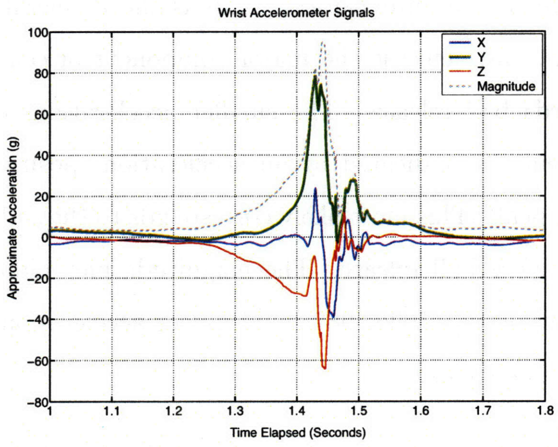
7.7 Results

The pitching study was a technical success except for a few caveats. First, in order to build the high range sensor nodes, the pitch and roll gyroscopes (x and y direction, see Figure 2-11) were transferred from old sensor daughtercards to new high range daughter-cards. Unfortunately, the gyroscopes come in ball grid grid array (BGA) packages, which are notoriously difficult to solder without proper tools. Most of the sensor boards were assembled professionally, but this modification had to be made by hand. Once a BGA package has been soldered, it is simple enough to test for short circuits, but hard to test for open circuits and poor connections. In this case, if a soldered gyro could be turned on and responded properly to movement in the lab, it was deemed to work. The difficulty turned out to be ensuring reliability. After transportation and a few hours of heavy usage on site in high humidity and high acceleration, none of the modified gyros worked properly. This was assumed to be the result of poor connections and not stress on the MEMS components, as the gyros which had not been re-soldered all worked consistently. Still, only one gyro axis was captured on the high range sensors at the upper arm, wrist, and hand, meaning peak rotational velocities could not be measured directly as intended. Luckily, acceleration was captured on all three axes at every measurement point.

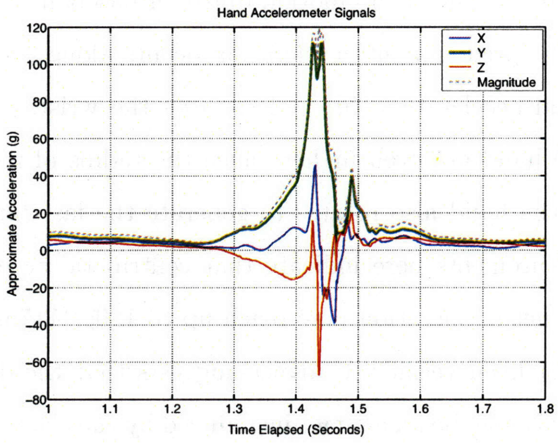
The second issue was synchronization between nodes. Because the sample timestamps were not recorded to flash memory on each node as originally intended in Section 7.4.2, all information about timing anomalies present during sampling, such as dropped RF packets, was lost. Thus, sensor signals recorded from different nodes in this trial cannot be accurately compared in time. The next iteration will, of course, eliminate the problem by recording the appropriate timestamps.

Finally, an unanticipated problem was encountered in the data retrieval scheme. Data is sampled and stored at the nodes at 1kHz, but then it is retrieved using the original TDMA scheme, with each node sending a single data packet every cycle. In other words, data is collected from the network at only 100Hz, ten times slower than it was generated. Therefore, the 6 seconds of data stored on the nodes takes an entire minute to retrieve. In the laboratory, it was considered more important to capture the pitch than to optimize response time. However, the fact that a training professional baseball pitcher throws a strict number pitches per session with no breaks and then rests his arm was not considered. During the study, there was a limited window in which to collect data, and only 4 pitches were successfully recorded because of the long data retrieval times. This will not be as much of an issue for future studies — since the TDMA cycle fits 25 nodes, retrieval time can be reduced by a factor of four by simply allowing the six nodes to transmit four times per cycle. Further optimizations can most likely be achieved by developing a more specialized RF communications structure.

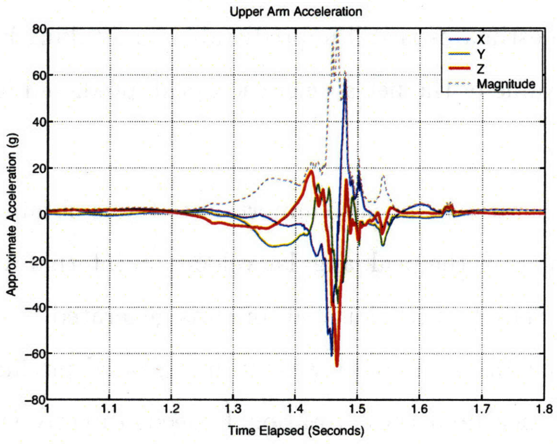
Figure 7-6 shows an example of the accelerometer signals and estimated magnitudes recorded from the wrist, hand, and upper arm during one of the pitches. The results indicate an area of significant movement over roughly 400ms, with a rapid peak acceleration phase occupying under 30ms. For reference, the IMU sampling at 1kHz captures 30 samples during this acceleration peak, while the video system is able to capture only 5 frames in the same interval. The acceleration values shown in the plots are approximate, as the sensors were never calibrated. Nevertheless, a very sharp increase in acceleration was recorded on all nodes, with approximated peak values in accordance with the video tracking results and predictions in the literature.



(a) Wrist.



(b) Hand.



(c) Upper arm.

Figure 7-6: Acceleration phase of a baseball pitch measured on various accelerometer axes.

At the wrist, the predominant axis of acceleration is the y-axis as shown in green, which peaks above 80g ($784 m/s^2$) (Figure 7-6(a)). This is the acceleration directed down the arm towards the elbow, and hence is the major component of centripetal acceleration as the arm rotates towards ball release. The smaller accelerations on the other two axes are transverse movements, as indicated by an acceleration (positive) peak followed by a deceleration (negative) peak. This most likely corresponds to the sudden burst in tangential velocity immediately before ball release. The magnitude plot indicates that at its peak, the wrist is experiencing nearly 100g ($980 m/s^2$) of net acceleration, although this is only sustained for several milliseconds.

The hand shows a similar result, as its motion during a baseball pitch closely follows that of the wrist. Again, the predominant acceleration occurs along the y-axis, and is mainly the result of centripetal acceleration directed towards the wrist. As might be expected, the peak values at the hand are even higher, since the radius of rotation from the elbow is slightly longer and the hand also rotates to generate the last bit of momentum before ball release. The measurements here indicate that centripetal acceleration peaks around 110g ($1078 m/s^2$), and net acceleration may reach up to 120g ($1176 m/s^2$) (Figure 7-6(b)). Although it has not yet been verified, evidence suggests that the distinct double peak in acceleration on the y-axis may be an artifact introduced by the elasticity of the athletic tape securing the sensor package to the pitcher's hand. However, the effect of this 'snapping' artifact on acceleration estimates is still unpredictable, and taking it into account does not necessarily reduce the value of the net acceleration peak predicted here.

In addition to peak accelerations, it is also important to try to obtain a preliminary estimate from the inertial system as to the peak angular velocity of internal rotation of the shoulder. Most of the power associated with a baseball pitch is generated by this process. Also, the shoulder is the joint that takes the most wear from repeated pitching. Unfortunately, the gyros that would have measured these rotational speeds directly were among the sensors that failed during this study. Still, it was possible to obtain rough estimates by employing centripetal acceleration. Given a radius r, for circular motion the angular velocity ω in rad/sec is related to the centripetal acceleration a_c by:

$$\omega = \sqrt{\frac{a_c}{r}} \quad rad/sec$$

At the upper arm, internal rotation of the shoulder shows up as centripetal acceleration directed inwards towards the bone. This corresponds to a negative acceleration on the z-axis of the upper arm sensor, which is clearly apparent in Figure 7-6(c). The peak magnitude of centripetal acceleration appears to reach approximately 66g ($647m/s^2$). With a biceps radius of 6cm, this results in a peak angular velocity of 104 rad/sec, or roughly 6,000 deg/sec. This is slightly lower than the expected value of 10,000 deg/sec, but it is a very vague estimate that may have been affected by the lack of calibration. It may also happen that peak shoulder rotational velocity is highly variable among individual pitching styles.

There has not yet been an attempt to formally compare the results of the inertial system with the corresponding data from the video motion capture system. Subjectively, however, the two systems produced a similar output, and the inertial data demonstrates very high peak values that seem to agree with those previously published. The advantage of the IMU is seen in the high level of detail captured during the peak acceleration phase. The nonlinearities visible within the peak as different components of acceleration interact would not be easily captured by video or other tracking methods. For instance, the rapid up-down pulses of transverse acceleration present at the peak of wrist motion last only 20ms, and could easily be misinterpreted with only 4 video frames to describe them. With calibration and additional testing, a wireless inertial sensor system could be an important supplement for video tracking in a sports biomotion context. Once the relationship between pitching parameters and inertial features is more thoroughly studied, an inertial system can stand on its own, with the potential for bringing biomechanical analysis out of the laboratory and onto to the playing field.

Chapter 8

Test Application and Results for Dance

8.1 Concept

To demonstrate the utility of the system as a multi-user interface for interactive performance, it was necessary to explore mappings for translating extracted activity features into musical sound in a satisfying way. In a traditional free gesture interface, each degree of freedom might be mapped directly to a specific continuous effect control or set of musical event triggers. In this system, however, there are at least six degrees of freedom per node provided by the inertial sensors, and typically four nodes per user, making direct mapping impractical. For this reason, the focus has been on forming descriptions of motion at the group level rather than at the individual level, thereby reducing the number of features to consider. As suggested in Chapter 5, simple group features can be used to express a whole range of useful information, such as who is leading and who is following, degree of correlation across the ensemble, changes in activity level across the ensemble, the existence of subgroups or clusters within the ensemble that could be considered separately, principal axes of activity within subgroups, the location of an event unique to one individual, or relationships between levels of upper body motion and lower body motion. In turn, the treatment of the ensemble as an organic unit offers new possibilities for musical interpretation.

In this chapter, a preliminary implementation is presented which allows sensor data to be realized as sound in a simulation of real-time operation. The goal of this test application was not to create a coherent performance piece, but to verify the capabilities of the system, and to demonstrate how the features discussed in this work could be interpreted in a real dance setting to create responsive sounds. Five dancers were monitored, each wearing sensors on both wrists and ankles, for total of 20 sensor nodes. Data was collected during a rehearsal over several repetitions of a short dance piece. In this case, the test application could not be developed while the dancers performed, so the data was recorded and analyzed offline. These recorded performances can be recreated by playing the data back into Max/MSP just as if it had arrived in real-time, with the patch described in Section 4.2.3.

As the performance is played back, the data stream is interpreted by a variety of feature extraction algorithms, covering most of the strategies for group analysis. For simplicity, only the motion data provided by the inertial sensors is considered in this case. During the mapping process, the time varying features are transformed into control parameters for sampled and synthesized sound generation. In a broad sense, the possibilities are triggering sounds, stopping sounds, modifying sound sources, and modifying effects. These can be combined and executed in any number of ways. For this design, the mapping from features to sound parameters is very simple and direct, to highlight the nature of the features being measured.

8.2 Feature Extraction Tools for Max/MSP

All interpretation of data, including feature extraction and mapping, occurs in Max/MSP. As stated above, mapping is deliberately simplified in this application, but feature extraction may still require significant computation. Unfortunately, performing heavy computation in Max/MSP using built-in objects is notoriously slow and cumbersome. Many external objects have been developed to perform advanced math operations, but these are typically

MSP externals for audio processing. The data signals here are actually control rate messages, and converting them to the audio rate to perform math would only require more operations. Usually, it is not necessary to perform a great deal of math on control signals, so very few control-rate externals have been developed for this purpose. However, since a custom Max object for data collection had already been built for this project (see Section 4.2.3), it was simple enough to develop code for custom feature extraction objects as well. The advantage of having a custom-built external object is that it does not have to be a general system building block — it can take advantage of the special structure of a specific application and abstract away many of the processing steps, resulting in a faster and cleaner patch. In this case, the externals designed for feature extraction take advantage of the fact that the data received from each node is packed with a known structure. Specifically, a list of sensor values can be treated as one input message and the parallel operations required for different sensor signals can be encapsulated within the object.

8.2.1 Cross-covariance

The cause for heavy computation during feature extraction is undoubtedly championed by cross-covariance. As evident in the definition (see Equation 5.1), cross-covariance is a type of convolution between two sequences. As such, straight computation on a window of size N requires $(2N-1)N$ multiplications and $(2N-1)(N-1)$ additions. In other words, the process is potentially an $O(N^2)$ computation. Given the similarity to convolution, however, it is not surprising that the loss can be reduced to $O(NlogN)$ through the use of fast Fourier transforms (FFTs). To illustrate this, first consider two signals $f(t)$ and $g(t)$. Their cross-correlation, $r(t)$, can be expressed as the convolution between one signal and the time-inverted compex conjugate of the other:

$$r(t) \quad = \quad f^*(-t) \otimes g(t).$$

Cross-covariance, $c(t)$, is equivalent to cross-correlation for signals that have been shifted to zero mean. Additionally, if $f(t)$ and $g(t)$ are real, as they will be in practice, $f^*(-t) = f(-t)$.

Thus the cross-covariance can be expressed as:

$$c(t) \quad = \quad (f(-t) - \bar{f}) \otimes (g(t) - \bar{g}) \quad f, g \in \mathbb{R}.$$

The same calculation can be made in the frequency domain using two forward Fourier transforms and an inverse transform:

$$C(j\omega) \quad = \quad \mathcal{F}[f(-t) - \bar{f}] \cdot \mathcal{F}[g(t) - \bar{g}] \qquad (8.1)$$

$$c(t) \quad = \quad \mathcal{F}^{-1}\left\{\mathcal{F}[f(-t) - \bar{f}] \cdot \mathcal{F}[g(t) - \bar{g}]\right\} \qquad (8.2)$$

Using an FFT algorithm to compute the transforms, the number of computations required is $O(NlogN)$, not $O(N^2)$. Hence for large values of N it becomes much faster to calculate cross-covariance in the frequency domain than in the time domain. Incidentally, this is the procedure followed by the Matlab `xcov()` function (see Appendix F).

The Matlab implementation also highlights the fact that the result contains a scaling factor dependent on the variance of the input signals. To normalize, the output can be rescaled so that the autocorrelation of the input signals at zero lag equals one. For a sequence $x[n]$, the zero-lag component of the autocorrelation, r_{xx0}, is given by:

$$r_{xx0} \quad = \quad r_{xx}[m = 0] \quad = \quad \sum_{n=0}^{N} x^2[n].$$

Consequently, the normalizing factor for $x[n]$ is $\sqrt{r_{xx0}}$:

$$\sum_{n=0}^{N} \left(\frac{x[n]}{\sqrt{r_{xx0}}} \right)^2 \quad = \quad 1.$$

Now, let $c_{ab}[m]$ be the cross-covariance between two zero-mean, real sequences $a[n]$ and $b[n]$, with associated zero-lag autocorrelation components r_{aa0} and r_{bb0}. In the frequency domain, this cross-covariance can be expressed as:

$$C_{ab}(e^{j\omega}) \quad = \quad A(e^{-j\omega})B(e^{j\omega}).$$

Then, if normalization factors are applied to $a[n]$ and $b[n]$, the normalized cross-covariance can be defined as:

$$\begin{aligned} C_{norm}(e^{j\omega}) &= \frac{A(e^{-j\omega})}{\sqrt{r_{aa0}}} \cdot \frac{B(e^{j\omega})}{\sqrt{r_{bb0}}} \\ C_{norm}(e^{j\omega}) &= \frac{C_{ab}(e^{j\omega})}{\sqrt{r_{aa0} \cdot r_{bb0}}} \\ c_{norm}[m] &= \frac{c_{ab}[m]}{\sqrt{r_{aa0} \cdot r_{bb0}}} \end{aligned}$$

Therefore, to normalize the output, the result from Equation 8.2 must be scaled down by a factor of $\sqrt{r_{aa0} \cdot r_{bb0}}$. This summarizes the procedure that will be taken to compute cross-covariance within a custom-built Max external.

Depending on the situation, different types of cross-covariance features can be called upon. For instance, it can be measured between two distinct signals, but it may also be helpful to measure average cross-covariance between two sensor nodes. In the latter case, cross-covariance is computed separately for each of the six inertial sensor signals on a node, and then the results are averaged. For example, this was the technique employed for the analysis in Section 5.2.1. To accommodate the possibilities, two Max externals were designed, `xcov` and `xcovlist` (see Listings D.6, D.7). Like `rawusb`, these were also developed in C using Xcode and templates provided in the Max/MSP SDK. The first accepts streams of integers at its inputs, and is initialized with arguments specifying the window size and step size for the computation. The second accepts lists of integers at its inputs, with an additional argument specifying the length of the lists. Computation within these objects is similar, apart from the fact that `xcov` performs one cross-covariance calculation at every time step, while `xcovlist` performs a cross-covariance calculation per list element and computes the average at every time step. Although it is possible to design a Max object that accepts general input and changes its behavior depending on the input format, it was simpler in this case to use two specialized pieces of code. Also, the behavior of `xcovlist` could be achieved with a series of `xcov` objects and a scaling factor, but as suggested above it is much more efficient to encapsulate the parallel operations in one object.

Both cross-covariance objects have two inlets, one for each of the signal sources to be correlated. Generally, because of the scheduling structure of Max/MSP, objects generate an output in response to activity on the leftmost inlet. In accordance with this principal, the left inlet here continually buffers new data points and initiates all computation. If `step_size` new data points have been received on the left inlet since the last calculation, a new calculation is performed over the last `window_size` data points. Meanwhile, data received on the right inlet is only stored to its buffer when corresponding data appears on the left inlet. This ensures that the sequences being correlated are properly synchronized, but care must be taken that when a pair of data points arrive, they appear in right to left order. This follows naturally with the Max/MSP scheduling scheme.

When it is time to make a computation, cross-covariance is computed in the frequency domain, as discussed above, and normalized with the appropriate scaling factor. The core algorithm, implemented in `compute_xcov()` (see Listing D.6), uses FFT subroutines provided by FFTW, a popular and freely distributed C library [78]. In order to achieve the full speed advantage of the FFT over time domain computation, the length of the sequences should be a power of two. Therefore, when the object is initialized, the value provided in the `window_size` field is rounded to the nearest power of two, and the actual values being used are reported to the Max output window.

Both `xcov` and `xcovlist` have 7 outlets. For a window size N, the computation at each time step results in a cross-covariance vector of length $2N - 1$. This result is sent to the leftmost outlet as a list of floating point values. However, in the course of the computation, a number of other useful features are generated, including the Fourier coefficients and the means of both input signals. These features are made available to outlets as well. Finally, the value of the cross-covariance peak is output as a float, and the location of the peak in the cross-covariance list is output as an integer index. In this way, `xcov` and `xcovlist` provide a versatile set of features beyond acting as number-crunching units. Figure E-4 illustrates the layout of `xcov` and `xcovlist` and their use in a simple patch.

8.2.2 Running Mean and Variance

Compared to cross-covariance, mean and variance are very basic computations that could be handled adequately by existing Max/MSP tools. However, in the case of variance, there is no dedicated object for performing the calculation on control rate values. Also, in the course of making the activity measurements discussed in Section 5.2.2, there is a need to compute running variance on windows of sensor data, with a variety of possible update rates and window lengths. While a patch in Max/MSP could be designed with this structure, it would not be as fast and flexible as a simple buffering scheme in C. In the case of computing mean, there are several existing objects. For instance, `mean` computes the mean of all the values it has received between resets, and `Lmean` computes the element-by-element mean of two lists. Still, a running mean feature with similar behavior to windowed variance would be useful and not easily implemented with the existing options. In order to simplify the prospect of extracting running mean and variance features, the externals `Rmean` and `Rvar` were developed (see Listings D.8, D.9).

The description of `Rmean` and `Rvar` is brief, as their function is simple and their only difference is in the central calculation. Both objects are initialized with a window length and a step size. They have one inlet, which accepts either integers or floating point numbers, and one floating point outlet for the result (see Figure E-2). Values received at the input are buffered, and any time `step_size` new samples are received, the mean or variance of the last `window_size` samples is computed and sent to the outlet.

8.3 Sound Palette and Control Space

In this design, the control medium of choice for interactive sound was MIDI, which can link Max/MSP with other music generation tools. Max/MSP is capable of generating audio output directly, of course, and can also output other standard formats such as OSC. These options would likely be more efficient than MIDI, in terms of processing power in the case of direct audio output, or in terms of throughput of control data in the case of OSC, as

MIDI has limited speed and resolution. However, for testing purposes it was considered more appropriate to use a familiar system capable of providing quick results.

Also with this in mind, Propellerhead's Reason™ was used for sound generation. Reason is a powerful modular software synthesizer and sequencer with a well-designed interface, allowing easy access to amazingly wide variety of appealing sounds. Using the `rewire~` object and running Reason as a ReWire slave gives Max/MSP full control over most of Reason's interface elements via MIDI. MIDI notes and controller values can also be sent to any Reason track from within Max/MSP. Thus, in this context, MIDI provides a very convenient way to move between data processing and sound design.

However, it is important to mention that both the MIDI protocol and the Reason interface set up certain assumptions about the control space, which can often be a limitation in terms of what types of interaction are most naturally supported by the system. MIDI was not invented as a description for music so much as a communications link for musical keyboards. Because of this, the protocol is laced with the usual keyboard-centric limitations — notes are on or off and have a strict value, buttons are pressed, knobs turn. For all its complexity, Reason was completely built around this type of interaction. Many synthesizer modules offer a battery of parameters to shape a keyboard attack, sustain, and release into something that resembles the behavior of another instrument, but control is still passed through this keyboard channel. Here, the limitations are handled as best as possible, but nevertheless strongly affect the mapping strategy. Broadening the scope of the control space and bringing in new mapping possibilities is left for future implementations where MIDI can be avoided or supplemented.

8.4 Implementation

8.4.1 The Mapping Process

Mapping can be thought of as the process of making connections between three interpretive layers, where information is passed from top to bottom. At the top layer is a set of direct features taken from the raw data; in other words, means, variances, and the like. The middle layer contains a set of descriptive elements — the observations or patterns one hopes to derive from the direct features. Contingent on the complexity of the computations linking the two layers, these descriptive elements could potentially be as specific as a set of gestures to recognize, or a set of emotional states to predict based on body movement. Since pattern recognition was not the focus of this project, the descriptive elements used here are broad reinterpretations of the direct features, such as ensemble synchronicity or average group activity. Finally, the bottom layer is made up of musical elements, typically a set of digital instruments, effects, score structures, and their associated control parameters.

Connections between the middle and bottom layers should be designed so that the descriptive elements affect the musical output in a satisfying way. Typically, the ear appreciates the complexity associated with correlation across multiple musical parameters; for instance, volume and timbre vary mutually on any physical instrument. Therefore, it is beneficial to give a single descriptive element control over several musical parameters in varying capacities, or potentially to give multiple descriptive elements control over the same musical parameter. However, for a complex mapping, the difficulty becomes balancing rich responsive sound with interpretability for an audience. At some point, it is necessary to incorporate feedback and learning to oversee such a system. This can be assisted by software, for instance supplanting the mapping process with a software agent [15], but the role of the human performer cannot be understated.

For the purposes of this work, analysis and experimentation have only begun to progress far enough to be concerned about complex interaction with feedback and learning. Rather, the mapping designed for this test application was meant to provide a simple overview of the possibilities for transforming group features into sound, and to test real-time operation for the first time. Because of this, simple sounds were preferred, with relatively direct relationships to descriptive elements, making the influence of each feature distinctly audible.

The mapping strategy is illustrated in Figure 8-1. It was designed by selecting the set of descriptive elements first. In order to address most of the group features discussed in Chapter 5, the synchronicity across the ensemble, similarity across the ensemble, global activity level, individual activity level, and several comparative activity levels were chosen. Given this set of descriptive elements, the top level features to extract were the acceleration and angular velocity magnitudes, and running variance envelopes for each sensor value. The sensor magnitudes are used to compute average cross-covariance features for each dancer, while the variance envelopes are averaged and compared in various ways to produce the activity features. Musical elements were selected to provide a simple palette of four very distinct sounds, violin, bass synth, plucked string, and flute. In the case of the bass synth, sounds were generated using a subtractive synthesis module in Reason. Other sounds were generated with a sample synthesis module, also in Reason, using built-in samples.

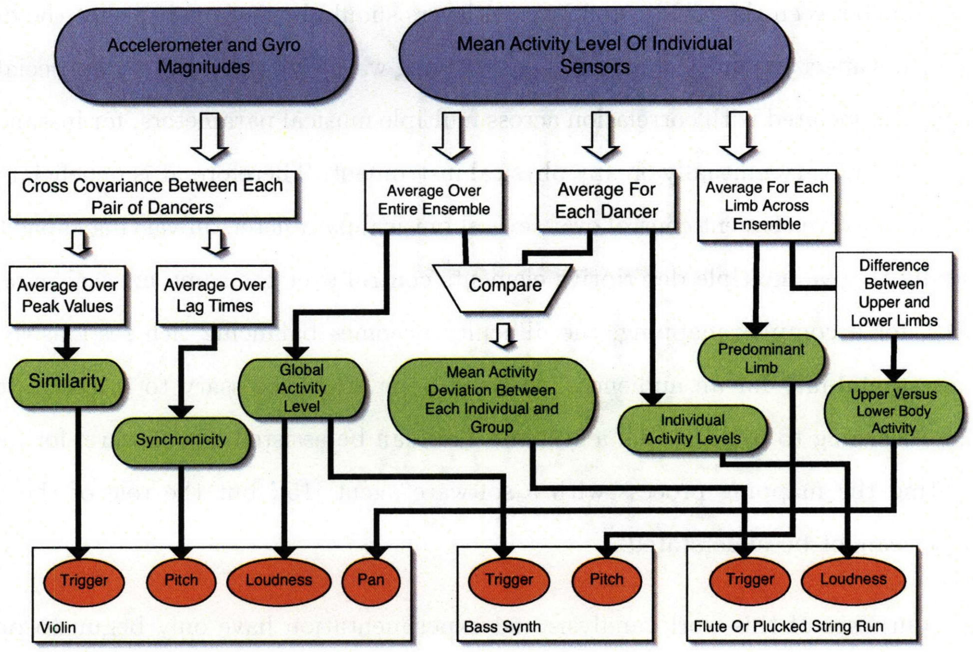

Accelerometer and Gyro Magnitudes
Mean Activity Level Of Individual Sensors
Cross Covariance Between Each Pair of Dancers
Average Over Entire Ensemble
Average For Each Dancer
Average For Each Limb Across Ensemble
Average Over Peak Values
Average Over Lag Times
Compare
Difference Between Upper and Lower Limbs
Similarity
Global Activity Level
Mean Activity Deviation Between Each Individual and Group
Predominant Limb
Individual Activity Levels
Upper Versus Lower Body Activity
Synchronicity
Trigger
Pitch
Loudness
Pan
Violin
Trigger
Pitch
Bass Synth
Trigger
Loudness
Flute Or Plucked String Run

Figure 8-1: Basic strategy for a test mapping from sensors to sound.

This structure was explored using data streamed from pre-recorded text files to simulate real-time operation, with the help of the `seq~` patch discussed in Section 4.2.3 (see Figure E-3). The full test interface is shown in Figure E-5. Playback is achieved by loading the text file with the read button (1), setting the proper Rewire channels (2), turning on the

DAC (3), and hitting start (4). The start position can be set with the upper slider (5), and the current position in the file is reported on the lower slider (6). Playback will loop between the start position and the end of the file. The generated audio can also be recorded to an output file, by selecting open (7) to create a new audio file prior to playback, and then activating record (8). The interface view gives the user some idea about how the data is being handled, but does not show details of feature extraction and mapping to sound.

8.4.2 From Data to Features

All of the feature extraction in encapsulated within a sub-patch called `giantfeaturemess` (see Figure E-6). The purpose of this object is to accept bulk streaming data from the sensor network, generate the top level features, process them, and output all of the middle layer features or descriptive elements. The actual contents of `giantfeaturemess` are true to its name, so rather than displaying the full patch, the processes involved will be broken down and illustrated piece by piece. In this case, the input data comes directly from the output stream of the `seq~` object. Therefore, the first step is to parse the bulk data stream back into individual streams for each sensor node (see Figure E-7). Then, the top level feature set, comprised of magnitude and windowed variance, can be generated for each node. The Max patch for magnitude calculation is shown in Figure E-8(a). It accepts a list with all of the sensor values for one node, ignores the capacitive sensor readings, and shifts the expected bias of the inertial sensors to zero. Then, a standard magnitude computation ($\sqrt{x^2 + y^2 + z^2}$) is performed separately on the three accelerometer axes and the three gyro axes, resulting in a two element list at the output. A similar patch for windowed variance is shown in Figure E-8(b). The preparation of the inertial data is the same, except that in addition to shifting the bias, the signals are normalized to fit within the maximum range ± 1. The `Rvar` object performs a running variance calculation on each sensor value, with a window size of 20 samples and a step size of one. The `smoother` objects apply 20-sample-long median filters, which create smoothed envelopes while preserving signal edges. The output of the patch is a six element list containing the variance envelopes for each inertial sensor value. Magnitude of acceleration, magnitude of angular rotation, and variance envelopes of each sensor signal now form the basis of the rest of the feature extraction and analysis process.

The first five outlets of `giantfeaturemess` generate descriptive features which encompass the ideas of both similarity and synchronicity, for each of the five dancers in the ensemble analyzed here. These features are loosely termed average lag times, and their intent is to quantify the average temporal separation of each dancer from the rest of the group during a period of similar activity. For instance, if one dancer is consistently lagging behind all of the other dancers, a high average lag time should be reported for this dancer. On the other hand, if one dancer is ahead of some group members but equally behind others, an average lag time of near zero should result. Average lag time is only meaningful when movements are similar, hence it is only updated when correlated activity is detected. The calculation begins by computing average running cross-covariance between each pair of dancers using sensor magnitudes (see Figure E-9(a)). Cross-covariances here are calculated with a window of 128 samples and a step size of 10 samples. When the peak value of average cross-covariance exceeds a threshold, the location of the peak in the output frame is recorded terms of an index. This index is shifted by half the size of the output frame to produce a lag estimate in samples. This procedure is performed once for every pairing of dancers, or ten times in the case of the five dancers measured for this application. At this point, each dancer is implicated in a set of four lag estimates which describe a relationship to the other dancers in the ensemble. By merging the four appropriate lag estimates, it is possible to obtain the net temporal separation of a single dancer from the group average. However, one difficulty is that lag estimates are only valid while correlated activity exceeds a threshold. Because of this, data comes in asynchronous bursts which cannot be averaged reliably simply by taking the mean of four inputs. Instead, the last step of the calculation is carried out as illustrated in Figure E-9(b). The `thresh` object collects bursts of lag estimates from all sources over 100ms, and then outputs a list which can be averaged to compute the final result. This measurement has the additional property that if a pair of dancers are correlated for a long period of time as compared to other pairs, their associated lag estimate will be more heavily weighted in the average lag. Since lag estimates are only recorded for pairings with high correlation, it is also possible that two pairs of dancers could be completely uncorrelated, but everyone in the group will have average lag times zero near because within the pairs the dancers are correlated and highly synchronized. Thus, average lag time is not strictly a measure of temporal correlation — it describes time synchronicity conditioned on spatial similarity.

The next most important descriptive elements are the global activity parameters, consisting of average upper versus lower body activity difference, ensemble activity level, predominant limb across the ensemble during an activity peak, and value associated with the predominant limb. From left to right, these are supplied by the last four outlets of **`giantfeaturemess`**. Each of these features is a result of the combined influence of windowed variance envelopes calculated on individual sensor signals. First, data from all of the sensor nodes on common limbs is combined to form average activity profiles for each limb (see Figure E-10). The four limb activity profiles can be combined in different ways to generate all of the global activity parameters, as shown in Figure E-11.

Once the ensemble activity level has been computed, it can be compared to individual activity levels to generate the final set of features, which are mean activity deviations between each dancer and the group. These values are supplied by outlets 6 through 10 of **`giantfeaturemess`**, one outlet for each dancer. An individual activity level is computed by averaging the variance envelopes of all the sensor signals common to one dancer, as in Figure E-12(a). Then, the global activity level is subtracted from an individual activity level and normalized to determine deviation as the percent difference between the two signals, as in Figure E-12(b). In this implementation, it is worth noting that the deviation between individual activity and the activity of the whole is left as a signed distance measure. In other words, a dancer standing still while the rest of the ensemble moves will receive a negative deviation, while a dancer moving alone will receive a positive deviation. This is useful for differentiating ‘solo’ movement from held-back movement.

8.4.3 From Features to Musical Sound

The descriptive elements or middle level features were mapped to sound in a very straightforward way, roughly following the plan presented in Figure 8-1. In this test application, the primary goal was to illustrate the effectiveness of the low-latency feature extraction and analysis algorithms developed in this work for generating meaningful feedback. Each instrument was designed to play a specific role in this context.

The violin was meant to highlight areas of synchronous and correlated movement. To accomplish this, a violin note is triggered whenever an average lag time value is updated, and the pitch of the note is proportional to the magnitude of the average lag estimate (see Figure E-13). The closer the lag estimates are to zero, the higher the pitches produced will be. This means that high levels of synchronicity are reflected by high pitched sounds. Also, the more correlation there is across the ensemble, the more lag updates will arrive, increasing the density of note activity. Even with the stream of average lag estimates coming from a single dancer, there is a potential for very dense bursts of notes. In the mapping designed here, the violin is driven by the cumulative influence of average lag estimates from all five dancers. In order to prevent excessive clutter in pitch space, notes are actually selected from two ranges, a narrow, low 'viola' range, and a broader high range. Three dancers are mapped to the high range and two dancers control the low range. With this arrangement, temporal and spatial alignments between dancers can excite complex clouds of violin notes. However, additional control is necessary to make the sounds more interesting. The MIDI velocity parameter sent with each note message is the standard way to convey dynamics and shifting timbral qualities to a keyboard-based synthesizer. In this design, the velocity for all violin notes is continually varied in step with the global activity envelope. As average activity builds across the ensemble, the violin notes become louder and slightly harsher when they are triggered. A more subtle embellishment was explored by allowing sensor data to affect the position of the violin sounds in the stereo field. The upper to lower body activity ratio was used to pan the violin to the right when upper body movement was prevalent, and to the left when lower body movement was prevalent. This mapping was not intended to be entirely intuitive for illustrating the classification of predominant movements, but was meant to add some motion to draw attention to the violin sounds.

Guitar and flute sounds were employed to highlight solo movement, in other words, activity by single dancers that was not reflected in the rest of the ensemble. In a similar arrangement to the high and low violin ranges, three dancers were given control over the flute and the remaining two dancers were given control of the guitar. The activity deviation feature was used to trigger both instruments (see Figure E-14). A solo event for a dancer was defined by activity deviation crossing a threshold, indicating that the activity level of the individual is higher than the activity level of the group by a certain percent. When an event is detected, rather than triggering a single note with a pitch determined by an external process, the event triggers a brief sequence of notes that are selected at random from a list. This enables the guitar and flute sounds to have a more predictable tonal quality and some degree of rhythmic structure within an event. In the case of the flute, the continuous activity deviation value after an event is triggered is also mapped to note velocity and a controller value that determines breathiness. Therefore, more dramatic differences between the dancers in control of the flute and the average ensemble behavior result in louder and breathier flute tones. In the case of the guitar, continuous expressive control is limited to note velocity, which is proportional to the mean of the activity deviations of both dancers delegated to guitar sounds.

The bass synthesizer was meant to provide a steady underpinning for the random note events occurring in the other instruments. The notes making up the bass line also had to be triggered occasionally using thresholds on features, but they could be long pedal notes able to meld together into a more continuous layer. The idea was to have sensor features slowly affect the pitch of the droning note. This was accomplished by using the predominant limb across the ensemble to select one of four pitches for the bass note (see Figure E-15). The note is activated when the global activity level crosses a threshold, but a note can only be triggered every 500ms. Global activity also continuously controls an effect parameter setting the cutoff of a lowpass filter, and the activity level of the selected predominant limb is mapped to note velocity. Also, the bass patch uses the upper versus lower body activity ratio to pan the bass track to the opposite side of the stereo field as the violin track.

8.5 Results

8.5.1 Real-Time Operation

Currently, the closest this system has come to true real-time operation with dancers has been through playback of recorded data. As mentioned previously, the testing process has involved an ensemble of five dancers wearing four sensor nodes each. This arrangement approaches the network limit of 25 nodes and produces 144kbps of intertial data alone, presenting a challenge for low-latency processing. Assuming for the moment that during live operation data can be transferred to the host computer with minimal intrinsic processing requirements, reading from a pre-recorded text file is adequate for assessing the real-time performance of the host system.

In this case, all testing was run on a 1.6GHz Power Mac G5 with 1GB of RAM. This system could handle text file access, full feature analysis and interpretation in Max/MSP, audio rendering in Reason (as a ReWire slave) including a few reverb units, and recording audio to an output file, with about 80% CPU usage. Audio was generated from several short segments of a dance rehearsal and then resynchronized to align with the sensor data segment and recorded video segment.

Qualitatively, the rendered audio and video material shows that the generated sound clearly corresponds to dance movements, possibly with small delays. For the purpose of this thesis, Figures 8-2, 8-3, 8-4, 8-5, and 8-6 attempt to recreate two pertinent video segments and their alignment with generated features and musical events. The segments are each over 20 seconds long, so the numbered video progressions displayed in Figures 8-2 and 8-3 show only major turning points. In the data plots, vertical lines indicate the location of each video frame in time. Note that the beginning and end of each data plot also correspond to the first and last of the pictured frames.

In the first dance segment, several 'solo' movements are present as dancers laying in a ring spring upwards one by one before rising together. As expected, this activity is reflected in the guitar and flute triggers set to accompany high levels of deviation between individual activity and ensemble activity. Although the latency appears relatively adequate when listening to the generated sound, most of the features extracted in the Max/MSP implementation described above are already known to incur delays which exceed the goal of 100ms maximum latency. For instance, windowed variance has been calculated with a 20-sample window followed by a 20-sample smoothing filter, which creates a 400ms delay. It is very difficult to verify this by looking at the plots in Figure 8-4, since it is unclear exactly where a gesture begins. The best example might be the area between video frames 6 and 7, in which Dancers C and D mistakenly spring upwards at nearly the same time (see video frames in Figure 8-2). By the time both dancers reach the apex of the movement in video frame 7, their variance peaks have been reported in Figure 8-4(a). Since the time from the beginning of the gesture in frame 6 to the height of the gesture in frame 7 is only about half a second, the latency appears not to exceed 400ms. However, determining the latency accurately will require more directed tests.

The second dance segment is an area of highly active and highly synchronized movement across the ensemble, including running in place and energetic arm swinging by all five dancers at once. This provided a good opportunity to look at the cross-covariance features driving the violin sounds. In terms of latency, the 128-sample window used here was expected to create a very noticeable delay of 1.28 seconds. The most significant change in behavior during this segment occurs in video frame 11, when tightly synchronized arm movement gives way to an extended period of running in place. At this point, the dancers are moving their feet as quickly as possible and consequently get out of step with each other. This event is reflected in Figure 8-6 at around 10 seconds, when the lag times tightly centered around zero suddenly dissolve into a sparse cloud. In Figure 8-5(b), the violin notes also plummet in pitch and thin out in response. The actual timestamp of video frame 11 is 8.73 seconds into the segment, meaning that the observed latency in cross-covariance measurements for this event is about 1.27 seconds, essentially equal to the expected feature latency. While there is nothing surprising in this result, it is reassuring to know that the major sources of latency are still very predictable artifacts of the feature extraction strategy. Although cross-covariance requires latency to measure large time separations, windowed variance can easily be improved by using a window and smoothing filter length as small as

5 samples to achieve 100ms latency. The tradeoff in this case is a slightly rougher envelope with less smoothing between the variance peaks at the beginning and end of gestures.

It should be mentioned again, however, that the analysis here was performed offline and is only a simulation of real-time performance. Even with improved lower-latency features, there is still the possibility of a 40ms worst case transmission latency from the basestation, as discussed in Section 3.5. Clearly, it is very difficult to meet the goals of real-time feedback with the current basestation firmware and the feature set that has been developed in this work. The best way to assess real-time operation will be to take more extensive measurements in a live setting and allow performers to respond to the output. It remains to be seen how dancers are able to handle live interaction with a system given the current latencies, and how much these requirements will have to be pushed for future progress.

Despite the latency inherent in some of the design elements, the mechanics of the real-time simulation held up well, in the sense that data arriving at realistic rates was translated from features into sound successfully on one processor. However, even this may falter in a live situation, as the assumption that the requisite USB data transfers are not processor intensive does not appear to be true. With the current basestation firmware, the host must continuously poll to receive data over the USB interface, and these USB transfers take low priority. In Windows, if the host is busy and cannot request data every 20ms, then the data is buffered and the only loss is some additional latency. However, in Mac OSX, the basestation stalls when the host cannot make requests in time, resulting in patches of missed data. Because of the low priority of USB transfers from the basestation, moving the mouse and performing GUI tasks can bring the basestation to its knees, even when data is being collected from just a handful of nodes and no intensive analysis algorithms are being run. Unfortunately, reliable timing at the basestation is crucial for the correct operation of the sensor network. If the frequent USB packets are causing such a load on the processor, it is unclear whether changing the firmware to implement interrupt transfers will actually improve the situation (see Section 3.4). In the end, short of designing a very sophisticated USB device, it may be necessary to dedicate a small computer to USB data collection and pass data via sockets to the analysis and audio rendering systems.

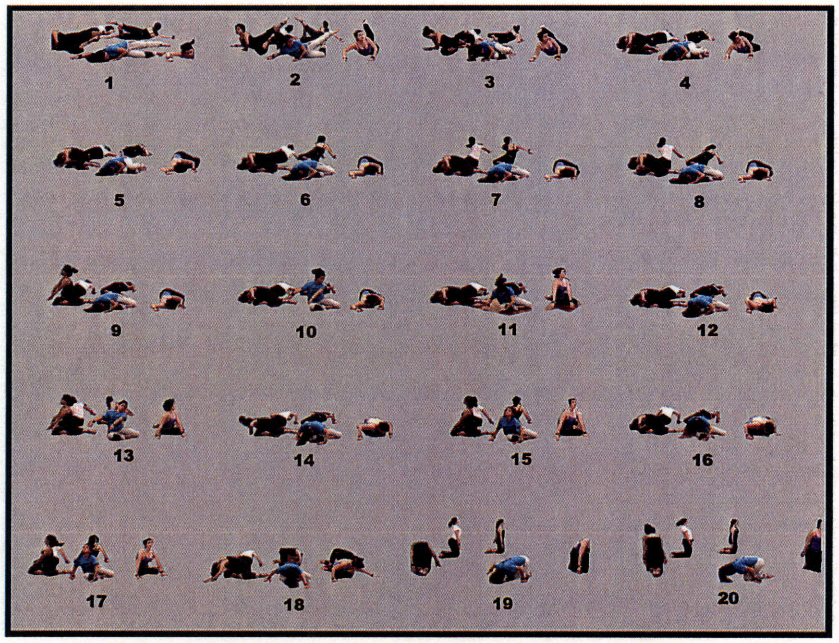


Figure 8-2: Selected video frames showing a segment of dance with individual gestures that progress from dancer to dancer. Features and musical parameters generated from this segment are shown in Figure 8-4.

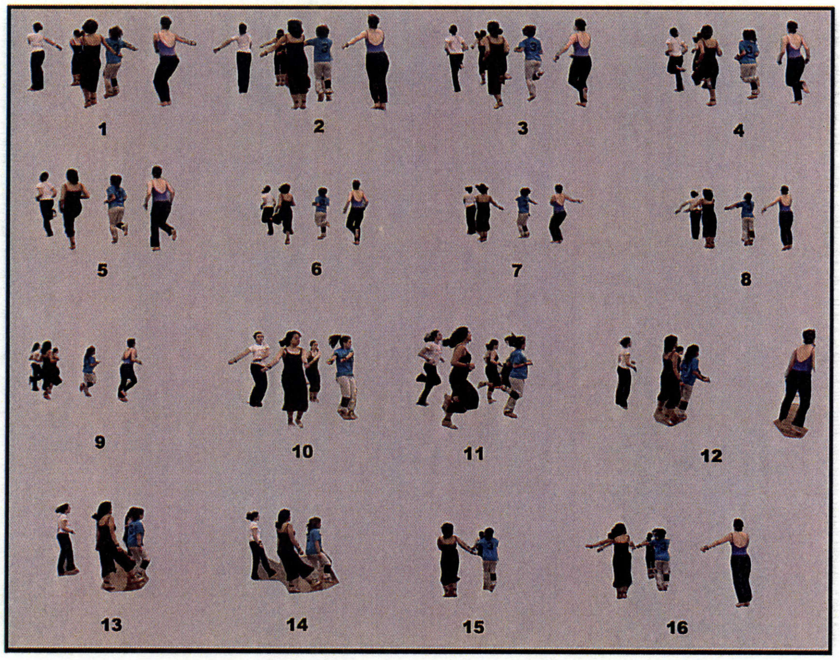
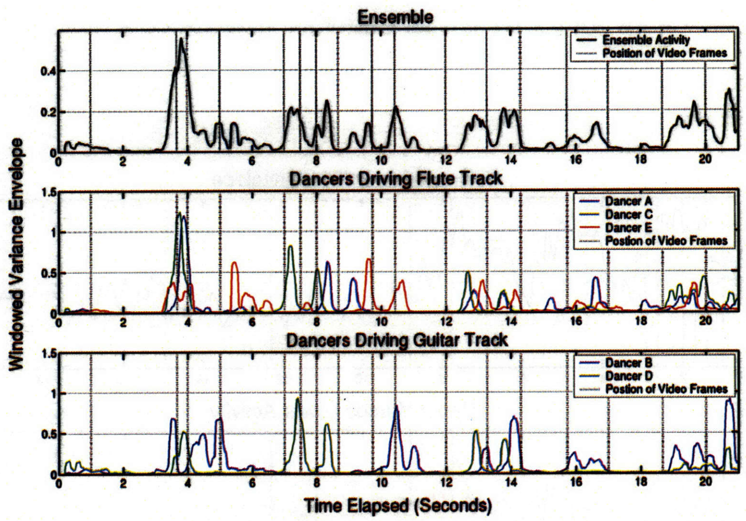
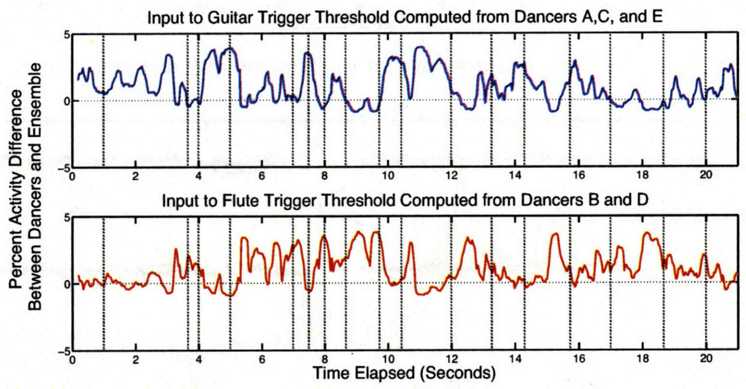


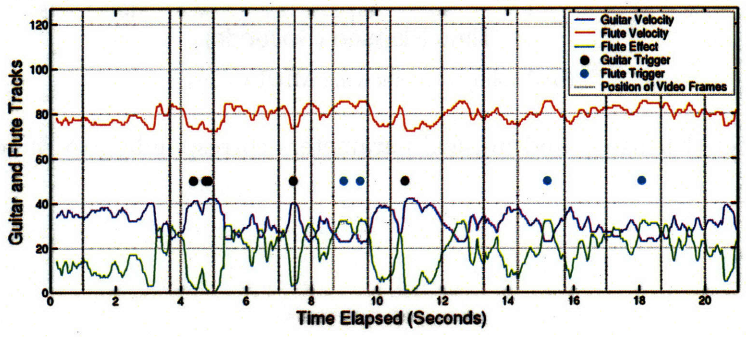
Figure 8-3: Selected video frames showing a dance segment with high levels of correlated activity. Features and musical parameters generated from this segment are shown in Figure 8-5 and 8-6.



(a) Ensemble and personal activity features used to control guitar and flute.

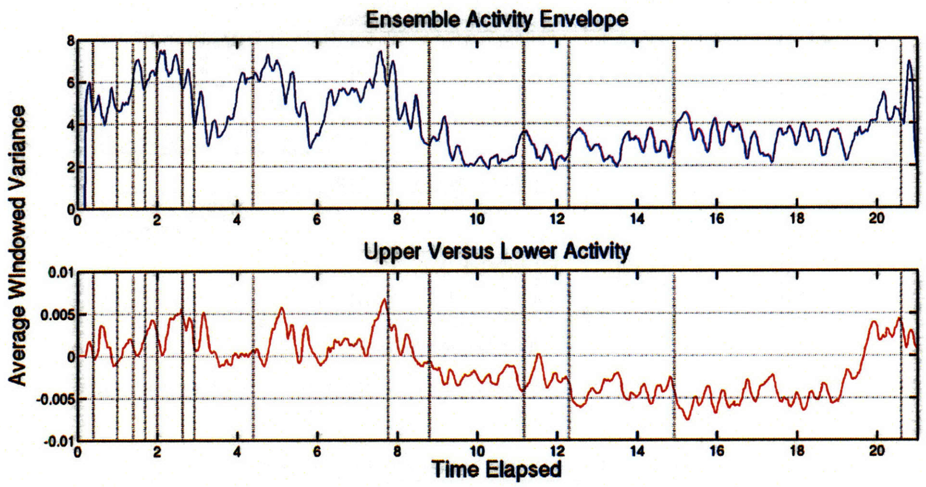


(b) Activity deviation features computed by comparing personal and ensemble activity envelopes.

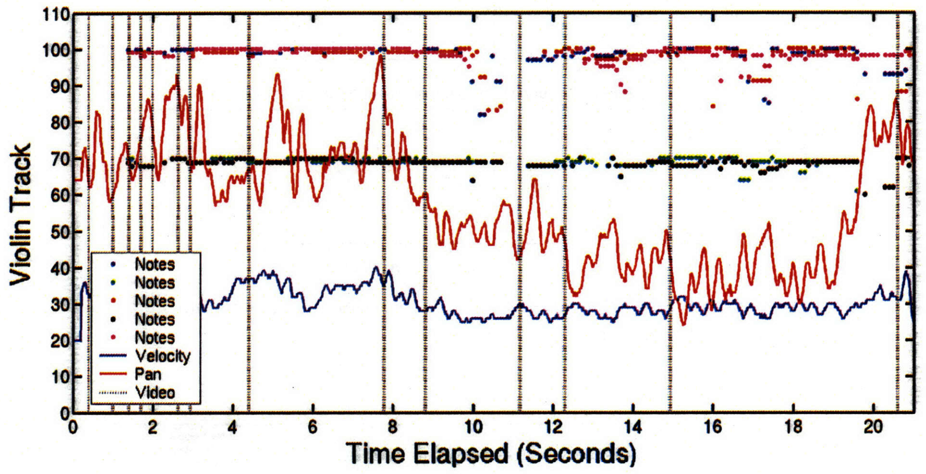


(c) Guitar and flute tracks showing MIDI values and note triggers.

Figure 8-4: Selected features and musical parameters corresponding to the dance segment in Figure 8-2.



(a) Activity features relevant to violin velocity and pan.



(b) Violin track showing MIDI values.

Figure 8-5: Selected features and musical parameters corresponding to the dance segment in Figure 8-3.

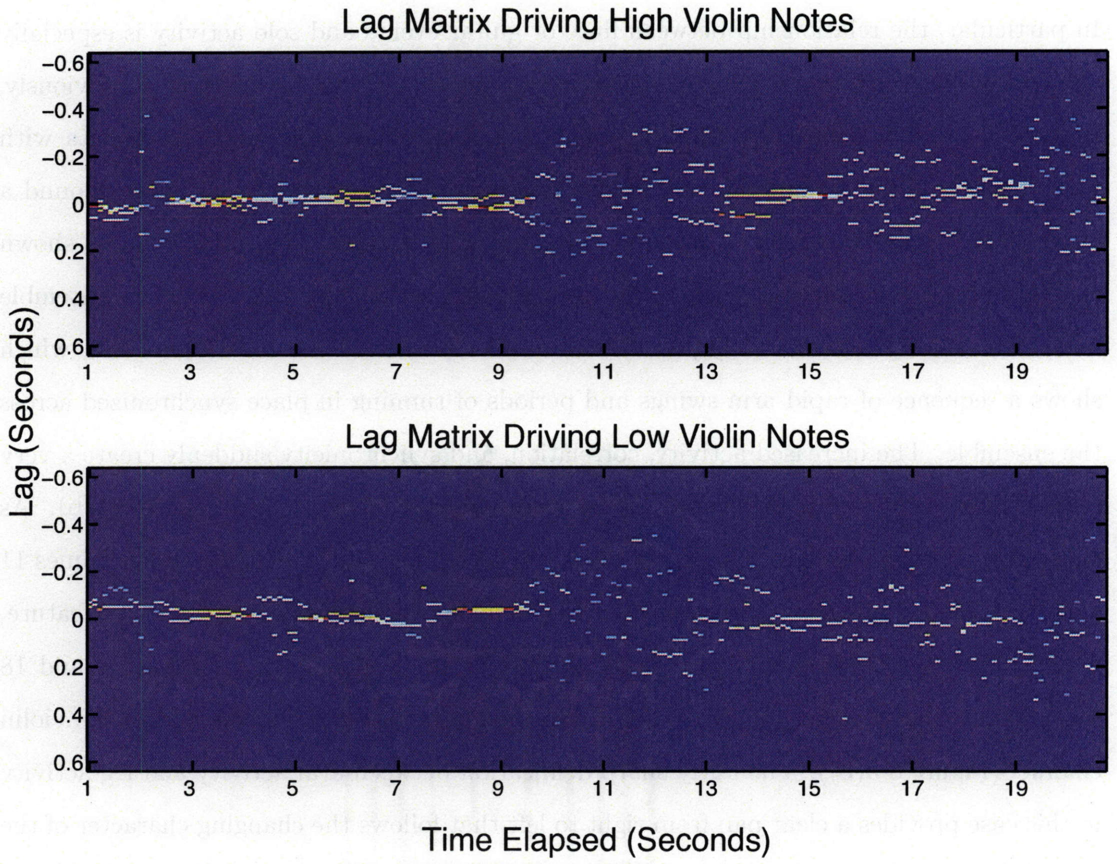


Figure 8-6: Average lag features between dancers, corresponding to the segment in Figure 8-3, used to determine violin pitch and note triggers.

8.5.2 Musical Output

As a demonstration application, the musical aspect of this trial focused on simple interactions, simple sounds, and contained little structure imposed outside of the data itself. Therefore, a raw or stochastic quality is evident in the output, including random bursts of note entrances and very few musical relationships between instruments. Still, in a number of instances, the mapping from data to sound manages to be effective.

In particular, the relationship between flute or guitar events and solo activity is especially clear when very little movement is occurring in the ensemble. As mentioned previously, Figures 8-2 and 8-4 illustrate the alignment of musical parameters and sensor data with video snapshots for a short dance segment containing solo gestures which move around a circle from person to person. Flute and guitar entrances stand out prominently, as shown in Figure 8-4(c). Conversely, the violin sounds are particularly effective when ensemble movement is very active and highly synchronized. This is the case in Figure 8-3, which shows a sequence of rapid arm swings and periods of running in place synchronized across the ensemble. The increased activity, correlation, and synchronicity suddenly create a very tense, thick texture of high pitched violin notes, which can be seen in Figure 8-5(b). As the segment progresses, arm movement gives way to leg movement between video frames 11 and 14. This behavior is clearly reflected in the upper versus lower body activity feature, which indicates predominant lower body activity in the region between about 10 and 18 seconds into the segment (Figure 8-5(a)). This in turn controls the stereo pan of the violin channel (Figure 8-5(b)). The fairly sharp delineation between arm activity and leg activity in this case provides a clear pan from right to left that follows the changing character of the dance. The pitch and note density of the violin also changes subtly in this region, adding to the response.

Not surprisingly, the least effective dance segments in terms of the mapping designed here are those which are hard to interpret visually as containing highly correlated or highly contrasting activities. During these segments, the generated sounds seem to lose interpretability as well. Part of the difficulty is the practice of triggering sounds on feature thresholds. By nature, this creates random note entrances and behaviors that are not robust to changing conditions. When the features clearly support a certain state, the random entrances cluster into patterns that seem to follow the input. However, when the features are ambiguous, the random entrances become clutter. In addition, relying on triggering makes the system sensitive to latency, and the use of single empirical thresholds is typically an oversimplified decision process.

Of course, triggering shows up in this design because of the MIDI protocol and the Reason interface, which rely on note events with strict beginnings and ends. A different control environment could be better suited to an ambient setting where notes are driven in and out with continuous parameters. Still, it is unclear whether this structure would help clarify the relationship between movement and sound during ambiguous states. Given its intended purposes and deliberate simplicity, the mapping described here generated satisfying results.

8.5.3 Directions for Improvement

The most pressing direction for improvement is to add live feedback so that dancers can respond to the music they are generating. In terms of getting a meaningful output from the system, if would appear that the designer is at an advantage with the current setup, in which pre-recorded data is played back and as such is completely repeatable. However, without the dancers being able to respond to the musical output, there is no feedback or adaptability to lend shape to the mapping. Once live interaction can occur, the relationship between movement and sound will be clearer because of the dialog between performer and computer. In this situation, a mapping can also be learned and explored to push its most useful aspects. Most likely, when the system is tested with real-time feedback, more satisfying musical output will be posible. More complex mappings can then be developed, potentially incorporating machine learning and adaptability in software. Eventually, the system should support a satisfying live performance.

Further development of the feature set and analysis strategies may also be necessary to improve the system. In particular, many of the features discussed in this work have relatively high latencies which compromise the intent of real-time data collection. Future implementations could focus more aggressively on simpler, low-latency features. At the same time, the detection and interpretation algorithms can become more sophisticated to glean information from the data with a higher degree of detail and reliability. Certain pattern classification algorithms, such as support vector machine (SVM) or tree based approaches, can be relatively fast to evaluate once they have been trained. These techniques can be used to process windows of data to look for a specific gesture, activity characteristic, or event with much higher dependability than single thresholds. A hidden Markov model (HMM) can be used to track progressions from gesture to gesture, although the application of HMMs to inertial sensor data has been met with mixed results [45]. Machine learning techniques were avoided previously, mainly because of the need to generate and process a large amount of training data. Now that the means for generating this data are in place, they would be useful to investigate. Since the current implementation only considers inertial data, in would also be interesting to incorporate the capacitive sensor into the next iteration of feature analysis.

Even with machine learning techniques employed to refine the feature extraction process, the potential amount of information expressed by all of the descriptive elements of group motion suggested in this work is still cumbersome for direct mapping to a general musical palette. The problem can be simplified by interpreting group dynamics in the context of a specific piece. For example, the music can be generated from a loose framework or score designed alongside the choreography. At a given point in the score, one may be looking for a specific set of possible changes in the dancers movements that signal musical events such as changing timbral qualities, the entrance of a new melodic line, or a shift to a new section. By placing contextual limits on the decision space, pattern recognition algorithms can be trained on a specific performance to streamline the control process. Although the dancers do not actually generate music directly under this model, they are able to freely control their progression through sections of the score, alter their interpretation of the context, and add embellishments. This approach should provide a balance between musical continuity and the sense of causality between the movements of the dancers and the generated sound, which is essential for an engaging interactive performance.

Chapter 9

Conclusions

9.1 Summary

In summary, this thesis has presented the design of a system of compact, wearable, wireless inertial sensing devices, as well as their application in analyzing human motion and providing real-time feedback, especially for interactive dance and music. With dance performance applications in mind, the design effort has been focused on building a platform for scalability, speed, durability, distributed measurement, and interpretation of group interactions. The novelty that has been explored in this area is the insistence on instrumenting entire ensembles without sacrificing the measurement resolution typically reserved for a single user. With this requirement satisfied, the result is a high performance system equally applicable in any situation where human motion analysis is coupled with the need for high data rates.

The sensor network is made up of wristwatch-sized nodes that can be attached to various locations on the body, typically wrists and ankles. Each node contains a full six-degree-of-freedom IMU and a capacitive-node-to-node proximity sensor, as well as its own 1Mbps wireless radio module. A central computer controls the network and collects data via a USB basestation. Using a simple TDMA scheme, one basestation can handle up to 25 nodes sending full state updates at 100Hz. Thus, the current system can potentially operate with an ensemble of up to 6 dancers wearing sensors on each limb.

So far, the system has been successfully tested with a group of five dancers and 20 nodes running simultaneously. The results of this study show that it is possible to collect and analyze data quickly enough to generate meaningful feedback that responds to dance with tolerable latencies. More work is required to find features and analysis algorithms that minimize these latencies while at the same time creating more sophisticated mappings to improve the quality of the output. In an effort to extend beyond dance applications, the system has also been evaluated in a preliminary study measuring the arm movement of professional baseball pitchers. In this case, fewer sensors were used, but with higher sampling rates. With some modifications, the sensors were able to measure the extremely high accelerations involved with more temporal precision than a state of the art motion capture system. Statements as to the accuracy of these measurements and their bearing on traditional motion capture practices will require further study and rigorous calibration.

9.2 Evaluation

9.2.1 Design

Hardware design was the most significant aspect of this project, as the results could not be achieved with any previously existing system. The choice of MEMS inertial sensor components was straightforward, and their capability in fine gesture tracking and activity classification applications is well documented. Fewer inertial systems have been documented for very strenuous movement, and fewer still demonstrate scalability to multiple users. The $\pm$10g and $\pm$1,200 deg/sec devices used here in conjunction with a 1Mbps radio network are more than enough to balance high ranges and scalability with high resolution. The devices are also relatively small, light, and rugged enough to be worn comfortably in a dance situation.

However, some improvements are left to be made. The first and most general is to make the device smaller, from a bulky replacement for a wristwatch to a unit fitting inside an existing wristwatch, and eventually even smaller. Already, the size could be reduced by about a factor of two by adding layers to the printed circuit board to accommodate the traces which take up a significant portion of the current board area. Further reductions could be made quite easily with the recent advent of multi-axis gyroscopes, which obviate the cumbersome sensor daughtercards, although the performance of such gyroscopes has not been evaluated. Radio devices are becoming available with increasingly powerful integrated MCUs, which may quickly eliminate the need for a dedicated MCU as well. Of course, as more systems become integrated, the progression moves towards a single die or stacked die implementation which could house an entire IMU within the size of a ring [79, 80]. As this happens, it will become increasingly easy to integrate devices into clothing, making the applications described here more feasible and accessible. However, this may not yet be cost effective.

More immediate hardware improvements might include changing daughtercard layouts to accommodate the more appropriately ranged ± 5g accelerometers or a 3-axis accelerometer, reducing area and cost by replacing the flexible high-efficiency 5V regulator with a cheaper solution specifically intended for the current battery pack and requiring fewer external components, or upgrading to the new 2Mbps radio. The capacitive sensor system performed very well, considering its inclusion in the design as a supplement. Its range limitations may be mainly physical, but could potentially be improved by increasing the transmit voltage and using an op-amp with a lower noise floor. In general, the hardware design has been more than adequate by providing a very stable and reliable foundation on which to experiment freely with applications and analysis.

The communications structure and associated TDMA scheme has been a very effective way of sharing the network among as many devices as required here, while enforcing low-latency and keeping a very simple firmware design. The only major drawback is the insistence on arbitrating the nodes from the basestation on a sample-by-sample basis. Although this keeps the samples perfectly synchronized across the network, it becomes a problem when broadcast packets are dropped or the basestation falters in its transmission pattern. The design was intended to cope with significant clock skews between the nodes. In reality, the clock skew appears to be small enough that a node could remain synchronized with the basestation for several samples without being reset. Thus, a node could continue to operate by at least storing data for several samples when the communication from the basestation is lost, potentially recovering from short dropouts that would normally result in lost data. This behavior would also have an advantage in that the node could save power by duty cycling the radio while it waits to regain contact with the basestation, rather than leaving it in receive mode continuously.

The other major communications improvement could be made in basestation firmware by eliminating the dependence on a proprietary API for USB communication. Rather than using bulk transfers, the basestation could operate as an interrupt transfer device, hopefully resulting in a more responsive communications link to the host computer. The other advantage would be enhanced operability in MacOSX, which does not support the proper drivers for the current basestation design and causes difficulties as a result. Designing the USB device firmware from the ground up would also allow other customizations and enhancements, such as making the system recognizable as a human interface device. These efforts will improve compatibility and reduce the burden on application software.

9.2.2 Application

The applications and analysis techniques described in this work were mainly intended to test and validate the design. For this purpose, they performed quite well. Firstly, it was demonstrated that features could be extracted in order to make relevant conclusions about the data. This was achieved for both dance and baseball applications. Then, various pieces of software were written to show that data could be collected in real-time from the entire network. Finally, features were computed on streaming data from dance and processed to produce music with low-latency. The final step, linking collection to analysis and response so that dancers can listen and react to the system, will be discussed below.

9.3 Future Work

Primarily, the goal for future development would be to run the system in a live situation where dancers can hear the musical feedback they generate. Although sound was generated from a streaming file of pre-recorded data for research purposes, and all of the elements are now in place, the system has not yet been used in a live context with a full ensemble. Three or four sensor nodes have been used together to control sound for various small scale demonstrations, but the mapping considerations are different and the network requirements are not as impressive without full deployment. Apart from clearly illustrating the degree of low-latency processing possible with a full group, using the sensors in a live interactive setting is crucial for further development of analysis mapping algorithms. This is because the ability of dancers to learn and react to responses adds a new dynamic and malleable layer, indeed the whole purpose of an interactive system. Additionally, closer collaboration with experienced dancers, choreographers, and composers will shed light on which features and mapping strategies are most relevant to real parameters.

New directions for analysis and mapping will most likely strive for more aggressively minimizing latency and adding to the subtlety of the response. As mentioned previously, this implies the divergent courses of faster, simpler base features and the addition of elaborate pattern recognition techniques to the detection and mapping process. In many cases, when applying pattern recognition to human motion analysis the first instinct is to detect gestures. Given the depth and difficulty of the problem, pattern recognition was completely avoided here in favor of driving musical feedback directly from more general features without interpreting them in a gestural context. However, trying to detect specific gestures is not necessarily a better approach. Inevitably, a full gesture cannot be detected until it is complete, meaning that gesture recognition suffers from the same type of latency as cross-covariance features. Also, once gestures have been determined, in the case of dance interpretation it is very likely that movements across multiple people will be combined and brought back down to the vague level of group activity features. Still, the activity envelopes and thresholds employed in this work are too general and unreliable for more involved mapping. Eventually, pattern recognition will be necessary, but in a way more conducive to group activity, by recognizing styles rather than gesture. It seems by experience that the brain interprets collective motion, dance in particular, by emphasizing style, some palpable quality of movement, rather than by picking apart individual movements. This interpretation would be more valuable for generating musical responses that have meaning to a human audience. A response to changing style, being a slow process, is also more tolerant to latency.

As suggested previously, one way to control music in the context of slowly varying styles is to follow a score, which shifts characters as the ensemble moves through different sections of a dance piece. Changing the control space may also be useful. MIDI protocol favors situations in which events must be triggered, leaving latency issues glaringly obvious, and sparse changes like the transition between styles reflected in sparse music. Pursuing new models for mapping outside of the MIDI may be the next stage of the musical aspect of this design. Fitting the pieces of interaction, analysis, and musical structure together should eventually involve a performance piece to showcase the technology.

Further development is also required for the purposes of moving forward with athletic training applications. In this case, the next steps are simply to work on calibrating sensors and comparing their measurements directly with the video motion capture system to determine the degree of accuracy. Another trial study with improved synchronization between the two systems and working gyroscopes may be necessary to begin this process. Additional work will explore the application of the system to athletic training and evaluation.

9.4 Outro

In conclusion, a high-speed sensor network large enough to instrument the arms and legs of a modest dance ensemble has been built and tested with real-time data collection. Furthermore, the analysis, interpretation, and effective realization of sensor data as sound has been accomplished with low-latency and with reasonable processing power. The implementation developed in this work has been successful in generating several collective activity features relevant to dance, and preliminary results suggest that the focus on these group features and the ability to distribute measurement over multiple people has intriguing possibilities for performance art. Although the concept of a collaborative interface with inertial sensors has only been roughly outlined here, the capabilities of this system provide a huge space for future exploration that should be taken advantage of. The benefits do not stop with interactive art and human interface research, however. This work has demonstrated, in preliminary testing, that the system will eventually be capable biomotion capture at the speeds and resolutions necessary for physiological analysis of professional athletes. Thus, distributed, wireless, inertial sensing may well find applications in human biomechanics, spanning the range from high-performance athletic research to consumer physical training tools. In many of these cases, the ability to generate instant feedback in a natural environment fills an important pedagogical role that would be an improvement to existing measurement systems. Towards this goal, that is, of improving the value and accessibility of human motion analysis tools, this thesis has hopefully established one viable model for future progress.

Appendix A

Schematics and PCB Layouts

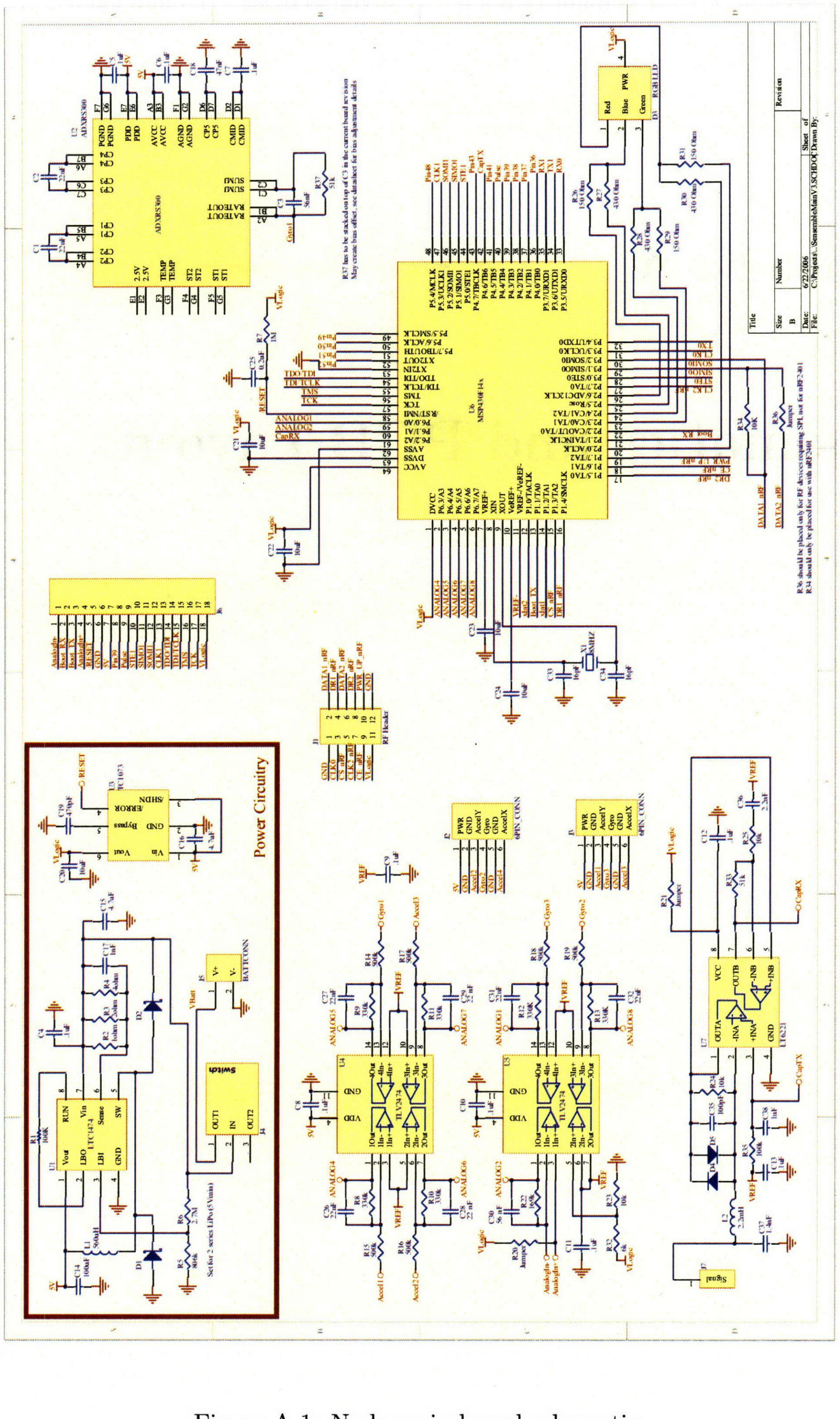


Figure A-1: Node main board schematic.

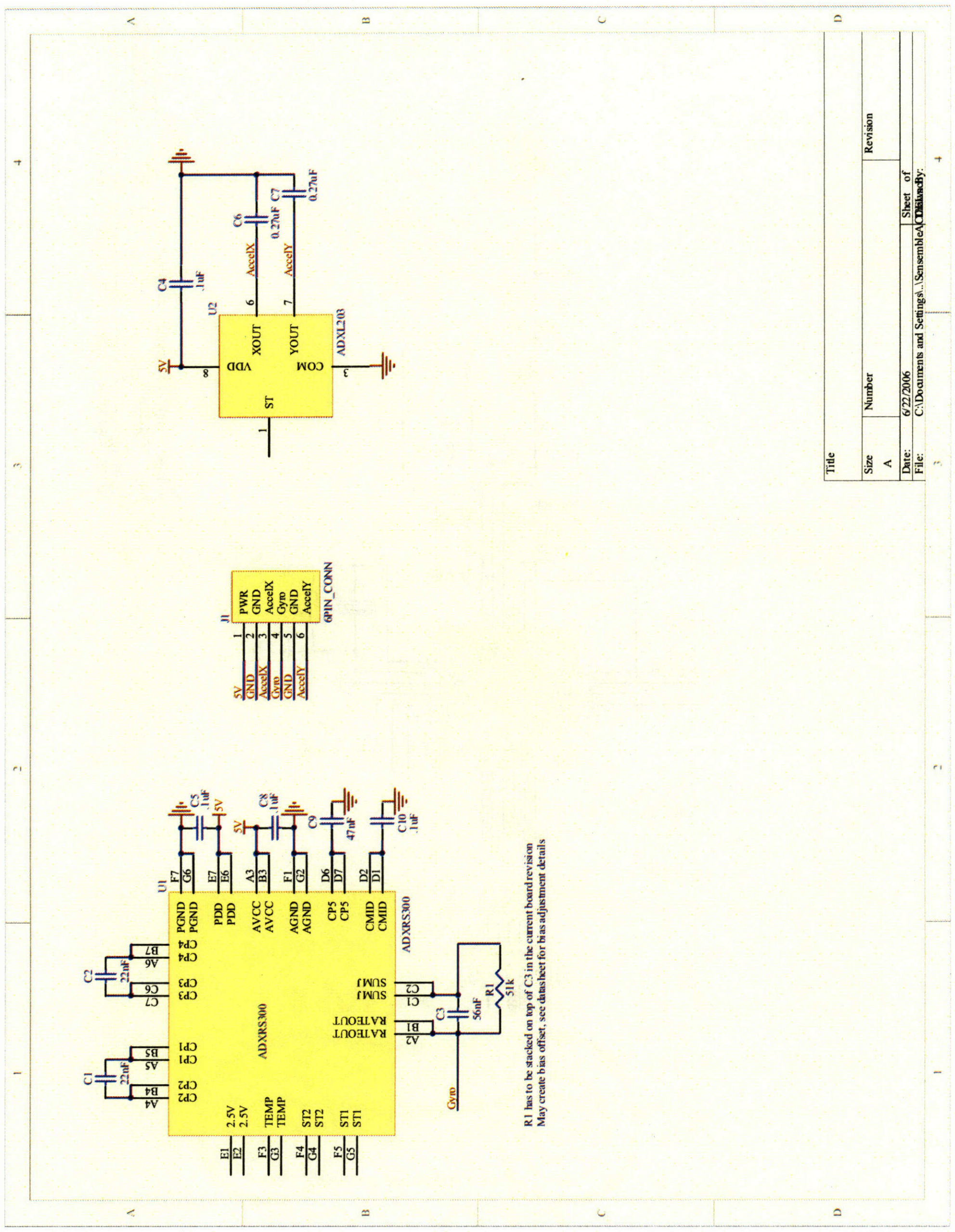


Figure A-2: Sensor daughtercard schematic.

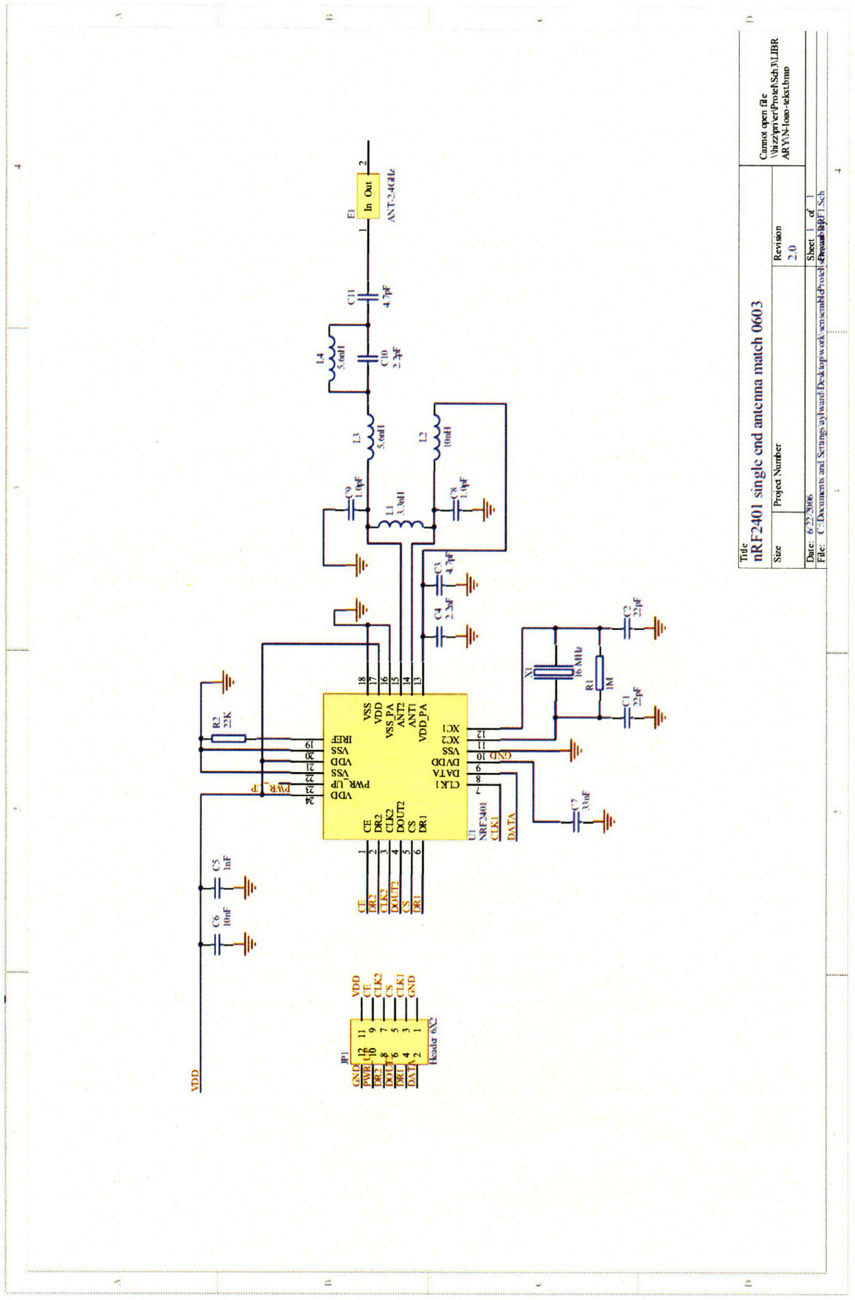


Figure A-3: RF daughtercard schematic.

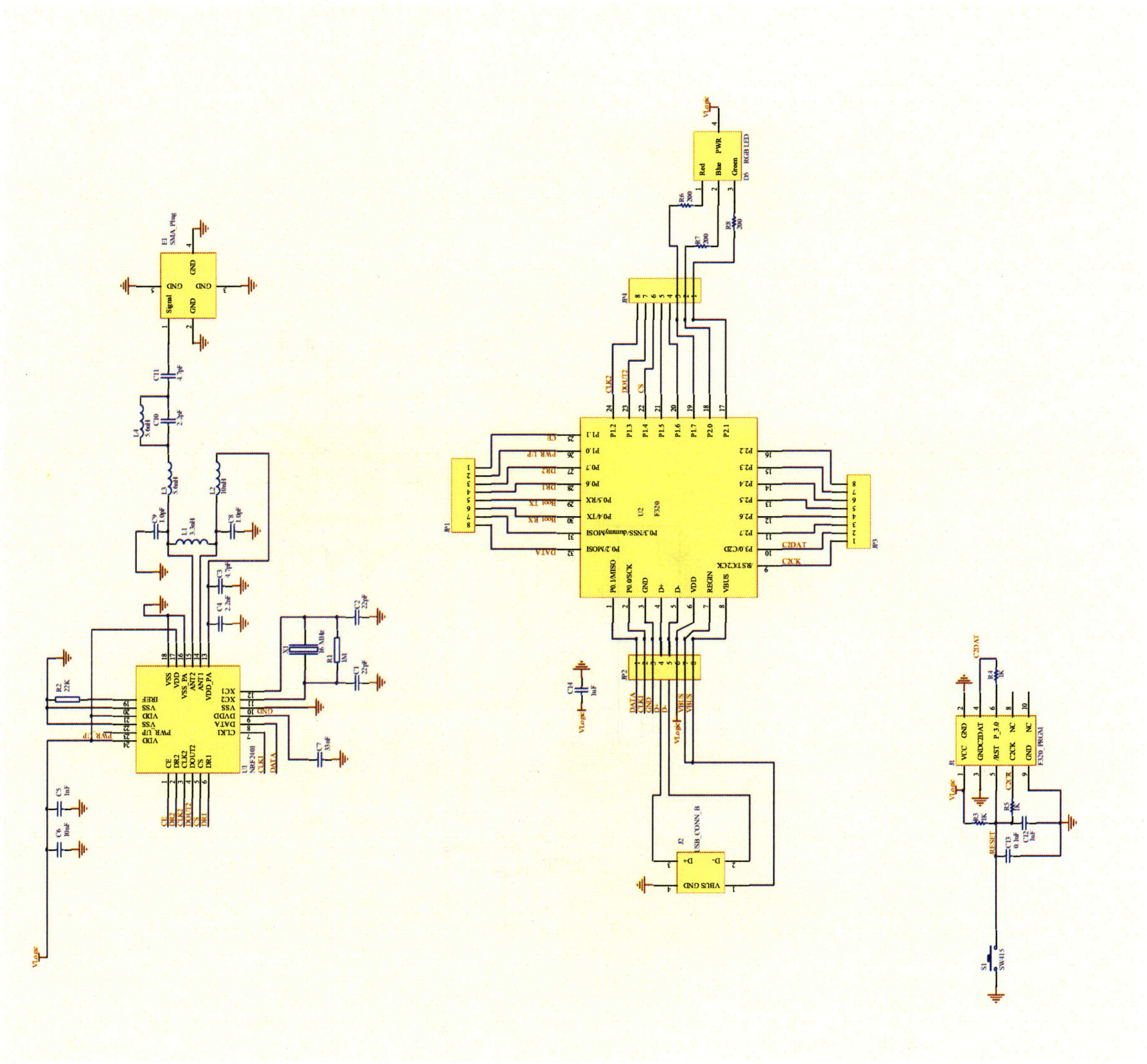


Figure A-4: Basestation schematic.

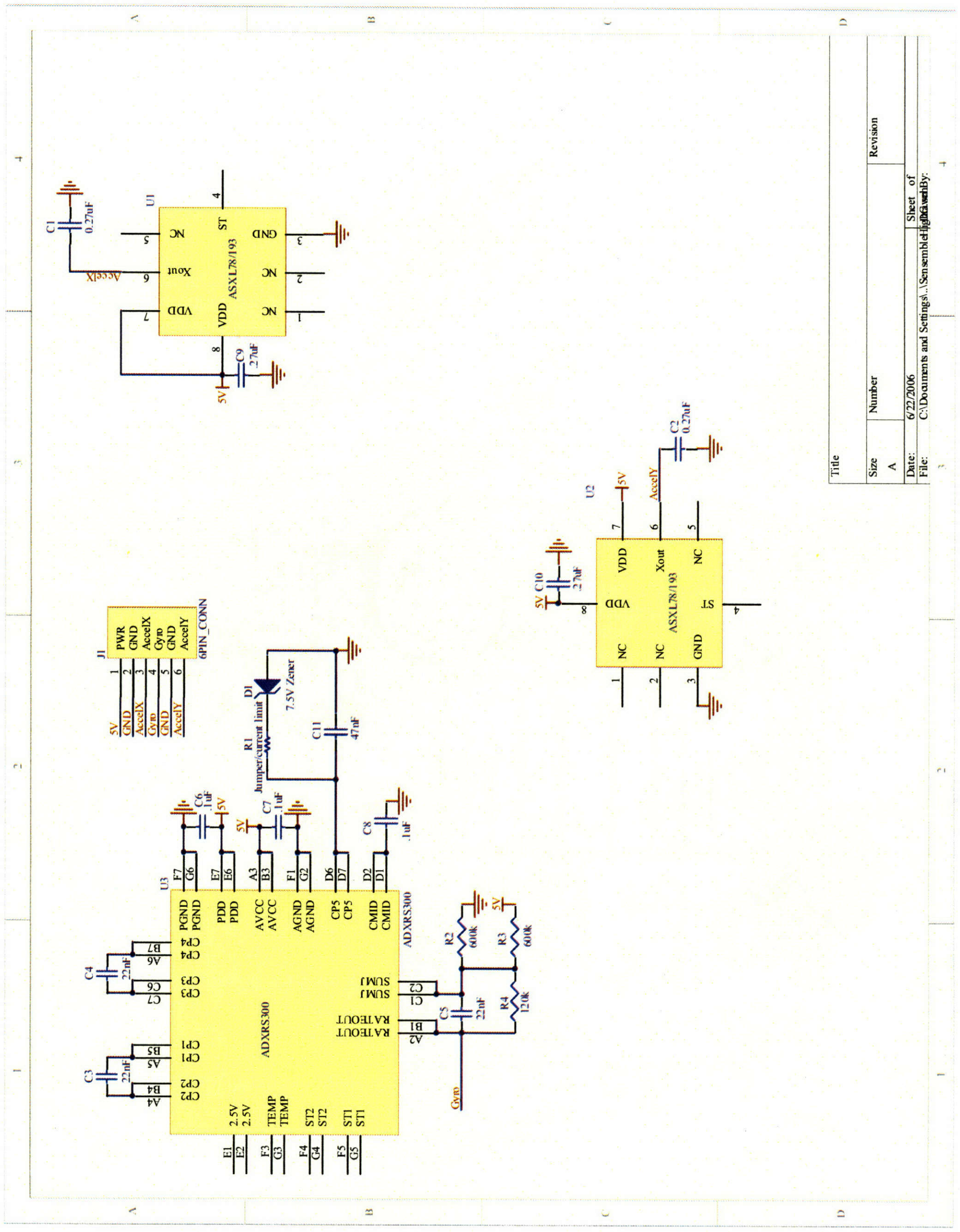


Figure A-5: High range sensor card schematic.

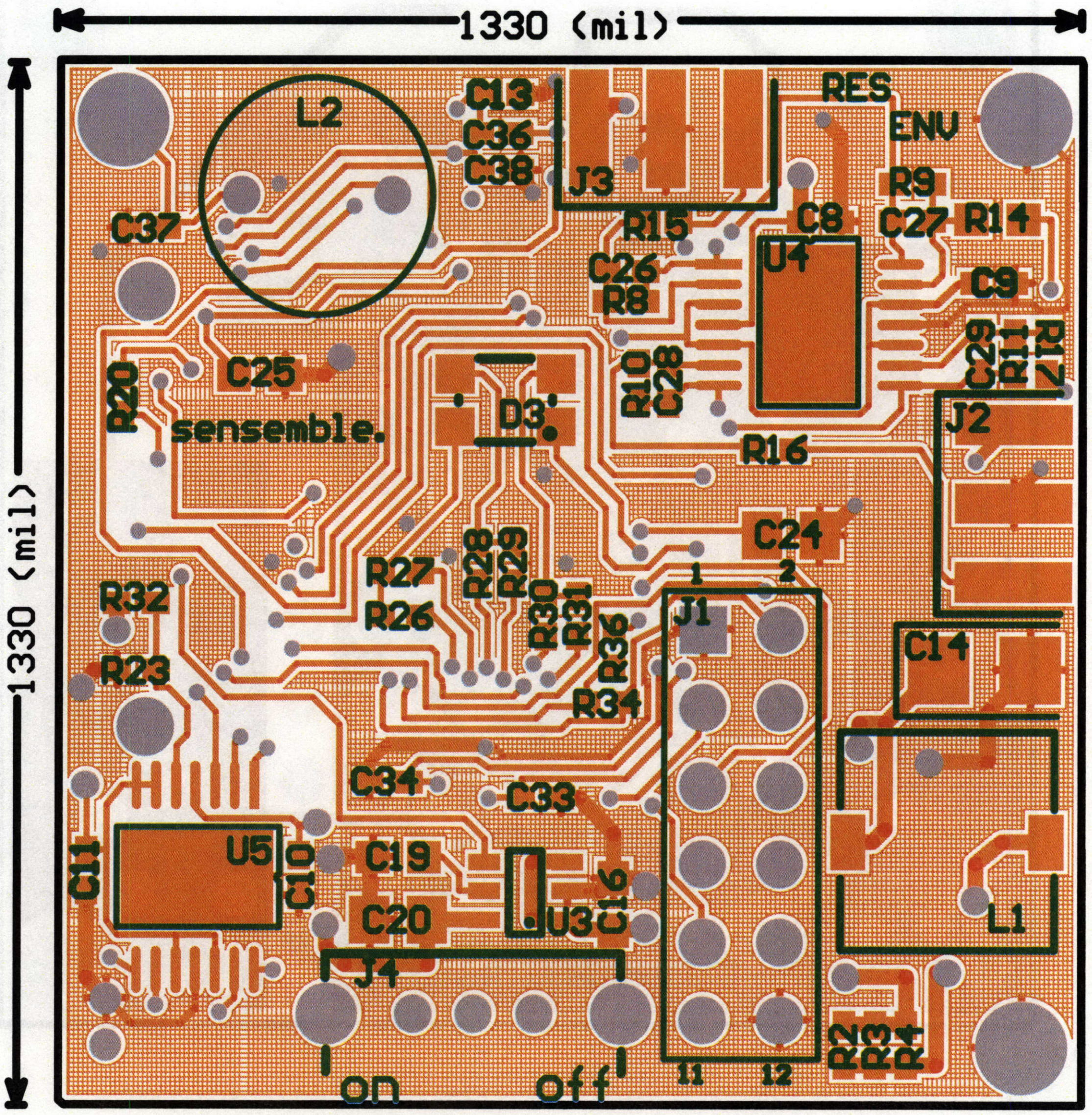


Figure A-6: Node main board layout, top layer.

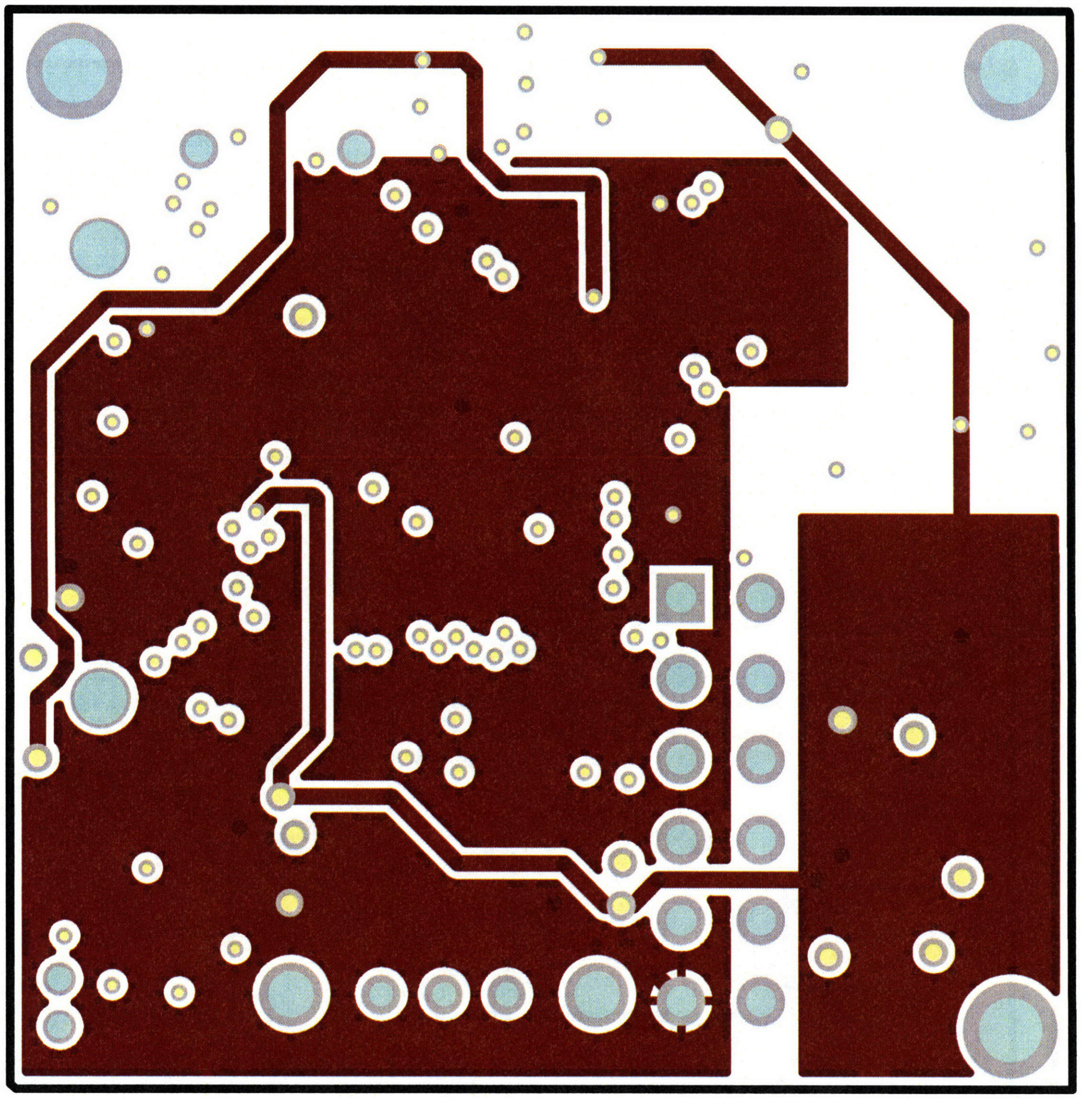


Figure A-7: Node main board layout, internal layer.

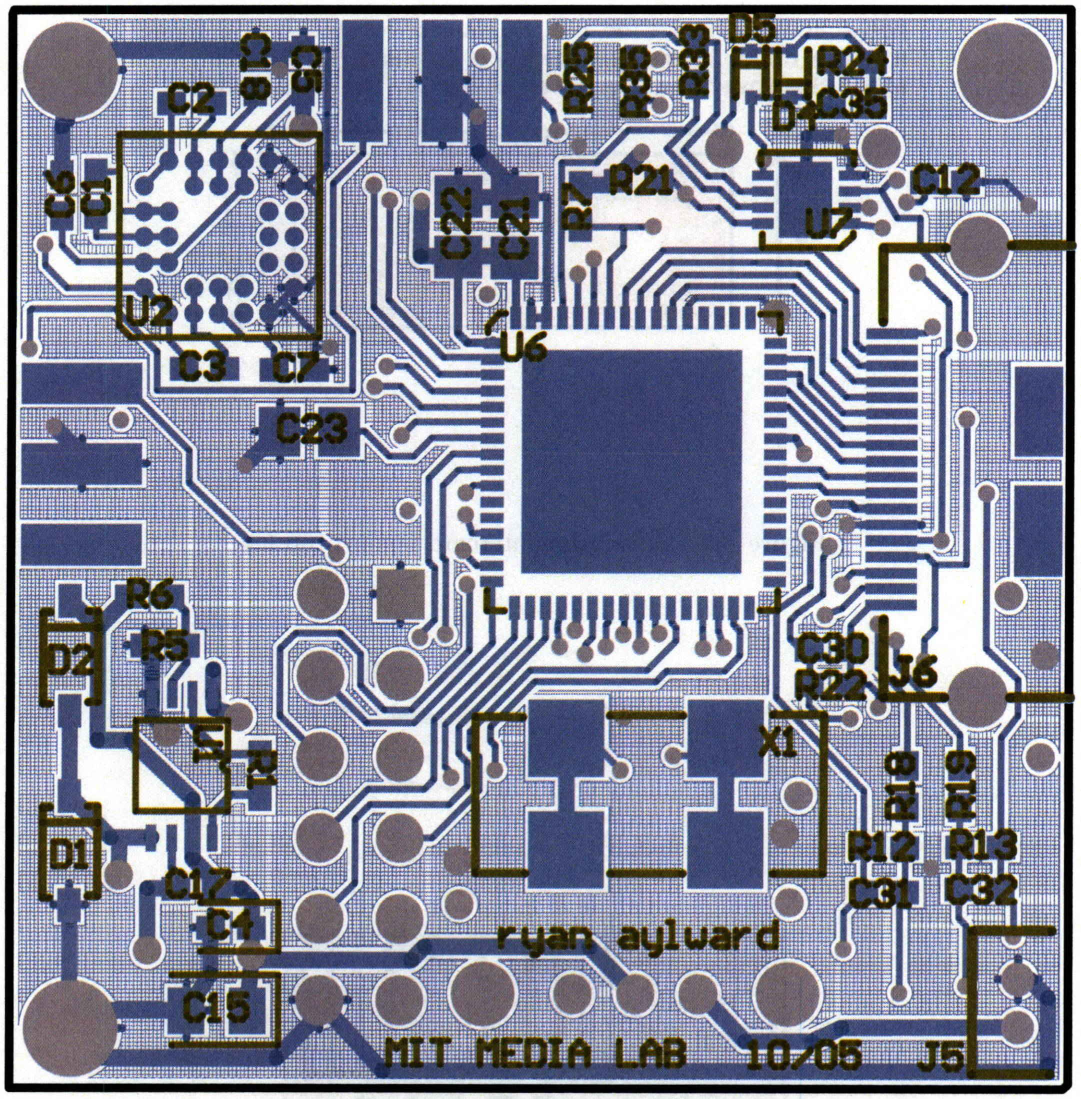


Figure A-8: Node main board layout, bottom layer.

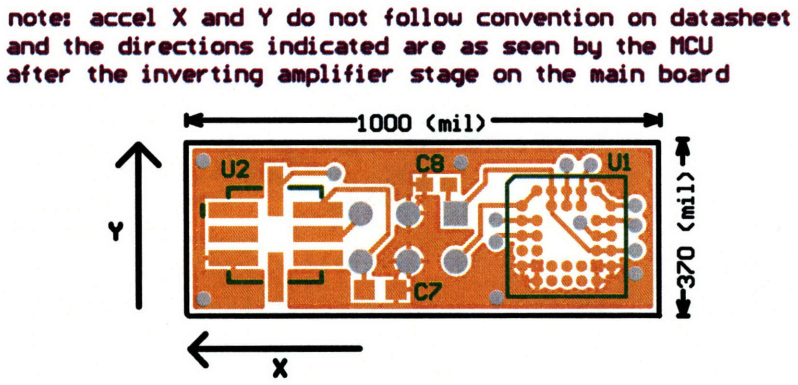

note: accel X and Y do not follow convention on datasheet
and the directions indicated are as seen by the MCU
after the inverting amplifier stage on the main board
1000 (mil)
370 (mil)
U2
C8
U1
C7
Y
X

Figure A-9: Sensor daughtercard layout, top layer.

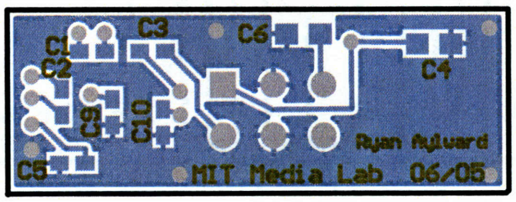


Figure A-10: Sensor daughtercard layout, bottom layer.

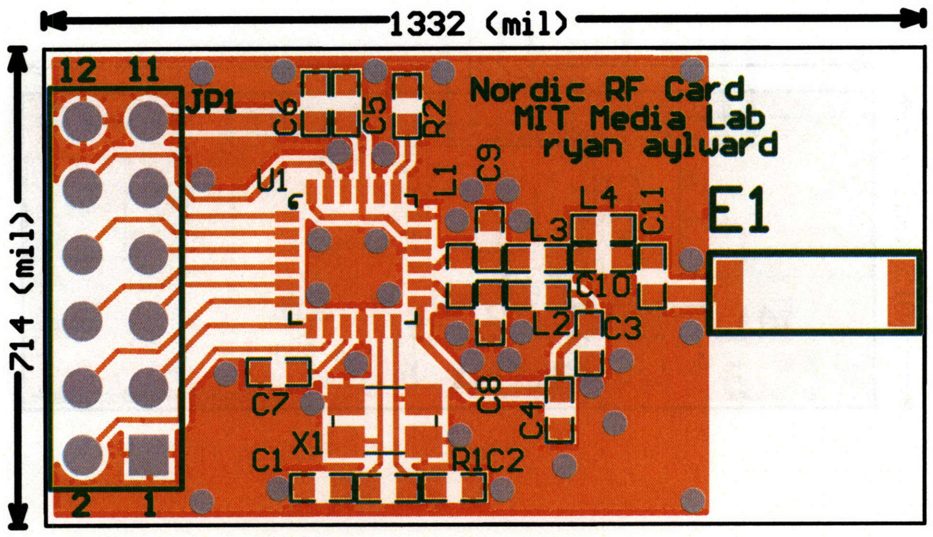


Figure A-11: RF daughtercard layout, top layer.

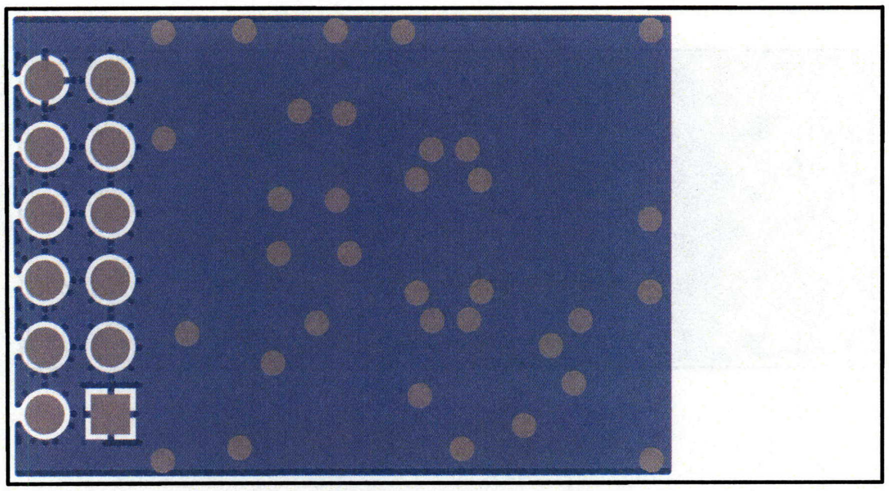


Figure A-12: RF daughtercard layout, bottom layer.

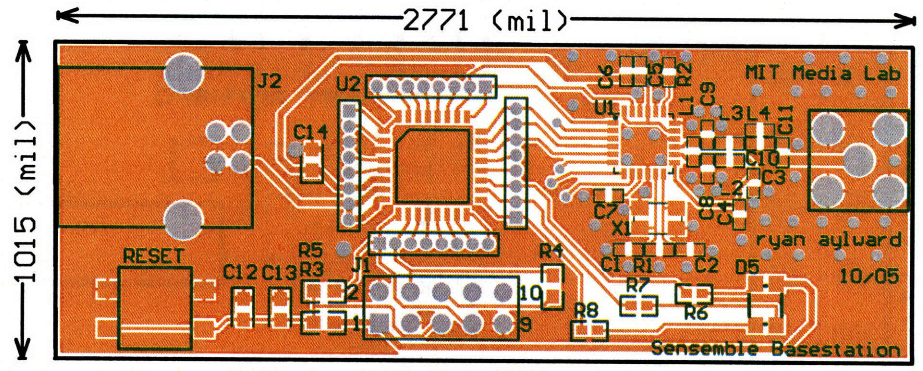


Figure A-13: Basestation layout, top layer.

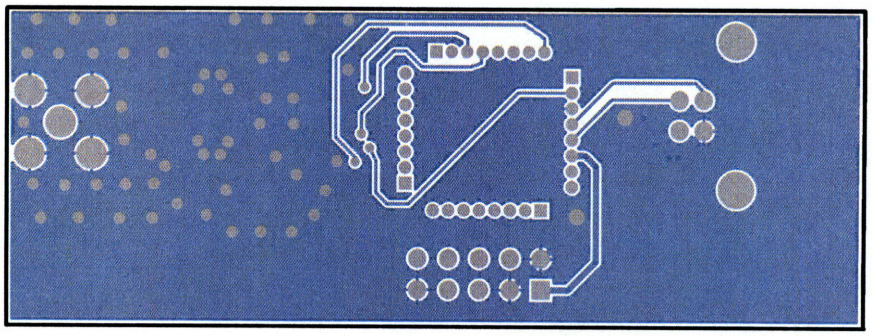


Figure A-14: Basestation layout, bottom layer.

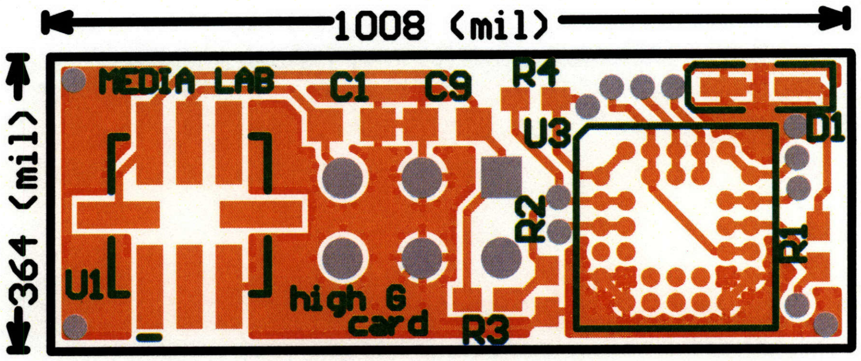


Figure A-15: High range sensor card layout, top layer.

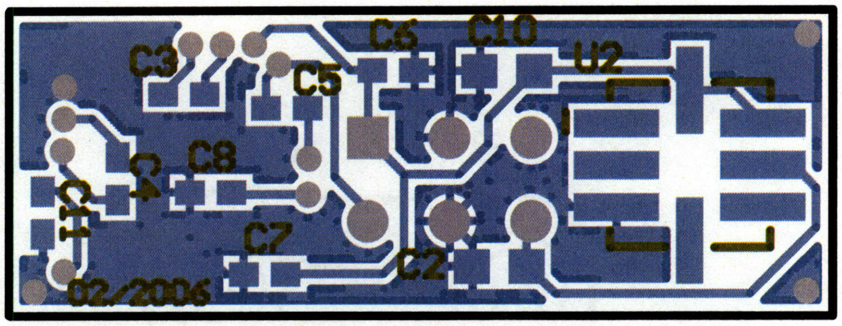


Figure A-16: High range sensor card layout, bottom layer.

Appendix B

Bill of Materials

Main Board										
Description	Value	Designator	Footprint	Layer	Special Instructions	Order Quantity/Node	Price/1u	Price/10u	Price/25u	Total / Node
Capacitor	22nF	C1	0402	Bottom		15		0.135		0.2052
Capacitor	22nF	C2	0402	Bottom						
Capacitor	22nF	C3	0402	Bottom						
Capacitor	.1uF	C4	0402	Bottom		18		0.135		0.243
Capacitor	.1uF	C5	0402	Bottom						
Capacitor	.1uF	C6	0402	Bottom						
Capacitor	.1uF	C7	0402	Bottom						
Capacitor	.1uF	C8	0402	Top						
Capacitor	.1uF	C9	0402	Top						
Capacitor	.1uF	C10	0402	Top						
Capacitor	.1uF	C11	0402	Top						
Capacitor	.1uF	C12	0402	Bottom						
Capacitor	.1uF	C13	0402	Top						
Capacitor	100uF	C14	TC3528-1411	Top	polarized - band on package lines up with band in placement guide	1		10.88	28.01	1.1204
Capacitor	4.7uF	C15	0805	Bottom		1		2.25		0.27
Capacitor	4.7uF	C16	0603	Top		1		0.19		0.0228
Capacitor	1nF	C17	0402	Bottom		2		0.19		0.038
Capacitor	47nF	C18	0402	Bottom		3		0.19		0.0456
Capacitor	470pF	C19	0402	Top		1		0.19		0.0228
Capacitor	10uF	C20	0805	Top		5		1.94		1.0088
Capacitor	10uF	C21	0805	Bottom						
Capacitor	10uF	C22	0805	Bottom						
Capacitor	10uF	C23	0805	Bottom						
Capacitor	10uF	C24	0805	Top						
Capacitor	0.22uF	C25	0402	Top		1		0.69		0.0828
Capacitor	22nF	C26	0402	Top						
Capacitor	22nF	C27	0402	Top						
Capacitor	22 nF	C28	0402	Top						
Capacitor	22 nF	C29	0402	Top						
Capacitor	56 nF	C30	0402	Bottom		1		0.184		0.02208
Capacitor	22nF	C31	0402	Bottom						
Capacitor	22nF	C32	0402	Bottom						
Capacitor	16pF	C33	0402	Top		2		0.45		0.09
Capacitor	16pF	C34	0402	Top						
Capacitor	2.2nF	C35	0402	Bottom		2		0.19		0.038
Capacitor	2.2nF	C36	0402	Top						
Capacitor	1.3nF	C37	0603	Top		1		1.94		0.2328
Capacitor	1nF	C38	0402	Top						
Schottky Diode	mbr0530t	D1	SOD123	Bottom	polarized - cathode indicated on board silkscreen	2		1.75		0.35
Schottky Diode	mbr0530t	D2	SOD123	Bottom	polarized - cathode indicated on board silkscreen					
RGB LED	QTLP650DRGBTR	D3	Fairchild RGB	Top	green marking on package lines up with circle in placement guide	1	4.16	41.6	104	4.16
Diode		D4	SOD-523	Bottom	polarized - cathode indicated on board silkscreen	2	0.27	2.7	5.06	0.4048
Diode		D5	SOD-523	Bottom	polarized - cathode indicated on board silkscreen					
Header 6x2		J1	Header 6x2	Top		1		1.396	3.49	0.1396
Header 3x2		J2	Header 3x2	Top	place with daughtercard attached, maintain right angles	2		0.698	1.745	0.1396
Header 3x2		J3	Header 3x2	Top	place with daughtercard attached, maintain right angles					
Power Switch		J4	EG-1906	Top		1	0.71	6.4	16.35	0.654
Power Conn		J5	ZH-Conn	Bottom		1		0.89		0.1068
18 pin Conn		J6	HIROSE ST-18	Bottom		1	2.22	22.2	42.75	1.71
		J7	CapElectrode	Top	nothing to place					
Inductor	560uH	L1	CDRH74	Top		1	1.49	13.55	34.55	1.382
Inductor	2.2mH	L2	8RBS	Top		1	2.41	17.1	46.25	1.85
Resistor	100K	R1	0402	Bottom		2		0.74		0.148
Resistor	1ohm	R2	0402	Top		1		0.74		0.0888
Resistor	2ohm	R3	0402	Top		1		0.74		0.0888
Resistor	3.9ohm	R4	0402	Top		1		0.74		0.0888
Resistor	806k	R5	0402	Bottom		1		0.98		0.1176
Resistor	2.7M / 3.3M	R6	0402	Bottom	use 3.3M parts (R6 alt) after running out of R6	1		0.84		0.1008
Resistor	1M	R7	0402	Bottom		1				
Resistor	330k	R8	0402	Top		6		0.398		0.2388
Resistor	330k	R9	0402	Top						
Resistor	330k	R10	0402	Top						
Resistor	330k	R11	0402	Top						
Resistor	330k	R12	0402	Bottom						
Resistor	330k	R13	0402	Bottom						
Resistor	499k	R14	0402	Top		6		0.398		0.2388
Resistor	499k	R15	0402	Top						
Resistor	499k	R16	0402	Top						
Resistor	499k	R17	0402	Top						

Resistor	499k	R18	0402	Bottom						
Resistor	499k	R19	0402	Bottom						
Resistor	0 ohm	R20	0402	Top	don't place					
Resistor	0 ohm	R21	0402	Bottom		1		0.74		0.0888
Resistor	169k	R22	0402	Bottom		1		0.74		0.0888
Resistor	10k	R23	0402	Top		4		0.74		0.296
Resistor	10k	R24	0402	Bottom						
Resistor	10k	R25	0402	Bottom						
Resistor	150 Ohm	R26	0402	Top		3		0.398		0.12736
Resistor	430 Ohm	R27	0402	Top		3		0.398		0.12736
Resistor	430 Ohm	R28	0402	Top						
Resistor	150 Ohm	R29	0402	Top						
Resistor	430 Ohm	R30	0402	Top						
Resistor	150 Ohm	R31	0402	Top						
Resistor	6k	R32	0402	Top		1		0.98		0.1176
Resistor	51k	R33	0402	Bottom		1		0.452		0.05424
Resistor	10k	R34	0402	Top						
Resistor	100k	R35	0402	Bottom						
Resistor	0 ohm	R36	0402	Top	don't place					
5V DC-DC conv	LTC1474	U1	MS8	Bottom		1	7.5	75	124.5	4.98
Rate Gyro	ADXRS300	U2	BGA32	Bottom	BGA	3	35	350	875	105
3.3V LDO	TC1073	U3	SOT-23A	Top		1	0.9	9	19.5	0.78
Quad Op Amp	TLV2474	U4	HTSSOP14	Top		2	2.36	23.6	47.25	3.78
Quad Op Amp	TLV2474	U5	HTSSOP14	Top						
MCU	MSP430F14x	U6	QFN64	Bottom		1	10.08	100.8	226.88	9.0752
Dual Op Amp	LT6221	U7	DFN8	Bottom		1	4.13	41.3	73.75	2.95
Crystal	8MHZ	X1	XC856CT	Bottom		1	3.3	29.7	75.9	3.036

Sensor Daughtercards (Two Per Node)

Description	Value	Designator	Footprint	Layer	Special Instructions	Order Quantity/Node	Price/1u	Price/10u	Price/25u	Total / Node
Capacitor	22nF	SC1	0402	Bottom						
Capacitor	22nF	SC2	0402	Bottom						
Capacitor	22nF	SC3	0402	Bottom						
Capacitor	0.1uF	SC4	0402	Bottom						
Capacitor	0.1uF	SC5	0402	Bottom						
Capacitor	0.27uF	SC6	0603	Bottom		4		2.6		1.04
Capacitor	0.27uF	SC7	0603	Top						
Capacitor	0.1uF	SC8	0402	Top						
Capacitor	47nF	SC9	0402	Bottom						
Capacitor	0.1uF	SC10	0402	Bottom						
Header 3x2		SJ1 = J2,J3		Bottom	affix header to J1 on daughtercards, becomes J2,J3 on main board					
Rate Gyro	ADXRS300	SU1	BGA32	Top	BGA					
Accelerometer	ADXL210	SU2	E-8	Top	square logo on package denotes pin 1, follow example	2	17.46		407.4	32.592

RF Daughtercard

Description	Value	Designator	Footprint	Layer	Special Instructions	Order Quantity/Node	Price/1u	Price/10u	Price/25u	Total / Node
Capacitor	22pF	RC1	0603	Top		2		0.3		0.06
Capacitor	22pF	RC2	0603	Top						
Capacitor	4.7pF	RC3	0603	Top		2		0.23		0.046
Capacitor	2.2nF	RC4	0603	Top		1		0.23		0.0276
Capacitor	1nF	RC5	0603	Top		1		0.5		0.06
Capacitor	10nF	RC6	0603	Top		1		0.27		0.0324
Capacitor	33nF	RC7	0603	Top		1		0.34		0.0408
Capacitor	1pF	RC8	0603	Top		2		0.34		0.068
Capacitor	1pF	RC9	0603	Top						
Capacitor	2.2pF	RC10	0603	Top		1		0.51		0.0612
Capacitor	4.7pF	RC11	0603	Top						
Antenna		RE1	Chip Antenna	Top	chip antenna not used - please solder wire to pad as in example					
Header 3x2		RJ1 = J1		Bottom	affix card to header J1 on main board					
Inductor	3.3nH	RL1	0603	Top		1	0.232	2.32		0.232
Inductor	10nH	RL2	0603	Top		1	0.232	2.32		0.232
Inductor	5.6nH	RL3	0603	Top		2	0.175	1.75		0.35
Inductor	5.6nH	RL4	0603	Top						
Resistor	1M	RR1	0603	Top		1		0.8		0.096
Resistor	22K	RR2	0603	Top		1		0.69		0.0828
Data Radio	nRF2401A	RU1	QFN24	Top		1	4	40		4
Crystal	16MHz	RX1	4025/3025	Top	use 3025 crystal packages after the larger 4025s run out	1	3.3			3.3
							TOTAL PER NODE:			$188.27

Appendix C

Firmware Code

Listing C.1: Node initialization source file.

```
/********************initNode.c****************************
Firmware for Sensemble Node - intended for MSP430F148 or F149
First written by Ryan Aylward 9/05, MIT Media Lab, Responsive Environments
Adapted from code by S. Daniel Lovell
Latest version 7/06

Initialization and configuration of the node MCU
***********************************************************/

#ifndef initNode
#include <msp430x14x.h>
#include "nRF2401.h"
#include "initNode.h"
#define initNode

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;       //make interrupts occur on low to high transitions
  P1IFG &= ~BIT4;        //clear the interrupt flag before enabling them
  P1IE  |= BIT4;       //enable interrupts on P1.4 for DR1 from nRF

  //RGB LED control pins are 2.0,2.1,2.3,2.4,2.5,2.6 (2.2 is only input)
  //However, unused led control pins must be temporarily set as inputs
  //Initialize output to dim led pins (2.1,2.4,2.5)
  //Initialize input to remaining pins (2.0,2.3,2.6)
  P2DIR |=0x32;
  P2OUT |= 0x32; //reset leds - logic high means off
  P4DIR |= 0xEF; //pin 4.4 is Heart monitor digital input (unused)
  P6SEL |= 0xFF; // Port 6 ADC option select
}

void CONFIG(void) {
  //MCU clock initialization

  unsigned int i;
  WDTCTL = WDTPW + WDTHOLD;                   // Stop WDT
  BCSCTL1 |= XTS + XT2OFF;                   // 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) {
  //setup SPI link to RF module
  //node baud rate must be between 400k and 200k with nordic RF module
  UCTL0 |= SWRST;
  UTCTL0 = CKPH+SSEL0+STC;                    // ACLK, 3-pin mode
  UCTL0 = CHAR+SYNC+MM;                       // 8-bit SPI Master **SWRST disabled
       **
  //UBR00 = 0x20;                             //CLK0 = ACLK/32 (250kbps)
  //UBR00 = 0x10;                             //CLK0 = ACLK/16 (500kbps)
  UBR00 = 0x14;                               //CLK0 = ACLK/20 (400kbps)
  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;
}

void CONFIG_SPI1(void) {
  //Setup spare SPI port
  //Configuration has not been tested with external device
  UCTL1 |= SWRST;

  UTCTL1 = CKPL+SSEL0+STC;          // ACLK, 3-pin mode
  UCTL1 = CHAR+SYNC+MM;             // 8-bit SPI Master **SWRST disabled**
  UBR01 = 0x10;                     // CLK1 = ACLK/16 (500kbps)
  UBR11 = 0x00;                     //
  UMCTL1 = 0x00;                    // no modulation
  P5SEL |= 0x0F;                    // P5.0-5.4 SPI option select
  P3SEL &= ~0xC0;                   // Disable UART1
  ME2 = USPIE1;                     // Enable USART1 SPI mode

  UCTL1 &= ~SWRST;
}

void CONFIG_TIMER_A_INT(int interval, int sub_interval,char startNow) {
  //Configure timer for communications cycle

  TACTL = TASSEL0 + TACLR + ID1 + ID0;  // ACLK/8, clear TAR
  CCTL0 = CCIE;                         // CCR0 interrupt enabled
  CCTL1 = CCIE;                         // CCR1 interrupt enabled
  CCR0 = interval;                      //interval in us
  CCR1 = sub_interval;  //sub_interval must be smaller than interval
  if(startNow) {
    TACTL |= MC0;                       // Start Timer_a in upmode
  }
}

void CONFIG_TIMER_B0_INT(int interval, char startNow) {
  //Configure timer for capacitive sampling cycle

  TBCTL = TBSSEL0 + TBCLR + ID1 + ID0;  // ACLK/8, clear TAR, 16 bit counter
  TBCCTL0 = CCIE;                       // TBCCR0 interrupt enabled
  TBCCR0 = interval;                    //interval in us
  if(startNow) {
    TBCTL |= MC0;                       // Start Timer_B
  }
}

void CONFIG_TIMER_B1_INT(int interval, char startNow) {

  TBCTL = TBSSEL0 + TBCLR + ID1 + ID0;  // ACLK/8, clear TAR, 16 bit counter
  TBCCTL1 = CCIE;                       //TBCCR1 interrupt enabled
  TBCCR0 = interval;
  TBCCR1 = interval;                    //interval in us
  if(startNow) {
    TBCTL |= MC0;                       // Start Timer_B
    }
  }

  void CONFIG_ADC_CAPRX(void) {
    //Configure ADC for capacitive sampling

    ADC12CTL0 &= ~ENC; //reset enable conversions - conversions stop at the end
          of sequence
    ADC12CTL0 = ADC12ON;
    ADC12CTL1 &= ~0x20;  //reset SHP, in case it was previously set
    ADC12CTL1 = ADC12SSEL0 + CONSEQ0 + CSTARTADD_8; //trigger sample with
          ADC12SC, Use ACLK,
                                                   //sequence channels, start
                                                         at mem8
    ADC12IE &= ~0x40;     //disable interrupt on mem6, in case it was previously
          set
    ADC12MCTL6 = 0x00;
    ADC12MCTL8 = INCH_2; //stores Cap_RX to mem8 and mem9 alternating
    ADC12MCTL9 = INCH_2 + EOS;
    ADC12CTL0 |= ENC;    //set enable conversions
  }

  void CONFIG_ADC(void){
    //Configure ADC for standard sampling

    ADC12CTL0 &= ~ENC; //reset enable conversions - conversions stop at the end
          of sequence
    ADC12CTL0 = ADC12ON+SHT0_3+MSC;             // ADC12ON, SHT 32 cycles
    ADC12CTL1 = SHP + ADC12SSEL0 + CONSEQ0;  // Use sampling timer = ACLK,
                                            //single sequence, start at mem0
    ADC12IE = 0x40;                         //enable interrupt on mem6

    ADC12MCTL0 = INCH_0;                    // ref+=AVcc, channel = A0, analog1
    ADC12MCTL1 = INCH_1;                    // ref+=AVcc, channel = A1, analog2
    ADC12MCTL2 = INCH_3;                    // ref+=AVcc, channel = A3, analog4
    ADC12MCTL3 = INCH_4;                    // ref+=AVcc, channel = A4, analog5
    ADC12MCTL4 = INCH_5;                    // ref+=AVcc, channel = A5, analog6
    ADC12MCTL5 = INCH_6;                    // ref+=AVcc, channel = A6, analog7
    ADC12MCTL6 = INCH_7 + EOS;              // ref+=AVcc, channel = A7, analog8

    ADC12MCTL9 = 0x00;
    ADC12CTL0 |= ENC;                       // Enable conversions
  }

  void CONFIG_FLASH(void){
    //Configure flash

    //set clock to ACLK/20 (400kHz)
    FCTL2 = FWKEY + FN4 + FN1 + FN0;
  }

  #endif
```

Listing C.2: Node radio source file.

```
/*************nRF2401.c***************************
Firmware for Sensemble Node - intended for MSP430F148 or F149
First written by Ryan Aylward 9/05, MIT Media Lab, Responsive Environments
Adapted from code by S. Daniel Lovell
Latest version 7/06

Control and communication for the nRF2401 radio module
**************************************************/

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

void nRF_SetTXRXMode(void) {
  int i;
//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 (counter < length) {
    TXBUF0 = data[counter];          //fill tx buffer, reset UTXIFG0
    while (!(IFG1 & UTXIFG0));       // USART0 TX buffer ready?
    temp = RXBUF0;                   //reset URXIFG0
    counter++;
  }
  while(!(IFG1 & URXIFG0));          //transmission completed?
  temp = RXBUF0;                     //reset URXIFG0
}

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
  //NOT OPTIMIZED FOR SPEED
  char temp;
  int counter = 0;
    temp = RXBUF0;                      //reset URXIFG0
  while (counter < length) {
    while ((IFG1 & UTXIFG0) == 0);      // USART0 TX buffer ready?
    TXBUF0 = data;                      // fill tx buffer, reset UTXIFG0
    while ((IFG1 & URXIFG0) == 0);      // transmission completed?
    temp = RXBUF0;                      //reset URXIFG0
    counter++;
  }
}

void nRF_SHOCKBURST_BLOCKING_WRITE(unsigned char * data, int dataLength,
     unsigned char * addr,int addrLength) {
  int i;

  //TX WILL BE SENT ONLY WHEN nRF RETURNS TO STANDBY MODE
  //TX MODE MUST BE SET BEFORE CALLING BLOCKING WRITE

  //remember, more than 5us delay required between CE high and beginning of
       data

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

void nRF_SHOCKBURST_BLOCKING_WRITE_REPEATED(unsigned char data, int
     dataLength,unsigned char * addr,int addrLength) {
  int i;

  //TX WILL BE SENT ONLY WHEN nRF RETURNS TO STANDBY MODE
  //TX MODE MUST BE SET BEFORE CALLING BLOCKING WRITE

  //remember, 5us delay required between CE high and beginning of data

  //write dest address
  nRF_BLOCKING_WRITE(addr,addrLength);
  //write data (write_repeated is not optimized for speed)
  nRF_BLOCKING_WRITE_REPEATED(data,dataLength);
}

void nRF_BLOCKING_READ(unsigned char * data, int * data_length) {
  int temp;

  *data_length = 0;
  TXBUF0 = 0xFF;                        //write in order to receive, reset
       UTXIFG0
  //write ones so that MSP sources the current through R36 on wasted power
       cycles
  //(as opposed to the nRF sourcing current), see layout
  while (!(IFG1 & UTXIFG0));            // USART0 TX buffer ready?
  //temp = RXBUF0;                      //reset URXIFG0
  while(P1IN & 0x10) {
    TXBUF0 = 0xFF;                      //write in order to receive, reset
         UTXIFG0
    while (!(IFG1 & UTXIFG0));          // RX Complete?
    temp = RXBUF0;                      //reset URXIFG0
    data[(*data_length)++] = RXBUF0;  //store data, reset URXIFG0
  }
  while (!(IFG1 & URXIFG0));//make sure last (garbage) byte has been clocked
       out
  temp = RXBUF0;                        //reset URXIFG0
}

void nRF_READ_FROM_EXPECTED_PACKET(unsigned char * data, unsigned char length
     ){
  char idx=0;
  int temp;
  /***READS length OF VALID DATA FROM A RX PACKET - ***
   ***IF YOU CAN INSURE THAT length <= PACKET LENGTH**/
  //this is better since there is no garbage byte as in nRF_BLOCKING_READ
  TXBUF0 = 0xFF;                        //write in order to receive, reset
       UTXIFG0
  //write ones so that MSP sources the current through R36 on wasted power
       cycles
  //(as opposed to the nRF sourcing current)
  while (!(IFG1 & UTXIFG0));            // USART0 TX buffer ready?
  while(idx < length - 1){
    TXBUF0 = 0xFF;                      //write in order to receive, reset
         UTXIFG0
    while (!(IFG1 & UTXIFG0));          // RX Complete?
    temp = RXBUF0;                      //reset URXIFG0
    data[idx++] = RXBUF0;               //store data, reset URXIFG0
  }
  while (!(IFG1 & URXIFG0));
  temp = RXBUF0;
  data[idx] = RXBUF0;
}

void nRF_CONFIG(unsigned char * data,int data_length) {
  int 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
  nRF_BLOCKING_WRITE(data,data_length);
  //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
```

Listing C.3: Node flash source file.

```
//flash.c
//Handle the on chip flash memory
//Configuration is handled in initNode
#include <msp430x14x.h>
#include "initNode.h"
#ifndef flash
#include "flash.h"
#define flash

char * FlashPtr = (char *)MAX_ADDR;

void init_flash_pointer(unsigned int addr){
//flash write pointer marches DOWN from MAX_ADDR during successive writes
//init_flash_poitner only resets pointer if given valid address
//assumes space is already erased
  if((addr<=MAX_ADDR && addr>MAX_CODE_SPACE)||(addr<=INFO_HIGH_ADDR && addr>=
       INFO_LOW_ADDR)){
    FlashPtr = (char *)addr;
    }
}

void erase_flash_segment(unsigned int addr){
//only erases if given a valid address

  char * addr_ptr;

  addr_ptr = (char *)addr;
  if((addr<=MAX_ADDR && addr>MAX_CODE_SPACE)||(addr<=INFO_HIGH_ADDR && addr>=
       INFO_LOW_ADDR)){
    FCTL1 = FWKEY + ERASE;
    FCTL3 = FWKEY;
    *addr_ptr = 0;
    FCTL3 = FWKEY + LOCK;
  }
}

unsigned char write_flash_word(unsigned int word,int * addr){
//assumes flash clock is set and location is already erased
//WDT and interrupts must be disabled outside this function
  //unlock
  FCTL3 = FWKEY;
  //enable write with no pre-erase
  FCTL1 = FWKEY+WRT;
  //write word — IS LITTLE ENDIAN
  *addr = word;
  //disable write
  FCTL1 = FWKEY;
  //lock
  FCTL3 = FWKEY + LOCK;

  return 1;
}

unsigned char write_flash_byte(unsigned char byte){
//assumes flash clock is set and location is already erased
//WDT and interrupts must be disabled outside this function

  if(FlashPtr<=(char*)MAX_ADDR && FlashPtr>(char*)MAX_CODE_SPACE){
    //unlock
    FCTL3 = FWKEY;
    //enable write with no pre-erase
    FCTL1 = FWKEY+WRT;
    //write byte and decrement address
    *FlashPtr-- = byte;
    //disable write
    FCTL1 = FWKEY;
    //lock
    FCTL3 = FWKEY + LOCK;
    return 9;
  }
  else{
    return 10;
  }
}

unsigned char write_flash_array(unsigned char * data,char length){
  char i;

  //assumes flash clock is set and location is already erased
  //WDT and interrupts must be disabled outside this function

  if(FlashPtr<=(char*)MAX_ADDR && (FlashPtr-length)>(char*)MAX_CODE_SPACE){
    //unlock
    FCTL3 = FWKEY;
    //enable write with no pre-erase
    FCTL1 = FWKEY+WRT;
    //write byte and decrement address
    for(i=0;i<length;i++){
      *FlashPtr-- = *(data+i);
    }
    //disable write
    FCTL1 = FWKEY;
    //lock
    FCTL3 = FWKEY + LOCK;
    return 9;
  }
  else{
    return 10;
  }
}

void read_flash_array(unsigned char * data,char length,unsigned int addr){
  char i;
  char * addr_ptr;

  addr_ptr = (char *)addr;
  if(addr<=MAX_ADDR && (addr-length)>MAX_CODE_SPACE){
    for(i=0;i<length;i++){
      data[i] = *(addr_ptr-i);
    }
  }
}
#endif
```

Listing C.4: Node peripherals source file.

```
/********peripherals.c*********************
Firmware for Sensemble Node - intended for MSP430F148 or F149
First written by Ryan Aylward 9/05, MIT Media Lab, Responsive Environments
Latest version 7/06

LED settings
******************************************/

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

void LedDriver(unsigned char red,unsigned char green,unsigned char blue){
  //dim led pins (2.1,2.4,2.5)
  //bright led pins (2.0,2.3,2.6)
  //logic high means OFF
  //passing a 4 keeps previous settings

  switch(red){
    case 0:
      P2OUT |= 0x03;
      //P2DIR &= ~0x03;
      P2DIR |= 0x03;
    break;
    case 1:
      P2DIR &= ~0x03;
      //P2OUT |= 0x03;
      P2OUT &= ~0x02;
      P2DIR |= 0x02;
    break;
    case 2:
      P2DIR &= ~0x03;
      //P2OUT |= 0x03;
      P2OUT &= ~0x01;
      P2DIR |= 0x01;
    break;
    case 3:
      P2OUT &= ~0x03;
      P2DIR |= 0x03;
    break;
    case 4:
    break;
    }
  switch(blue){
    case 0:
      P2OUT |= 0x60;
      //P2DIR &= ~0x60;
      P2DIR |= 0x60;
    break;
    case 1:
      P2DIR &= ~0x60;
      //P2OUT |= 0x60;
      P2OUT &= ~0x20;
      P2DIR |= 0x20;
    break;
    case 2:
      P2DIR &= ~0x60;
      //P2OUT |= 0x60;
      P2OUT &= ~0x40;
      P2DIR |= 0x40;
    break;
    case 3:
      P2OUT &= ~0x60;
      P2DIR |= 0x60;
    break;
    case 4:
    break;
    }
  switch(green){
    case 0:
      P2OUT |= 0x18;
      //P2DIR &= ~0x18;
      P2DIR |= 0x18;
    break;
    case 1:
      P2DIR &= ~0x18;
      //P2OUT |= 0x18;
      P2OUT &= ~0x10;
      P2DIR |= 0x10;
    break;
    case 2:
      P2DIR &= ~0x18;
      //P2OUT |= 0x18;
      P2OUT &= ~0x08;
      P2DIR |= 0x08;
    break;
    case 3:
      P2OUT &= ~0x18;
      P2DIR |= 0x18;
    break;
    case 4:
    break;
    }
}

#endif
```

Listing C.5: Node main source file.

```
/***************mainNode.c*******************************
Firmware for Sensemble Node - intended for MSP430F148 or F149
First written by Ryan Aylward 9/05, MIT Media Lab, Responsive Environments
Adapted from code by S. Daniel Lovell
Latest version 7/06
```

```
Main execution processes
*******************************************************/
#include <__cross_studio_io.h>
#include <msp430x14x.h>
#include <in430.h>  //has a _NOP()
#include <inmsp.h>  //has a __delay_cycles()
#include <math.h>
#include "nRF2401.h"
#include "peripherals.h"
#include "initNode.h"
#include "mainNode.h"


//ID should start from 1 and should not exceed MAX_NUM_NODES
const unsigned char ID = 2;

/**********NRF Configuration**************************
      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] = {RF_PACKET_BIT_LENGTH,
     RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
     xA1,0x6F,0x42};
const int TX_CONFIG_BYTES_LENGTH = 15;
unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x42};
const int TX_SHORT_CONFIG_BYTES_LENGTH = 1;
unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
     RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
     xA1,0x6F,0x43};
const int RX_CONFIG_BYTES_LENGTH = 15;
unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x43};
const int RX_SHORT_CONFIG_BYTES_LENGTH = 1;
unsigned char ADDR_BYTES[5] = {0xBB,0xBB,0xBB,0xBB,0xAA}; //basestation
     address
const int ADDR_BYTES_LENGTH = 5;

/************** Comm data ****************************/
//used to store the message received from the basestation
unsigned char RX_MESSAGE[RF_PACKET_BYTE_LENGTH];
int RX_MESSAGE_LENGTH = 0;
//data packet to transmit (data packed into 16 bytes)
//current packet structure:
//8 bit ID, 12 bit AccX,AccY,AccZ,GyrP,GyrR,GyrY,Cap,Cap,Cap,Unused/timestamp
unsigned char DATA_PACKET[RF_PACKET_BYTE_LENGTH];
unsigned char LEDS[3] = {0,0,0};

/************TDMA timing setup************************/
int TIMER_A1_DELAY;
int TIMER_A0_DELAY;
int FIRST_NODE_DELAY;
int CYCLE_OFFSET = 20; //extra time at beginning of TDMA cycle
const int CYCLE_PERIOD = 330; //length of TDMA slots in us
const int TIMER_A1_DURATION = 1180; //about how long you spend inside timer
      a1

/********** Capacitive sensing ********************/
#ifdef CAPSENS
const unsigned int CAP_LENGTH = 32; //Capacitive transmit length in cycles at
      90.9kHz
const unsigned int CAP_PWROF2 = 5; //Cap_Length nearest power of 2
//accumulator arrays must have at least length=CAP_LENGTH
signed int PAC_MAN0[50];
signed int PAC_MAN1[50];
unsigned int CapRX[MAX_CAP_RX_NODES]; //hold capacitive values before packing
const int BCYCLE = 1320; //can read data every 4 nodes in TDMA cycle
unsigned char bcounter = 0; //counts iterations of timer b
unsigned int btime; //stores the millisecond offset of timer b from timer a1
#endif


void main(void) {
  int i;
  char bytes;

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

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

  //compensate for some clock skew bugs - highly dependent on hardware
  //using 15 byte packets or increasing the TDMA slots by a few usec may help
  if(ID == 11){
    CYCLE_OFFSET = 24;
  }
  else if(ID==15){
    CYCLE_OFFSET = 29;
  }
  else if(ID==16){
    CYCLE_OFFSET = 24;
  }

  //initialize the interval to be timed
  //THIS DEPENDS ON RF PACKET SIZE AND TIME SPENT IN nRF_data_waiting()
  //large interval TIMER_A0_DELAY is between RX of broadcast and preparation
       for next broadcast
  //should be somewhat less than BROADCAST_INTERVAL
  //tradeoff is between power consumption and tolerance to node/basestation
       clock skew
  TIMER_A0_DELAY = BROADCAST_INTERVAL - 500;
  TIMER_A1_DELAY = CYCLE_OFFSET+(ID*CYCLE_PERIOD);
  FIRST_NODE_DELAY = CYCLE_OFFSET+CYCLE_PERIOD;
  //Here TIMER_A1_DELAY + time in timer A1 must be less than TIMER_A0_DELAY
  if(ID > 0 && (TIMER_A1_DELAY+TIMER_A1_DURATION)<TIMER_A0_DELAY){
    //max num nodes is (just barely) 25 with 16B packets
    CONFIG_TIMER_A_INT(TIMER_A0_DELAY,TIMER_A1_DELAY,0);
  }
  else{
    _EINT();
    //ERROR BLINK
    while(1){
      LedDriver(1,0,0);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      LedDriver(0,0,2);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      LedDriver(0,2,0);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
      for(i=0;i<30000;i++);
    }
  }

  //initialize Nordic Radio
  //give nRF time to go to PWR_UP mode before going to config and standby
  for(i=0;i<5000;i++) {}
  nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  for(i=0;i<100;i++) {} //necessary?
  nRF_SetTXRXMode();

  LedDriver(0,0,1); //dim blue led
  _EINT();
  while(1) {
    LPM0; //enter low power mode
  }
}


void sample_data(void) {

  unsigned int adc1,adc2;

  //about 50us to complete conversions
  ADC12CTL0 |= ADC12SC; // Start conversion
  while(!(ADC12IFG & 0x40));

  adc1 = ADC12MEM2;
  adc2 = ADC12MEM4;
  DATA_PACKET[1] = adc1>>4; //analog 4 - AccX high byte
  DATA_PACKET[2] = (adc1<<4)|(adc2>>8); //AccX low nibble and AccY high
       nibble
  DATA_PACKET[3] = adc2; //analog 6 - AccY low byte
  adc1 = ADC12MEM5;
  adc2 = ADC12MEM6;
  DATA_PACKET[4] = adc1>>4;  //analog 7 - AccZ high byte
  DATA_PACKET[5] = (adc1<<4)|(adc2>>8); //AccZ low nibble and GyrP high
       nibble
  DATA_PACKET[6] = adc2; //analog 8 - GyrP low byte
  adc1 = ADC12MEM0;
  adc2 = ADC12MEM3;
  DATA_PACKET[7] = adc1>>4; //analog 1 - GyrR high byte
  DATA_PACKET[8] = (adc1<<4)|(adc2>>8); //GyrR low nibble and GyrY high
       nibble
  DATA_PACKET[9] = adc2; //analog 5 - GyrY low byte
  //adc1 = ADC12MEM1;
  //DATA_PACKET[10] = adc1>>4; //analog2 - Aux high byte
  //DATA_PACKET[11] = adc1<<4; //Aux low nibble
}


void cap_tx_pulse(void){
   int i;
   int length;

   i=0;
   P4DIR |= 0x40; //CAP_TX
   length = (CAP_LENGTH<<1); //Number of toggles is twice the number of
        periods
  //toggle CAP_TX at 90.9kHz
  while(i<length){
      __delay_cycles(35); //90.9kHz
      P4OUT ^= 0x40;      //CAP_TX signal
      i++;
    }
  P4DIR &= ~0x40; //CAP_RX
}


void cap_rx_listen(void){
  int temp;

  P4DIR &= ~0x40; //CAP_RX

  //pre delay sampling to compensate for TX ring up
  for(temp=0;temp<220;temp++){}
  temp=0;

  //This method samples at about 363.6 kHz
  //It captures samples in quadrature (4 samples per cycle)
  //for a duration of CAP_LENGTH/2 cycles of the transmit pulse

  LedDriver(4,4,2);
  ADC12CTL0 |= ADC12SC;
  ADC12CTL0 &= ~ADC12SC;
  __delay_cycles(7);
  while(temp<CAP_LENGTH){
    ADC12CTL0 |= ADC12SC;
    ADC12CTL0 &= ~ADC12SC;
    PAC_MAN0[temp]=ADC12MEM8;
    __delay_cycles(6);
    ADC12CTL0 |= ADC12SC;
    ADC12CTL0 &= ~ADC12SC;
    PAC_MAN1[temp]=ADC12MEM9;
    temp++;
  }
  LedDriver(4,4,0);
}

void cap_rx_calculate(unsigned char source_idx){
  int temp;
  long signed int I=0;
  long signed int Q=0;
  long signed int diff;
  unsigned int val;
  float S;

  if (source_idx < MAX_CAP_RX_NODES){
    //Accumulate samples in quadrature
    diff = PAC_MAN0[0]-PAC_MAN0[1];
    I = diff*diff;
    diff = PAC_MAN1[0]-PAC_MAN1[1];
    Q = diff*diff;
    for(temp=2;temp<CAP_LENGTH-1;temp++){
      diff = PAC_MAN0[temp]-PAC_MAN0[temp+1];
      I += diff*diff;
      diff = PAC_MAN1[temp]-PAC_MAN1[temp+1];
      Q += diff*diff;
    }
    S = sqrtf((float)((I + Q)>>CAP_PWROF2));
    val = (unsigned int)ceilf(S);

    //Hard limit to 12 byte range
    if(val < 0x0FFF){
      CapRX[source_idx] = val;
    }
    else{
      CapRX[source_idx] = 0x0FFF;
    }
  }
}

void nRF_data_waiting(void) __interrupt[PORT1_VECTOR] {
  unsigned int i;

  _DINT();
  //clear the interrupt flag
  P1IFG &= ~BIT4;
  //Start timing the RF interval
  TACTL |= MC0;

  //nRF_BLOCKING_READ(RX_MESSAGE,&RX_MESSAGE_LENGTH);//read all the received
      data
  nRF_READ_FROM_EXPECTED_PACKET(RX_MESSAGE,8);//OR read a known length of
      data

  //Interpret received packet
  if(RX_MESSAGE[0] == 0xAF){
      //Sample Mode
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      //config leaves nRF in standby
      //sample IMU and pressure sensors
      sample_data();
      //set packet header back to ID in case it's coming out of idle mode
      DATA_PACKET[0]=ID;
  }
  else{
      //Idle Mode
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      //config leaves nRF in standby
      DATA_PACKET[0]=0xF5; //set packet header to "read done"
  }
  //Process special message
  if(RX_MESSAGE[3] == 0xFF || RX_MESSAGE[3] == ID){
    //we have a message
    if(RX_MESSAGE[4] == 0x01){
      LEDS[0]=RX_MESSAGE[5];
      LEDS[1]=RX_MESSAGE[6];
      LEDS[2]=RX_MESSAGE[7];
    }
  }
  //Reset LEDs
  LedDriver(LEDS[0],LEDS[1],LEDS[2]);
  _EINT();
}


void Timer_A1 (void) __interrupt[TIMERA1_VECTOR]{
  int i,flag;

  _DINT();
  flag = TAIV; //have to manually reset A1

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

  #ifndef CAPSENS
    for(i=0;i<950;i++) {}  //wait for TX to occur 195us + TOA
  #else
    bcounter = 0;
    btime = TAR;
    if(ID & 0x01){
      CONFIG_TIMER_B0_INT(BCYCLE,1);
    cap_tx_pulse(); //with 32 cycles, takes about 340us to execute
    }
    else{
      CONFIG_TIMER_B0_INT(BCYCLE-CYCLE_PERIOD,1);
    for(i=0;i<700;i++){} //compensate for missing cap tx time
    }
    for(i=0;i<240;i++){} //continue waiting for TX to occur
  #endif

  //prepare return to receive mode
  nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
  //config leaves nRF in standby
  _EINT();
}

void Timer_A0 (void) __interrupt[TIMERA0_VECTOR] {
  _DINT();
  TACTL &= ~MC0;  //turn off/reset timer A0
  //Return to receive mode
  nRF_SetTXRXMode();
  LedDriver(0,0,1);//Dim Blue LED
  _EINT();
}

void Timer_B0 (void) __interrupt[TIMERB0_VECTOR] {
  _DINT();

  /****Current Timer B allows capacitive sampling for the following Nodes:
      ID+BCYCLE/CYCLE_PERIOD, ID+2B/C, ID+3B/C for odd numbered nodes
      ID-CYCLE_PERIOD+BCYCLE/CYCLE_PERIOD, ID-C+2B/C, ID-C+3B/C for even
          nodes
  provided that the sampling & calculation can occur before the end of the
       cycle
  this means that nodes with high ID numbers will not send capacitive data
       yet.
  **************************************************/
  TBCTL &= ~MC0; //Turn off timer B
  CONFIG_TIMER_B0_INT(BCYCLE,1); //reset interval
  btime += BCYCLE; //BCYCLE microseconds since last timer b or timer b start
  //we spend about 1260 microseconds in timer b, allow for extra buffer time,
  //so must be 1400us until the timer A0 interrupt in order to fit another
       CAP receive
  //if there is not enough time, we pack up the samples taken so far.
  if(bcounter < MAX_CAP_RX_NODES && (btime+1400)<TIMER_A0_DELAY){
    CONFIG_ADC_CAPRX();
    cap_rx_listen(); //with 32 cycles takes about 360us to execute (only
         samples during 16 cycles)
    cap_rx_calculate(bcounter); //with 32 cycles takes about 850us to execute
    bcounter++;
    CONFIG_ADC();
  }
  else{
    TBCTL &= ~MC0; //Turn off timer B
    DATA_PACKET[10] = CapRX[0]>>4;
    DATA_PACKET[11] = (CapRX[0]<<4) | (CapRX[1]>>8);
    DATA_PACKET[12] = CapRX[1];
    DATA_PACKET[13] = CapRX[2]>>4;
    DATA_PACKET[14] = CapRX[2]<<4;
  }
   _EINT();
}
```

Listing C.6: Node main source file adapted for pitching study.

```
/******************mainRedSoxNode.c*****************
Firmware for sensemble inertial sensing nodes - high rate/range sports
     application
Pared down version of mainMultiNode.c, provides no provision for capacitive
     sensing
Geared towards logging data to flash at 1kHz

Written by Ryan Aylward 03/2006, MIT Media Lab
Adapted from mainNode.c
***************************************************/
#include <__cross_studio_io.h>
#include <msp430x14x.h>
#include <in430.h>        //has a _NOP()
#include <inmsp.h>        //has a __delay_cycles()
#include "nRF2401.h"
#include "peripherals.h"
#include "flash.h"
#include "initRedSoxNode.h"
#include "mainRedSoxNode.h"


//ID should start from 1 and should not exceed MAX_NUM_NODES
const unsigned char ID = 6;
/*
      nRF Configuration Bytes:
       Byte 0: 16 Bit payload length for RX channel 2
       Byte 1: 16 Bit payload length for RX channel 1
       Byte 3-5: Address for RX channel 2
       Byte 6-10: Address for RX channel 1
       Byte 11, upper 6 bits: Address width in # bits
       Byte 11, lower 2 bits: CRC settings
       Byte 12: Various settings including Two Channel Receive, RF power
            output
       Byte 13, upper 7 bits: Frequency channel selection
       Byte 13, lower 1 bit: RX?
 */
//unsigned char TX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
      xA1,0x6F,0x04};
//unsigned char TX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0x00,0x00,0xBB,0xBB,0xBB,0x00,0x00,0xBB,0xBB,0xBB,0
      x61,0x6F,0x42};
unsigned char TX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
      xA1,0x6F,0x42};
const int TX_CONFIG_BYTES_LENGTH = 15;
//unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x04};
unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x42};
const int TX_SHORT_CONFIG_BYTES_LENGTH = 1;
//unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
      xA1,0x6F,0x05};
//unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0x00,0x00,0xBB,0xBB,0xBB,0x00,0x00,0xBB,0xBB,0xBB,0
      x61,0x6F,0x43};
unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0
      xA1,0x6F,0x43};
const int RX_CONFIG_BYTES_LENGTH = 15;
//unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x05};
unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x43};
const int RX_SHORT_CONFIG_BYTES_LENGTH = 1;
unsigned char ADDR_BYTES[5] = {0xBB,0xBB,0xBB,0xBB,0xAA};//basestation
      address
const int ADDR_BYTES_LENGTH = 5;
//unsigned char ADDR_BYTES[3] = {0xBB,0xBB,0xAA};//basestation address
//const int ADDR_BYTES_LENGTH = 3;

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

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

//data to transmit later or send to flash
```

```
unsigned int FlashReadIdx = MAX_ADDR;
unsigned char flash_state = 0;
unsigned char flash_cycle = 0;
//unsigned int timestamps[650];
//unsigned char timestamps[650];
//unsigned int time_idx = 0;


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

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

  //initialize the interval to be timed - DEPENDS ON RF PACKET SIZE AND TIME
       SPENT IN DATA WAITING
  //large interval temp is between RX of broadcast and preparation for next
       broadcast
  //should be somewhat less than BROADCAST_INTERVAL
  //tradeoff between power consumption and tolerance to node/basestation
       clock skew
  //second arg of CONFIG_TIMER_A_INT must be less than temp
  temp = BROADCAST_INTERVAL - 0x0200; //maximum timer arg is 9487 - max num
       nodes should be 26
  if(ID > 0 && ID <= MAX_NUM_NODES){
    delay = 225+(ID*330); //225 and 330 are offset based on a minimum
         overhead time of 1095 usec
    CONFIG_TIMER_A_INT(temp,delay,0);
    }

  //give nRF time to go to PWR_UP mode before going to stand by
  for(i=0;i<5000;i++) {}
  nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  for(i=0;i<100;i++) {} //debug
  nRF_SetTXRXMode();

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

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

  //enable interrupts
  _EINT();


  while(1) {

  }
}


void sample_data(void) {
  unsigned int adc1,adc2;

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


void nRF_data_waiting(void) __interrupt[PORT1_VECTOR] {
  unsigned int i;

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

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

  if(flash_state & 0x08){
    //FLASH WRITING ACTIVE
    if(flash_state == 10){
    //flash just filled
      //stop timer B
      TBCTL &= ~MC0;
      //enable A1 interrupts with original timer configuration
      CCTL1 = CCIE;
      CCTL0 = CCIE;
      CCR1 = 225+(ID*330);
      CCR0 = BROADCAST_INTERVAL - 0x0200;
      flash_state = 2;
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      DATA_PACKET[0]=0xFF; //set packet header to "flash full"
      LedDriver(0,2,0);
    }
    else if(flash_state == 8){
    //first instance
      //Stop and reset Timer A
      TACTL &= ~MC0;
      TAR = 0;
      //configure timer B
    CONFIG_TIMER_B1_INT(1000,0);
    //sleep radio but prepare for next receive
    //nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
    //config leaves nRF in standby until A0 interrupt is called
    TBCTL |= MC0; //start timer B
    sample_data(); //100us to complete conversions with CAPSENSE defined
    flash_state = write_flash_array(DATA_PACKET+1,9); //write to flash
    LedDriver(1,1,1);
  }
  else{
    //sleep radio but prepare for next receive
    //nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
    //config leaves nRF in standby until A0 interrupt is called
    TBCTL &= ~MC0; //stop timer B
    if(TBR > 500){
      TBR = 0;
      TBCTL |= MC0; //reset timer B
      sample_data(); //100us to complete conversions with CAPSENSE defined
      flash_state = write_flash_array(DATA_PACKET+1,9); //write to flash
      }
    else{
      TBR = 0;
      TBCTL |= MC0; //reset timer B
      }
    //sleep radio but prepare for next receive
    //nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
    //config leaves nRF in standby until A0 interrupt is called
  }
}
else{
  nRF_SHOCKBURST_BLOCKING_READ(RX_MESSAGE,&RX_MESSAGE_LENGTH);//read the
      received data
  if(RX_MESSAGE[0] == 0xAF){
    // Standard Low Latency Mode
    //get ready for next TX
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
    //config leaves nRF in standby
    DATA_PACKET[0]=ID; //set packet header to node ID
    //sample IMU and pressure sensors
    sample_data(); //100us to complete conversions with capacitive sensing
        on
    LedDriver(2,2,0);
    //Process Message
    if(RX_MESSAGE[1] == 0xFF || RX_MESSAGE[1] == ID){
      //we have a message
      if(RX_MESSAGE[2] == 0x01){
        LedDriver(RX_MESSAGE[3],RX_MESSAGE[4],RX_MESSAGE[5]);
      }
    }
  }
  else if(RX_MESSAGE[0] == 0xBE){
  //Enter flash writing mode
    if(flash_state & 0x02){
      //flash either full or previously in a read mode
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      DATA_PACKET[0] = 0xFF; //set packet header to "flash full"
      LedDriver(0,2,0);
    }
    else if(flash_state & 0x01){
      //flash empty
      if(RX_MESSAGE[1] == 0xFF){
        flash_state = 8; //enter write mode
      }
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      DATA_PACKET[0] = 0xF0; //set packet header to "flash empty"
      LedDriver(0,2,0);
    }
    else{
      //unknown flash state - erase flash, takes 1.4 seconds
      LedDriver(0,0,2);
      //Stop and reset Timer A
      TACTL &= ~MC0;
      TAR = 0;
      i = MAX_ADDR;
      while(i>MAX_CODE_SPACE){
        erase_flash_segment(i);
        i -= SEG_SIZE;
      }
      init_flash_pointer(MAX_ADDR);
      flash_cycle = 0;
      flash_state = 1; //flash now empty
      //listen for next RF broadcast to re-synch
      nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
      nRF_SetTXRXMode();
      LedDriver(0,2,0);
    }
  }
  else if(RX_MESSAGE[0] == 0xCD){
  // Flash Read Mode
    if(flash_state == 2){
      //flash is full, ready
      //get ready for next TX
      LedDriver(2,1,0);
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      if(FlashReadIdx<=MAX_ADDR && (FlashReadIdx-9)>MAX_CODE_SPACE){
        //read data from flash
        DATA_PACKET[0]=ID; //set packet header to node ID
        read_flash_array(DATA_PACKET+1,9,FlashReadIdx);
        FlashReadIdx -= 9;
      }
      else{
      //data segment of flash has been read from top to bottom
        DATA_PACKET[0]=0xF5; //set packet header to "read done"
        flash_state = 0;
        FlashReadIdx = MAX_ADDR; //reset read index
        LedDriver(0,2,0);
      }
    }
    else{
      //unknown or empty flash state - nothing to read
      //get ready for next TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      DATA_PACKET[0]=0xF5; //set packet header to "read done"
      LedDriver(0,2,0);
    }
  }
  else{
    //Node Idle Mode
    //get ready for next TX
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
```

```
      DATA_PACKET[0]=0xF5; //set packet header to "read done"
      LedDriver(0,0,0);
      //Process Message
      if(RX_MESSAGE[1] == 0xFF || RX_MESSAGE[1] == ID){
        //we have a message
        if(RX_MESSAGE[2] == 0x01){
          LedDriver(RX_MESSAGE[3],RX_MESSAGE[4],RX_MESSAGE[5]);
        }
      }
    }
  }
  _EINT();
}


void Timer_A1 (void) __interrupt[TIMERA1_VECTOR]{
  int i,flag;

  _DINT();
  flag = TAIV; //have to manually reset A1
  //set TX mode
  nRF_SetTXRXMode();
  //write packet to nRF buffer
  nRF_SHOCKBURST_BLOCKING_WRITE(DATA_PACKET,RF_PACKET_BYTE_LENGTH,ADDR_BYTES,
      ADDR_BYTES_LENGTH);
  //take CE low to start TX, then enter standby
  nRF_SetStdByMode();
  //prepare return to receive mode
  for(i=0;i<950;i++) {}//wait for TX to occur 195us + TOA
  nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
  //config leaves nRF in standby
  _EINT();
}

void Timer_A0 (void) __interrupt[TIMERA0_VECTOR] {
  _DINT();
  //turn off/reset timer A0
  TACTL &= ~MC0;
  //Return to receive mode
  nRF_SetTXRXMode();
  LedDriver(1,0,0);//Red LED on
  _EINT();
}

void Timer_B0 (void) __interrupt[TIMERB0_VECTOR] {

  _DINT();
  TBCTL &= ~MC0; //Turn off timer B
  _EINT();
}

void Timer_B1 (void) __interrupt[TIMERB1_VECTOR] {
  int i;

  _DINT();
  i = TBIV; //have to manually reset B1
  sample_data();
  flash_state = write_flash_array(DATA_PACKET+1,9);
  if(flash_state == 10){
    //flash just filled
      //stop timer B
      TBCTL &= ~MC0;
      nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
      nRF_SetTXRXMode();
      flash_state = 2;
      LedDriver(0,2,0);
  }
  _EINT();
}
```

Listing C.7: Node initialization header file.

```
//initNode.h

//Define if using capacitive sensor - note increased code space requirements
     in flash.h
#define CAPSENS

#define MAX_NUM_NODES 25 //was able to get 26 with simpler basestation
     firmware and no check for "idle" and "sample" states
#define RF_PACKET_BYTE_LENGTH  16 //note that currently one byte is wasted
     and better performance could be guaranteed with 15 bytes
#define RF_PACKET_BIT_LENGTH 0x80
#define MAX_ADDR_BYTES_LENGTH 5
#define MAX_CAP_RX_NODES 3

//Set broadcast interval
//interval based on 1MHz clock timer - 0x2710 cycles for 100Hz, 0x4E20 cycle
     for 50Hz
//this is the frequency of basestation broadcasts
#define BROADCAST_INTERVAL 0x2710
//#define BROADCAST_INTERVAL 0x4E20

void CONFIG_IO(void);
void CONFIG(void);
void CONFIG_SPI0(void);
void CONFIG_SPI1(void);
void CONFIG_TIMER_A_INT(int,int,char);
void CONFIG_TIMER_B0_INT(int,char);
void CONFIG_TIMER_B1_INT(int,char);
void CONFIG_ADC_CAPRX(void);
void CONFIG_ADC(void);
void CONFIG_FLASH(void);
```

Listing C.8: Node radio header file.

```
//nRF2401.h

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

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_BLOCKING_READ(unsigned char *, int *);
void nRF_READ_FROM_EXPECTED_PACKET(unsigned char *,unsigned char);
void nRF_CONFIG(unsigned char *,int);
void nRF_RESET(void);
```

Listing C.9: Node flash header file.

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

#define MAX_ADDR 0xFDFF //interrupt memory segment begins at 0xFE00
#define SEG_SIZE 0x0200 //512 byte segments
#define MAX_CODE_SPACE 0x1BFF //msp430149 WITHOUT capacitive sensing (57856 B
     )
//#define MAX_CODE_SPACE 0x21FF //for msp430f149 only (56320 B of free flash
     segments)
//#define MAX_CODE_SPACE 0x50FF //for msp430148 only  (44544 B of free flash
     segments)

//information memory provides another 256 B of free flash
#define INFO_HIGH_ADDR 0x10FF
#define INFO_LOW_ADDR 0x1000
```

```
void init_flash_pointer(unsigned int);
void erase_flash_segment(unsigned int);
unsigned char write_flash_word(unsigned int , int *);
unsigned char write_flash_byte(unsigned char);
unsigned char write_flash_array(unsigned char *,char);
void read_flash_array(unsigned char *,char,unsigned int);
```

Listing C.10: Node peripherals header file.

```
//peripherals.h
void LedDriver(unsigned char, unsigned char, unsigned char);
```

Listing C.11: Node main header file.

```
//mainNode.h
void cap_tx_pulse(void); //toggle CAP_TX at 90.9kHz
void cap_rx_listen(void); //sample quadrature on CAP_RX
void cap_rx_calculate(unsigned char); //accumulate cap samples, perform
     calculation
void sample_data(void); //sample IMU and resistive sensors, load current
     capacitive data into packet
//void buffer_data(unsigned char *,char): //copy data to ring buffer
```

Listing C.12: Basestation radio source file.

```
//————————————————
// RF_Base.c
//————————————————
// MIT MEDIA LAB, RESPONSIVE ENVIRONMENTS GROUP
// Ryan Aylward
// Latest version: July 2006

//RF Routines for Sensemble Basestation


//————————————————
// Includes
//————————————————
#include <c8051f320.h>
#include <intrins.h>
#ifndef RF_Base
#include "RF_Base.h"
#define RF_Base


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
  P1 &= ~nRF_CS;
  P1 |= nRF_CE | nRF_PWRUP;

   //5us delay required between CE high and beginning of data
}

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
  P1 &= ~nRF_CE;
  P1 |= 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
  P1 &= ~(nRF_CE | nRF_CS) ;
  P1 |= 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
  P1 &= ~nRF_PWRUP;
}

void nRF_SPI_WRITE(unsigned char *buffer,int length){
//configuration of PWR_UP, CE, CS pins assumed
//configuration of port 0 must have mosi on 0.2!
  int counter = 0;

  while (counter < length) {
    SPIDAT = buffer[counter]; //fill tx buffer
    while (TXBMT == 0);        //buffer ready?
  SPIF = 0;              //reset interrupt flag
    counter++;
  }
  //assume one more byte is on its way out
  while(SPIF == 0);        //TX complete?
  SPIF = 0;             //reset interrupt flag
}


void nRF_SPI_READ(unsigned char *buffer,char *data_length){
//configuration of PWR_UP, CE, CS pins assumed
//configuration of port 0 must have mosi on 0.3 and 0.2 set to digital in!
//tries to read one more byte than nRF has in its buffer
  *data_length = 0;
  SPIDAT = 0xAA;        //write in order to receive
  while (TXBMT == 0);   // RX initiated?
  SPIF = 0;
  while(P0 & 0x40) {
    SPIDAT = 0xAA;        //write in order to receive
    while (SPIF == 0);  // RX complete and next RX initiated?
  SPIF = 0;          //reset interrupt flag
    buffer[(*data_length)++] = SPIDAT; //store data
  }
  while(SPIF == 0); //make sure last (garbage) byte has been clocked out
  SPIF = 0;
}

void nRF_SPI_READ_EXPECTED_PACKET(unsigned char *buffer){
//configuration of PWR_UP, CE, CS pins assumed
//configuration of port 0 must have mosi on 0.3 and 0.2 set to digital in!
//reads only the expected packet length - not robust but faster
  int idx=0;

  SPIDAT = 0xAA;          //write in order to receive
  while (TXBMT == 0);       // RX initiated?
  SPIF = 0;
  while(idx < RF_PACKET_BYTE_LENGTH - 1){
    SPIDAT = 0xAA;          //write in order to receive
    while (SPIF == 0);     // RX complete and next RX initiated?
  SPIF = 0;          //reset interrupt flag
  buffer[idx] = SPIDAT;    //store data
  idx++;
  }
  while(SPIF == 0);
  SPIF = 0;
  buffer[idx] = SPIDAT;
}

void nRF_SPI_WRITE_REPEATED_BYTE(unsigned char byte,int length){
//configuration of PWR_UP, CE, CS pins assumed
//configuration of port 0 must have mosi on 0.2!
  int counter = 0;

  while (counter < length) {
    SPIDAT = byte;        //fill tx buffer
    while (TXBMT == 0);       //buffer ready?
  SPIF = 0;           //reset interrupt flag
    counter++;
  }
  //assume one more byte is on its way out
  while(SPIF == 0);      //TX complete?
  SPIF = 0;          //reset interrupt flag
}


void nRF_TX(unsigned char * buffer, int dataLength,unsigned char * addr,int
     addrLength) {

  //TX WILL BE SENT ONLY WHEN nRF RETURNS TO STANDBY MODE
  //TX MODE MUST BE SET BEFORE CALLING WRITE
  //PROPER PORT AND SPI SETTINGS ASSUMED

  //write dest address
  nRF_SPI_WRITE(addr,addrLength);
  //write data
  nRF_SPI_WRITE(buffer,dataLength);
}


void nRF_TX_REPEATED_BYTE(unsigned char byte, int dataLength,unsigned char *
     addr,int addrLength) {

  //TX WILL BE SENT ONLY WHEN nRF RETURNS TO STANDBY MODE
  //TX MODE MUST BE SET BEFORE CALLING BLOCKING WRITE
  //PROPER PORT AND SPI SETTINGS ASSUMED

  //write dest address
  nRF_SPI_WRITE(addr,addrLength);
  //write data
  nRF_SPI_WRITE_REPEATED_BYTE(byte,dataLength);
}


void nRF_CONFIG(unsigned char * buffer,int data_length) {
/*
        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
  nRF_SPI_WRITE(buffer,data_length);
  nRF_SetTXRXMode();
}

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
```

Listing C.13: Basestation main source file.

```
//-----------------------------------------------------------------------------
// MAIN_BASE.c
//-----------------------------------------------------------------------------
// MIT MEDIA LAB, RESPONSIVE ENVIRONMENTS GROUP
// Ryan Aylward
// Latest version: July 2006

//Main execution firmware for Sensemble basestation

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f320.h>
#include <intrins.h>
#include "Main_Base.h"
#include "RF_Base.h"
#include "USB_API.h"


//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------

sbit LEDB = P2^0; //LED=0 means ON
sbit LEDG = P2^1;
sbit LEDR = P1^7;

//output to generate synch signal sent through external optocoupler - for
     RedSox
sbit SYNCHOUT = P2^2;
//inverse logic at pin is inverted again by external optocoupler
//external system at opto output sees logic low for SYNCHOUT = 0

// Last packet received from host
BYTE xdata Out_Packet[8] = {0,0,0,0,0,0,0,0};
// Data space for next packet to send to host
unsigned char xdata In_Buffer[IN_BUFFER_LENGTH];
// Packet for Nodes
unsigned char xdata Broadcast_Packet[RF_PACKET_BYTE_LENGTH];
// Packet sent to host in advent of no data
BYTE xdata Dummy_Packet[10] = {0,0,0,0,0,0,0,0,0,0};

/******HOST MESSAGES**********(limited by size of Out_Packet)*****
If the first byte == 0x00 then the host has no message
If the first byte == 0x0A then the message is evaluated locally
In these cases the current broadcast packet to the nodes remains unchanged
Otherwise, the message is a new broadcast packet and overwrites the current
     packet
```

```
Currently the message is only sent to the nodes once before being reset
***************************************************************/

/*****BROADCAST PACKET FORMAT****(limited by RF_PACKET_BYTE_LENGTH)***
Mode            Timestamp   Target ID   Message Code and Data ...
Sample & Respond 0xAF  2bytes  All nodes 0xFF     set LEDs: 0x01 Red Green Blue
                 No message 0x00
                 One node (node ID)

Writing Flash 0xBE       Node responds in TDMA slot when ready,
                 Send 0xFF as target ID when you want to start writing
                 Node responds in TDMA slot when full

Reading Flash 0xCD       Nodes send back RF packets with usual TDMA scheme

Idle       0xFF
*********************************************************************/

/**************DATA PACKET FORMAT FOR HOST**********************
Header | NodeHeader Timestamp Length ID Payload | NodeHeader Timestamp ...
  1        1            1        1    1    x        1           1
***************************************************************/

int packet_idx = 0;        //next index to fill in USB IN Buffer
unsigned int timestamp_counter = 0;   //timecode of current broadcast
const char node_header = 0xAC;
char USBFlag = 0;
char state = DEFAULT;
char flash_flag = 0;
char framenum;
char synchshift;

//RF configuration
/*
        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] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xAA,0xBB,0xBB,0xBB,0xBB,0xAA,0
      xA1,0x6F,0x42};
const int TX_CONFIG_BYTES_LENGTH = 15;
const unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x42};
unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,
      RF_PACKET_BIT_LENGTH,0xBB,0xBB,0xBB,0xBB,0xAA,0xBB,0xBB,0xBB,0xBB,0xAA,0
      xA1,0x6F,0x43};
const int RX_CONFIG_BYTES_LENGTH = 15;
const unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x43};
const unsigned char ADDR_BYTES[MAX_ADDR_BYTES_LENGTH] = {0xBB,0xBB,0xBB,0xBB
      ,0xBB};//node address
const int ADDR_BYTES_LENGTH = MAX_ADDR_BYTES_LENGTH;

//USB Configuration
code const UINT USB_VID = 0x10C4;
code const UINT USB_PID = 0xEA61;
code const BYTE USB_MfrStr[] = {0x1A,0x03,'S',0,'i',0,'l',0,'i',0,'c',0,'o'
      ,0,'n',0,' ',0,'L',0,'a',0,'b',0,'s',0};                        //
      Manufacturer String
code const BYTE USB_ProductStr[] = {0x10,0x03,'U',0,'S',0,'B',0,' ',0,'A',0,'
      P',0,'I',0}; // Product Desc. String
code const BYTE USB_SerialStr[] = {0x0A,0x03,'1',0,'2',0,'3',0,'4',0};
code const BYTE USB_MaxPower = 50;              // Max current = 100 mA (50 *
      2)
code const BYTE USB_PwAttributes = 0x80;       // Bus-powered, remote wakeup
      not supported
code const UINT USB_bcdDevice = 0x0100;        // Device release number 1.00

//-----------------------------------------------------------
// Main Routine
//-----------------------------------------------------------
void main(void)
{
  int i;

   PCA0MD &= ~0x40;          // Disable Watchdog timer
   Sysclk_Init();            // Initialize oscillator

   USB_Clock_Start();      //Init USB clock *before* calling USB_Init
   USB_Init(USB_VID,USB_PID,USB_MfrStr,USB_ProductStr,USB_SerialStr,
        USB_MaxPower,USB_PwAttributes,USB_bcdDevice);
   Port_Init();              // Initialize crossbar, GPIO, and external pin
        interrupts
   SPI_Init();          // Initialize SPI
   nRF_RESET();
   nRF_SetStdByMode();
   Timer_Init();             // Initialize timers

   //Put the global header into In_Buffer[0], increment packet_idx
   In_Buffer[packet_idx++] = 0x1F;
   //initialize broadcast packet to idle state
   Broadcast_Packet[0] = 0xFF;
   for(i=1;i<RF_PACKET_BYTE_LENGTH;i++){
      Broadcast_Packet[i] = 0;
    }

   //constant dummy packet indicates unanswered broadcast (no data), setup
        here:
   Dummy_Packet[0] = 0x1F;
   Dummy_Packet[1] = node_header;
   Dummy_Packet[3] = 6;

   //give nRF 3ms in standby before configuring
   for(i=0;i<8000;i++);
   nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);

   IE  = 0x8B;           // Enable timer0,1 interrupts and external interrupt 0
   USB_Int_Enable();        // Enable USB interrupts
   TR0 = 1;           // Turn on timer 0


   while (1)
   {
    }
  }

  //---------------------------------------------------------------
  //Interrupt Service Routines
  //---------------------------------------------------------------
void RF_data_waiting (void) interrupt 0
{

  EA = 0;
  LEDG = 0;

  //Move MOSI to dummy pin for SPI to Nordic hack! (see schematic)
    POMDOUT &= ~0x04;
    P0    |= 0x04;
    P0SKIP = 0xC4;

  //make sure a full packet will fit in the buffer with associated headers
    if(packet_idx+RF_PACKET_BYTE_LENGTH+3<= IN_BUFFER_LENGTH){
    //pack headers
      In_Buffer[packet_idx++]=node_header;
    In_Buffer[packet_idx++]=timestamp_counter; //we use only the lower byte
         of the timer
    In_Buffer[packet_idx++]=RF_PACKET_BYTE_LENGTH;
    //clock out and store packet (reads set length for speed)
      nRF_SPI_READ_EXPECTED_PACKET(&In_Buffer+packet_idx);
    //interpret node status - needed mainly for RedSox app state machine
    if(In_Buffer[packet_idx]<NODE_EMPTY){
      packet_idx += RF_PACKET_BYTE_LENGTH;
       }
    else{
       if(state == FLASH_WRITE){
        if(In_Buffer[packet_idx]==NODE_EMPTY){
            flash_flag = 1;
            synchshift = 0;
          }
        }
        else if(In_Buffer[packet_idx]==NODE_FULL){
            flash_flag = 0;
            state = DEFAULT;
            SYNCHOUT = 1;
            LEDB = 1;
          }
        }
      packet_idx -=3;
       }
    }

  //Move MOSI back to standard pin
    POMDOUT |= 0x04;
    P0    &= ~0x04;
    P0SKIP = 0xC0;


  LEDG = 1;
  EA = 1;
}

void Timer0_ISR (void) interrupt 1
{

  LEDR = 0;
  EA = 0;
  // reset timer 0
  TH0 = TIMER0_HIGH_BYTE;
  TL0 = TIMER0_LOW_BYTE;
  // set timer 1
  TH1 = TIMER1_HIGH_BYTE;
  TL1 = TIMER1_LOW_BYTE;
  TR1 = 1;

  //pack timestamp into broadcast
  Broadcast_Packet[1] = timestamp_counter>>8;
  Broadcast_Packet[2] = timestamp_counter;

   //broadcast to all nodes
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,1);
  //config leaves radio in TX mode
  nRF_TX(Broadcast_Packet,RF_PACKET_BYTE_LENGTH,ADDR_BYTES,ADDR_BYTES_LENGTH)
      ;
  //send tx and enter standby
  nRF_SetStdByMode();
  //MUST wait for TX to occur before entering RX mode

  //uncomment below for Red Sox app, otherwise it's too slow
  /*//if in flash mode, handle synchronization pulses for external system
  if(flash_flag){
    if(synchshift<8){
      SYNCHOUT=((framenum<<1)>>synchshift)&0x01;
      //LEDB=0;
      synchshift++;
    }
  }*/

  //reset any pending message from host - does reset node states, just
       special message
  Broadcast_Packet[3]=0;


  //clamp USB TX to rf cycle and send packets to the USB pipe only when host
       has requested data
  if(USBFlag == 1){
    if(packet_idx > 1){
    Block_Write(In_Buffer,packet_idx);
    }
    else{
    Dummy_Packet[2]=timestamp_counter;
    Block_Write(Dummy_Packet,10);
    }
    //packet index reset to 1, if you reset it to zero you will overwrite the
         header!
    packet_idx=1;
    USBFlag = 0;
  }

  timestamp_counter++;

  LEDR = 1;
  EA = 1;
}

void Timer1_ISR (void) interrupt 3
{
  //This allows radio to stay in standby until TDMA cycle begins
  //takes 64 us to execute
  TR1 = 0;
  LEDR = 0;
  //return to receive mode
    nRF_CONFIG(RX_SHORT_CONFIG_BYTES,1);
    //config leaves radio in RX mode
  LEDR = 1;
}

void USB_API_TEST_ISR(void) interrupt 16
{
  BYTE TEMP = 0;
  BYTE INTVAL;
  int i;

  LEDB = 0;
  IE  &= ~0x0B; //disable other interrupts
  INTVAL = Get_Interrupt_Source();
  if (INTVAL & RX_COMPLETE)
  {
    TEMP = Block_Read(Out_Packet,6);
    if(Out_Packet[0]){
      if(Out_Packet[0] == 0x0A){
        //evaluate local message for basestation here
        //will not be interpreted as a host request for data!
      }
      else{
        Broadcast_Packet[0]=Out_Packet[0];
        for(i=1;i<6;i++){
          Broadcast_Packet[i+2] = Out_Packet[i];
        }
        if(Out_Packet[0] == 0xBE && Out_Packet[1] == 0xFF){
          state = FLASH_WRITE;
          if(!flash_flag){
            framenum = Broadcast_Packet[4];
            //this line allows LED to be treated as in idle mode for non-
                flash nodes
            Broadcast_Packet[4] = 1;
          }
        }
        if(Out_Packet[0] != 0xFF){
          //THIS MEANS THE HOST IS READY FOR DATA
          USBFlag = 1;
        }
      }
    }
  }
  else if(INTVAL & DEVICE_CLOSE)
  {
    USBFlag = 0;
    Broadcast_Packet[0] = 0xFF;
    packet_idx=1;
    state=DEFAULT;
    flash_flag=0;
  }
  else if(INTVAL & DEV_SUSPEND){
    LEDB=1;
    LEDR=1;
    LEDG=1;
    nRF_SetPwrDownMode();
      USB_Suspend();
      nRF_RESET();
        nRF_SetStdByMode();
        for(i=0;i<8000;i++); //let nRF settle in stdby
        nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  }
  IE |= 0x0B; //enable other interrupts
  LEDB = 1;
}


//----------------------------------------------------
// Initialization Subroutines
//----------------------------------------------------

//---------------------------
// Sysclk_Init
//---------------------------
// SYSLCK Initialization
// - Initialize the system clock and USB clock
//
void Sysclk_Init(void)
{
  #ifdef _USB_LOW_SPEED_

      OSCICN |= 0x03;                          // Configure internal oscillator
           for
                                             // its maximum frequency and enable
                                             // missing clock detector

      CLKSEL  = SYS_INT_OSC;                 // Select System clock
      CLKSEL |= USB_INT_OSC_DIV_2;           // Select USB clock
  #else
      OSCICN |= 0x03;                        // Configure internal oscillator for
                                               // its maximum frequency and enable
                                               // missing clock detector

      CLKMUL  = 0x00;                        //reset clock multiplier, source is
           internal osc
      CLKMUL |= 0x80;                        // Enable clock multiplier
    Delay();                             // Delay for clock multiplier to begin
      CLKMUL |= 0xC0;                        // Initialize the clock multiplier
      while(!(CLKMUL & 0x20));               // Wait for multiplier to lock

    CLKSEL = SYS_4X_DIV_2;               // Select system clock
      CLKSEL |= USB_4X_CLOCK;                // Select USB clock
  #endif
}

//---------------------------
// Port_Init
//---------------------------
// Port Initialization
// - Configure the Crossbar and GPIO ports.
//
void Port_Init(void)
{
   //set high speed outputs to push-pull! data ready inputs are not push pull
   POMDIN  = 0xFF;               //Port 0 pins are digital I/O
  P0MDOUT = 0x0F;                          //Port 0 pins 0,1,2,3 are push/
      pull
  P0   |= 0xC0;               //Port 0 pins 6,7 are inputs
  P2MDIN  = 0xFF;               //Port 2 pins are digital I/O
  P2MDOUT = 0xA7;               //Port 2 pins 0,1,2,5,7 are push/pull
  P2     = 0x07;
  P1MDIN  = 0xFF;               //Port 1 is Digital I/O
  P1MDOUT = 0xFF;                          // Port 1 pins are push/pull
  P1      = 0x80;

  P1SKIP = 0xFF;              //crossbar skips port 1 and 2, 0.6,0.7
  P2SKIP = 0xFF;
  P0SKIP = 0xC0;

  IT01CF = 0xFE; // /INT0 on pin 0.6, /INT1 on 0.7, both active high


  XBR0      = 0x02;               // SPI enabled
  XBR1      = 0x40;                          // Enable Crossbar
}

//-----------------------
//SPI Init
//-----------------------
void SPI_Init(void){

  SPICFG |= 0x40; //set as master
  TXBSY = 0; //ensure 3 wire master mode
  SLVSEL = 0;
  SPICKR = 0x0B;  //1 Mbps
  SPIEN = 1;  //enable
}

//-----------------------
//Timer Init
//-----------------------

void Timer_Init(void){

  TCON = 0x05;  //stop timers 0,1, clear flags, make external ints edge
      sensitive
  TMOD = 0x11;  //timers 0,1 are 16bit counters
  CKCON = 0xF0; //timer 1, timer 0 use clk/12
  TH0 = TIMER0_HIGH_BYTE;  //set timer 0 to 100hz
  TL0 = TIMER0_LOW_BYTE;
  TH1 = TIMER1_HIGH_BYTE;
  TL1 = TIMER1_LOW_BYTE;
}


void Delay(void)
{
   int i;
   for(i=0;i<500;i++){}
}
```

Listing C.14: Basestation radio header file.

```
//RF_Base.h

#define nRF_CS     0x10 //pin 1.4
#define nRF_CE     0x02 //pin 1.1
#define nRF_PWRUP  0x01 //pin 1.0
#define nRF_CLK2   0x04 //pin 1.2
#define nRF_DR2    0x80 //pin 0.7
#define nRF_DOUT2  0x08 //pin 1.3
#define nRF_DR1    0x40 //pin 0.6
#define RF_PACKET_BYTE_LENGTH  16
#define RF_PACKET_BIT_LENGTH 0x80
#define MAX_ADDR_BYTES_LENGTH 5

void nRF_SetTXRXMode(void);
void nRF_SetConfigMode(void);
void nRF_SetStdByMode(void);
void nRF_SetPwrDownMode(void);
void nRF_SPI_WRITE(unsigned char *,int);
void nRF_SPI_READ(unsigned char *,char *);
void nRF_SPI_READ_EXPECTED_PACKET(unsigned char *);
void nRF_SPI_WRITE_REPEATED_BYTE(unsigned char, int);
void nRF_TX(unsigned char *, int,unsigned char *,int);
void nRF_TX_REPEATED_BYTE(unsigned char, int,unsigned char *,int);
void nRF_CONFIG(unsigned char *,int);
void nRF_SHORT_CONFIG_RX(void);
void nRF_SHORT_CONFIG_TX(void);
void nRF_RESET(void);
```

Listing C.15: Basestation main header file.

```
//Main_Base.h

#ifndef _MAINBASE_H_
#define _MAINBASE_H_

//#define _USB_LOW_SPEED_                         // uncomment for USB Low speed

#define SYSCLK                    12000000    // SYSCLK frequency in Hz

// USB clock selections (SFR CLKSEL)
#define USB_4X_CLOCK              0x00        // Select 4x clock multiplier,
       for USB Full Speed
#define USB_INT_OSC_DIV_2         0x10        // See Data Sheet section 13.
       Oscillators
#define USB_EXT_OSC               0x20
#define USB_EXT_OSC_DIV_2         0x30
#define USB_EXT_OSC_DIV_3         0x40
#define USB_EXT_OSC_DIV_4         0x50

// System clock selections (SFR CLKSEL)
#define SYS_INT_OSC               0x00        // Select to use internal
       oscillator
#define SYS_EXT_OSC               0x01        // Select to use an external
       oscillator
#define SYS_4X_DIV_2              0x02
```

```
// BYTE type definition
#ifndef _BYTE_DEF_
#define _BYTE_DEF_
typedef unsigned char BYTE;
#endif   /* _BYTE_DEF_ */

// WORD type definition, for KEIL Compiler
#ifndef _WORD_DEF_                          // Compiler Specific, written
      for Little Endian
#define _WORD_DEF_
typedef union {unsigned int i; unsigned char c[2];} WORD;
#define LSB 1                               // All words sent to and
      received from the host are
#define MSB 0                               // little endian, this is
      switched by software when
                                            // neccessary.  These sections
                                                  of code have been marked
                        // with "Compiler Specific" as above for easier
                              modification
#endif   /* _WORD_DEF_ */

// Define Endpoint Packet Sizes
#ifdef _USB_LOW_SPEED_
#define  EP0_PACKET_SIZE         0x08        // This value can be 8,16,32,64
      depending on device speed, see USB spec
#else
#define  EP0_PACKET_SIZE         0x40
#endif /* _USB_LOW_SPEED_ */


#define  MAX_EP1_PACKET_SIZE          0x0400      // Can range 0 - 1024
      depending on data and transfer type
#define  MAX_EP1_PACKET_SIZE_LE       0x0004      // IMPORTANT- this should be
       Little-Endian version of EP1_PACKET_SIZE
```

```
#define IN_BUFFER_LENGTH 952 //ensures two complete samples from 25 nodes can
       be taken between host reads

// TIMER INTERVALS FOR RF PROTOCOL
// 100 Hz broadcast rate 0xB1DA
#define TIMER0_LOW_BYTE   0xDA
#define TIMER0_HIGH_BYTE  0xB1
// Timer1 interval depends on time of RF broadcast + write to USB IN_BUFFER
// This interval will determine how soon nodes can respond with data
// In most cases it will take longer for the node to pack its data than the
       USB write
// Timer1 returns 980us after broadcast - 0xF8FF
#define TIMER1_LOW_BYTE   0xFF
#define TIMER1_HIGH_BYTE  0xF8

// FLASH WRITING STATE AND NODE RESPONSES
#define DEFAULT 0
#define FLASH_WRITE 1
#define FLASH_READ 2
#define IDLE 3
#define NODE_FULL 0xFF
#define READ_DONE 0xF5
#define NODE_EMPTY 0xF0

// Initialization Routines
void Sysclk_Init(void);                          // Initialize the system clock(
      depends on Full/Low speed)
void Port_Init(void);                            // Configure ports for this
      specific application
void Timer_Init(void);             //initialize timers
void SPI_Init(void);
void Delay(void);                                // Approximately 80 us/1 ms on
      Full/Low Speed
#endif       /* _MAIN_BASE_H_ */
```

Appendix D

Application Code

Listing D.1: Windows select form code.

```
Public Class frmSelect
    Inherits System.Windows.Forms.Form

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.
         EventArgs) Handles MyBase.Load

        'device number, device string, and temp var newdevstring
        Dim DevNum As Integer
        Dim DevStr(SI_MAX_DEVICE_STRLEN) As Byte
        Dim i As Integer
        Dim iMax As Integer

        'clear the combo box
        cmbDevice.Items.Clear()

        'determine how many devices are hooked up
        Status = SI_GetNumDevices(DevNum)

        'if we find a device, obtain the name of each device
        'and convert the string to a vb string to add to the
        'combo list, otherwise display the error and close the
        'application
        If Status = SI_SUCCESS Then
            For i = 0 To DevNum - 1
                Status = SI_GetProductString(i, DevStr(0),
                     SI_RETURN_SERIAL_NUMBER)
                cmbDevice.Items.Insert(i, ConvertToVBString(DevStr))
            Next

            cmbDevice.SelectedIndex() = 0 'then set combo list to first item
        Else
            Me.Hide()
            MsgBox("Error finding USB device. Aborting application.")
            End
        End If


    End Sub

    Private Sub cmdOK_Click(ByVal sender As System.Object, ByVal e As System.
         EventArgs) Handles cmdOK.Click

        'when ok is clicked, set the timeouts on the device
        'and open the device
        Status = SI_SetTimeouts(0, 1000)
        Status = SI_Open(Convert.ToUInt32(cmbDevice.SelectedIndex),
             hUSBDevice)

        If Status <> SI_SUCCESS Then
            frmSelect.ActiveForm.Visible = False
            MsgBox("Error opening device: " + cmbDevice.Text + ". Application
                  is aborting. Reset hardware and try again.")
            End
        End If

        'create a new main form and show it
        Dim MainForm As New frmMain
        Me.Hide()
        MainForm.Show()
    End Sub

    Private Sub cmdCancel_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles cmdCancel.Click

        Me.Hide()
        End

    End Sub

    Private Sub BrowseButton_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles BrowseButton.Click
        FolderBrowserDialog1.ShowDialog()
        If FolderBrowserDialog1.SelectedPath <> "" Then
            ShowDir.Text = FolderBrowserDialog1.SelectedPath + "\"
            outputDir = FolderBrowserDialog1.SelectedPath + "\"
        End If
    End Sub
End Class
```

Listing D.2: Windows main form code for data logging application.

```
Imports System.IO

Public Class frmMain
    Inherits System.Windows.Forms.Form

    Dim fsWriteOneFile As FileStream
    Dim bWriteOneFile As TextWriter
    Dim filenameOneFile As String
    'Dim outputDir As String = "C:\datadump\"
    Dim pktsRx As Integer = 0
    Dim oldPktsRx As Integer = 0
    Dim ClickCount As Integer = 1
    Dim firstPacket As Boolean = True
    Dim OutPacket(6) As Byte
    Dim Timestamp(6) As Integer
    Dim TimeGap(6) As Integer
    Dim NodeCount(6) As Integer
    Dim LEDCounter As Integer = 0
    Dim PacketCount As Integer = 0
    Dim Mode As Integer = 255

    Private Sub frmMain_Load(ByVal sender As System.Object, ByVal e As System
         .EventArgs) Handles MyBase.Load
        filenameOneFile = outputDir + Filename.Text + ".txt"
        'filenameOneFile = outputDir + filenameOneFile
        fsWriteOneFile = New FileStream(filenameOneFile, FileMode.Create,
             FileAccess.Write)
        bWriteOneFile = New StreamWriter(fsWriteOneFile)
    End Sub

    Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.
         EventArgs) Handles Timer1.Tick
```

```
Dim IOBufSize As Integer = 4096
Dim IOBuf(IOBufSize) As Byte

Dim BytesSucceed As Integer
Dim BytesWriteRequest As Integer
Dim BytesReadRequest As Integer
Dim ReceivedBytes As Integer = 0
Dim Index As Integer = 0
Dim J As Integer = 0
Dim Node As Integer
Dim NodeHeader As Integer
Dim Temp As Byte = 0
Dim HostTime As Integer
Dim Length As Byte

BytesSucceed = 0
BytesReadRequest = IOBufSize
Status = SI_FlushBuffers(hUSBDevice, 0, 0)
'send read request
Status = SI_Write(hUSBDevice, OutPacket(0), 6, BytesSucceed)
'reset pending message in OutPacket
'(taken out for current basestation, sampling mode needs to be sent
     with every USB control packet
'OutPacket(0) = 0
If (Status = SI_SUCCESS) Then
    'read data from the device
    While Temp = 0
        Status = SI_Read(hUSBDevice, IOBuf(ReceivedBytes),
             BytesReadRequest, BytesSucceed)
        If (Status = SI_SUCCESS) Then
            If BytesSucceed > 0 Then
                ReceivedBytes = ReceivedBytes + BytesSucceed
            Else
                Temp = 1
            End If
        Else
            Temp = 1
        End If
    End While
    HostTime = Convert.ToInt32(DateAndTime.Timer()*1000)
    If ReceivedBytes > 0 Then
        RX.Text = ReceivedBytes
        While Index < ReceivedBytes - 4
            Header.Text = IOBuf(Index)
            If IOBuf(Index) = 31 Then
                Index = Index + 1
                PacketCount = PacketCount + 1
                PktCount.Text = PacketCount
            End If
            NodeHeader = IOBuf(Index)
            Index = Index + 1
            Temp = IOBuf(Index)
            Index = Index + 1
            Length = IOBuf(Index)
            Index = Index + 1
            Node = IOBuf(Index)
            If Node <> 0 Then
                bWriteOneFile.Write(Node)
                bWriteOneFile.Write(vbTab)
                bWriteOneFile.Write(HostTime)
                bWriteOneFile.Write(vbTab)
                bWriteOneFile.Write(Temp)
                If Length <= ReceivedBytes - Index Then
                    J = 0
                    While J < Length - 3
                        bWriteOneFile.Write(vbTab)
                        bWriteOneFile.Write(IOBuf(Index + J + 1) * 16
                                + (IOBuf(Index + J + 2) >> 4))
                        bWriteOneFile.Write(vbTab)
                        bWriteOneFile.Write((IOBuf(Index + J + 2) And
                                &HF) * 256 + IOBuf(Index + J + 3))
                        J = J + 3
                    End While
                    bWriteOneFile.WriteLine(vbTab, 1, 1)
                End If
            End If
            If Node = 0 Then
                TimeGap(0) = TimeGap(0) + Temp - Timestamp(0) - 1
                If Temp - Timestamp(0) - 1 < 0 Then
                    TimeGap(0) = TimeGap(0) + 256
                End If
                Timestamp(0) = Temp
                If NodeCount(0) = 0 Then
                    TimeGap(0) = 0
                    TimeGap(1) = 0
                    TimeGap(2) = 0
                    TimeGap(3) = 0
                    TimeGap(4) = 0
                    TimeGap(5) = 0
                    TimeGap(6) = 0
                    PacketCount = 1
                End If
                NodeCount(0) = NodeCount(0) + 1
                Label10.Text = (TimeGap(0) * 100) / PacketCount
            ElseIf Node = 1 Then
                TimeGap(1) = TimeGap(1) + Temp - Timestamp(1) - 1
                If Temp - Timestamp(1) - 1 < 0 Then
                    TimeGap(1) = TimeGap(1) + 256
                End If
                Timestamp(1) = Temp
                If NodeCount(1) = 0 Then
                    TimeGap(0) = 0
                    TimeGap(1) = 0
                    TimeGap(2) = 0
                    TimeGap(3) = 0
                    TimeGap(4) = 0
                    TimeGap(5) = 0
                    TimeGap(6) = 0
                    PacketCount = 1
                End If
                NodeCount(1) = NodeCount(1) + 1
                Label4.Text = (TimeGap(1) * 100) / PacketCount
            ElseIf Node = 2 Then
                TimeGap(2) = TimeGap(2) + Temp - Timestamp(2) - 1
                If Temp - Timestamp(2) - 1 < 0 Then
                    TimeGap(2) = TimeGap(2) + 256
                End If
                Timestamp(2) = Temp
                If NodeCount(2) = 0 Then
                    TimeGap(0) = 0
                    TimeGap(1) = 0
                    TimeGap(2) = 0
```

```
                        TimeGap(3) = 0
                        TimeGap(4) = 0
                        TimeGap(5) = 0
                        TimeGap(6) = 0
                        PacketCount = 1
                    End If
                    NodeCount(2) = NodeCount(2) + 1
                    Label5.Text = (TimeGap(2) * 100) / PacketCount
                ElseIf Node = 3 Then
                    TimeGap(3) = TimeGap(3) + Temp - Timestamp(3) - 1
                    If Temp - Timestamp(3) - 1 < 0 Then
                        TimeGap(3) = TimeGap(3) + 256
                    End If
                    Timestamp(3) = Temp
                    If NodeCount(3) = 0 Then
                        TimeGap(0) = 0
                        TimeGap(1) = 0
                        TimeGap(2) = 0
                        TimeGap(3) = 0
                        TimeGap(4) = 0
                        TimeGap(5) = 0
                        TimeGap(6) = 0
                        PacketCount = 1
                    End If
                    NodeCount(3) = NodeCount(3) + 1
                    Label6.Text = (TimeGap(3) * 100) / PacketCount
                ElseIf Node = 4 Then
                    TimeGap(4) = TimeGap(4) + Temp - Timestamp(4) - 1
                    If Temp - Timestamp(4) - 1 < 0 Then
                        TimeGap(4) = TimeGap(4) + 256
                    End If
                    Timestamp(4) = Temp
                    If NodeCount(4) = 0 Then
                        TimeGap(0) = 0
                        TimeGap(1) = 0
                        TimeGap(2) = 0
                        TimeGap(3) = 0
                        TimeGap(4) = 0
                        TimeGap(5) = 0
                        TimeGap(6) = 0
                        PacketCount = 1
                    End If
                    NodeCount(4) = NodeCount(4) + 1
                    Label31.Text = (TimeGap(4) * 100) / PacketCount
                ElseIf Node = 5 Then
                    TimeGap(5) = TimeGap(5) + Temp - Timestamp(5) - 1
                    If Temp - Timestamp(5) - 1 < 0 Then
                        TimeGap(5) = TimeGap(5) + 256
                    End If
                    Timestamp(5) = Temp
                    If NodeCount(5) = 0 Then
                        TimeGap(0) = 0
                        TimeGap(1) = 0
                        TimeGap(2) = 0
                        TimeGap(3) = 0
                        TimeGap(4) = 0
                        TimeGap(5) = 0
                        TimeGap(6) = 0
                        PacketCount = 1
                    End If
                    NodeCount(5) = NodeCount(5) + 1
                    Label32.Text = (TimeGap(5) * 100) / PacketCount
                ElseIf Node = 6 Then
                    TimeGap(6) = TimeGap(6) + Temp - Timestamp(6) - 1
                    If Temp - Timestamp(6) - 1 < 0 Then
                        TimeGap(6) = TimeGap(6) + 256
                    End If
                    Timestamp(6) = Temp
                    If NodeCount(6) = 0 Then
                        TimeGap(0) = 0
                        TimeGap(1) = 0
                        TimeGap(2) = 0
                        TimeGap(3) = 0
                        TimeGap(4) = 0
                        TimeGap(5) = 0
                        TimeGap(6) = 0
                        PacketCount = 1
                    End If
                    NodeCount(6) = NodeCount(6) + 1
                    Label33.Text = (TimeGap(6) * 100) / PacketCount
                End If
                Index = Index + Length
                Label2.Text = Node
                Label3.Text = Length
            End While
            Label7.Text = TimeGap(1)
            Label8.Text = TimeGap(2)
            Label9.Text = TimeGap(3)
            Label29.Text = TimeGap(4)
            Label39.Text = TimeGap(5)
            Label40.Text = TimeGap(6)
            Label11.Text = TimeGap(0)
        End If
    End If
End Sub

Private Sub cmdExit_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles cmdExit.Click
    Dim BytesSucceed As Integer

    OutPacket(0) = 255 'ff
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
    SI_Write(hUSBDevice, OutPacket(0), 6, BytesSucceed)
    SI_FlushBuffers(hUSBDevice, 0, 0)
    'close usc device and exit program
    Status = SI_Close(hUSBDevice)
    Me.Hide()
    End
End Sub

Private Sub Timer2_Tick(ByVal sender As System.Object, ByVal e As System.
     EventArgs)
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
    OutPacket(LEDCounter + 3) = 1
    If LEDCounter < 2 Then
        LEDCounter = LEDCounter + 1
    Else
        LEDCounter = 0
    End If
End Sub

Private Sub Sample_button_Click(ByVal sender As System.Object, ByVal e As
      System.EventArgs) Handles Sample_button.Click

    filenameOneFile = outputDir + Filename.Text + Convert.ToString(
         ClickCount) + ".txt"
    'filenameOneFile = outputDir + filenameOneFile
    fsWriteOneFile = New FileStream(filenameOneFile, FileMode.Create,
         FileAccess.Write)
    bWriteOneFile = New StreamWriter(fsWriteOneFile)
    ClickCount = ClickCount + 1
    Mode = 175
    OutPacket(0) = 175 'af
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 1
    OutPacket(4) = 1
    OutPacket(5) = 0
End Sub

Private Sub Erase_button_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles Erase_button.Click
    Mode = 190
    OutPacket(0) = 190 'be
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
End Sub

Private Sub Write_button_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles Write_button.Click
    Mode = 190
    OutPacket(0) = 190 'be
    OutPacket(1) = 255 'ff
    OutPacket(2) = FrameNumber.Value
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
End Sub

Private Sub Read_button_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles Read_button.Click
    Mode = 205
    OutPacket(0) = 205 'cd
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
    FrameNumber.Value = FrameNumber.Value + 1
End Sub

Private Sub Idle_button_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles Idle_button.Click
    Mode = 255
    OutPacket(0) = 255 'ff
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 0
    OutPacket(4) = 0
    OutPacket(5) = 0
End Sub

Private Sub LEDButton_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles LEDButton.Click
    OutPacket(0) = Mode
    OutPacket(1) = NodeSelect.Value
    OutPacket(2) = 1
    OutPacket(3) = RedLevel.Value
    OutPacket(4) = GreenLevel.Value
    OutPacket(5) = BlueLevel.Value
End Sub
End Class
```

Listing D.3: Windows main form code for data visualization application.

```
Imports System.Runtime.InteropServices
Imports System.IO
Imports SoftwareFX.ChartFX.Lite

Public Class frmMainChart
    Inherits System.Windows.Forms.Form

    Dim OutPacket(6) As Byte
    Dim Timestamp(6) As Integer
    Dim TimeGap(6) As Integer
    Dim NodeCount(6) As Integer
    Dim Mode As Integer = 255
    Dim PlotThisNode As Integer = 4
    Dim IOBufSize As Integer = 4096
    Dim IOBuf(4096) As Byte
    Dim PlotBufSize As Integer = 100
    Dim MaxNumNodes As Integer = 25
    Dim MaxSensorVals As Integer = 10
    Dim PlotBuf(25, 10, 100) As Integer
    Dim PlotBufWriteIdx As Integer = 0

    Private Sub frmMainChart_Load(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles MyBase.Load
        Dim i As Integer
        OutPacket(0) = 255
        Chart1.OpenData(COD.Values, 3, PlotBufSize)
        Chart2.OpenData(COD.Values, 3, PlotBufSize)
        Chart3.OpenData(COD.Values, 3, PlotBufSize)
        Chart1.Value(0, 0) = 0
        Chart2.Value(0, 0) = 0
        For i = 0 To PlotBufSize - 1
            Chart3.Value(0, i) = 0
        Chart3.Value(1, i) = 0
        Chart3.Value(2, i) = 0
    Next i
    Chart1.CloseData(COD.Values)
    Chart2.CloseData(COD.Values)
    Chart3.CloseData(COD.Values)
    Chart3.SerLeg(0) = "Node 1 to Node 5"
    Chart3.SerLeg(1) = "Node 1 to Node 9"
    Chart3.SerLeg(2) = "Node 5 to Node 9"
End Sub

Private Sub SampleButton_Click(ByVal sender As System.Object, ByVal e As
     System.EventArgs) Handles SampleButton.Click
    Mode = 175
    OutPacket(0) = 175 'af
    OutPacket(1) = 255
    OutPacket(2) = 1
    OutPacket(3) = 2
    OutPacket(4) = 2
    OutPacket(5) = 0
End Sub

Private Sub TimerA_Tick(ByVal sender As System.Object, ByVal e As System.
     EventArgs) Handles TimerA.Tick
    Dim BytesSucceed As Integer
    Dim BytesWriteRequest As Integer
    Dim ReceivedBytes As Integer = 0
    Dim Index As Integer = 0
    Dim J As Integer
    Dim K As Integer
    Dim Node As Integer
    Dim NodeHeader As Integer
    Dim Time As Byte = 0
    Dim Debug As Integer
    Dim Length As Byte
    Dim Temp As Byte = 0

    BytesSucceed = 0
    'Status = SI_FlushBuffers(hUSBDevice, 0, 0)
    'send read request
    Status = SI_Write(hUSBDevice, OutPacket(0), 6, BytesSucceed)
    'reset pending message in OutPacket
    '(taken out for current basestation, sampling mode needs to be sent
         with every USB control packet
    'OutPacket(0) = 0
    If (Status = SI_SUCCESS) Then
        'read data from the device
        While Temp = 0
            Status = SI_Read(hUSBDevice, IOBuf(ReceivedBytes), 4096,
                 BytesSucceed)
            If (Status = SI_SUCCESS) Then
                If BytesSucceed > 0 Then
                    ReceivedBytes = ReceivedBytes + BytesSucceed
                Else
                    Temp = 1
                End If
            Else
                Temp = 1
            End If
        End While
        'unpack IOBuf
        If ReceivedBytes > 0 Then
            While Index < ReceivedBytes - 4
                If IOBuf(Index) = 31 Then
                    Index = Index + 1
                    'global packet header found, increment write index
                         into PlotBuf
                    Temp = (PlotBufWriteIdx + 1) Mod PlotBufSize
                    'initialize all values with last sample values
                    For J = 0 To MaxNumNodes - 1
                        For K = 0 To MaxSensorVals - 1
                            PlotBuf(J, K, Temp) = PlotBuf(J, K,
                                 PlotBufWriteIdx)
                        Next K
                    Next J
                    PlotBufWriteIdx = Temp
                End If
                NodeHeader = IOBuf(Index)
                Index = Index + 1
                Time = IOBuf(Index)
                Index = Index + 1
                Length = IOBuf(Index)
                Index = Index + 1
                Node = IOBuf(Index)
                If Node > 0 Then
                    Node = Node - 1
                    If Length <= ReceivedBytes - Index And Length <= 16
                         Then
                        J = 0
                        K = 0
                        While J < Length - 3
                            PlotBuf(Node, K, PlotBufWriteIdx) = IOBuf(
                                 Index + J + 1) * 16 + (IOBuf(Index + J +
                                  2) >> 4)
                            PlotBuf(Node, K + 1, PlotBufWriteIdx) = ((
                                 IOBuf(Index + J + 2) And &HF) * 256 +
                                 IOBuf(Index + J + 3))
                            J = J + 3
                            K = K + 2
                        End While
                    End If
                End If
                Index = Index + Length
            End While
            If Mode = 175 Then
                'plot new data for PlotThisNode
                Chart1.OpenData(COD.Values, 3, PlotBufSize)
                Chart2.OpenData(COD.Values, 3, PlotBufSize)
                Chart3.OpenData(COD.Values, 3, PlotBufSize)
                For J = 0 To PlotBufSize - 1
                    Temp = (PlotBufWriteIdx + PlotBufSize - J) Mod
                         PlotBufSize
                    For K = 0 To 2
                        Chart1.Value(K, J) = PlotBuf(PlotThisNode - 1, K,
                             Temp)
                        Chart2.Value(K, J) = PlotBuf(PlotThisNode - 1, K
                             + 3, Temp)
                        'Chart3.Value(K, J) = System.Math.Log(PlotBuf(0,
                             K + 6, Temp) + 1) * 492
                    Next K
                    Chart3.Value(0, J) = System.Math.Log(PlotBuf(0, 6,
                         Temp) + 1) * 492
                        Chart3.Value(1, J) = System.Math.Log(PlotBuf(0, 7,
                             Temp) + 1) * 492
                        Chart3.Value(2, J) = System.Math.Log(PlotBuf(4, 6,
                             Temp) + 1) * 492
                    Next J
                    Chart1.CloseData(COD.Values)
                    Chart2.CloseData(COD.Values)
                    Chart3.CloseData(COD.Values)
                End If
            End If
        End If
    End Sub

    Private Sub IdleButton_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles IdleButton.Click
        Mode = 255
        OutPacket(0) = 255 'ff
        OutPacket(1) = 255
        OutPacket(2) = 1
        OutPacket(3) = 0
        OutPacket(4) = 0
        OutPacket(5) = 0
    End Sub

    Private Sub NumericUpDown1_ValueChanged(ByVal sender As System.Object,
         ByVal e As System.EventArgs) Handles NumericUpDown1.ValueChanged
        PlotThisNode = NumericUpDown1.Value
    End Sub

    Private Sub ExitButton_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles ExitButton.Click
        Dim BytesSucceed As Integer

        OutPacket(0) = 255 'ff
        OutPacket(1) = 255
        OutPacket(2) = 1
        OutPacket(3) = 0
        OutPacket(4) = 0
        OutPacket(5) = 0
        SI_Write(hUSBDevice, OutPacket(0), 6, BytesSucceed)
        SI_FlushBuffers(hUSBDevice, 0, 0)
        'close usc device and exit program
        SI_Close(hUSBDevice)
        Me.Hide()
        End
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles Button1.Click
        PlotThisNode = 1
        NumericUpDown1.Value = 1
    End Sub

    Private Sub Button5_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles Button5.Click
        PlotThisNode = 5
        NumericUpDown1.Value = 5
    End Sub

    Private Sub Button9_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs) Handles Button9.Click
        PlotThisNode = 9
        NumericUpDown1.Value = 9
    End Sub
End Class
```

Listing D.4: Python script for recording data in Mac OSX.

```
import sys
import usb
import time
import struct
import array
import math

class DeviceDescriptor(object) :
    def __init__(self, vendor_id, product_id, interface_id) :
        self.vendor_id = vendor_id
        self.product_id = product_id
        self.interface_id = interface_id

    def getDevice(self) :
        """
        Return the device corresponding to the device descriptor if it is
        available on a USB bus.  Otherwise, return None.  Note that the
        returned device has yet to be claimed or opened.
        """
        buses = usb.busses()
        for bus in buses :
            for device in bus.devices :
                if device.idVendor == self.vendor_id :
                    if device.idProduct == self.product_id :
                        return device
        return None

class SensembleUSBDevice(object) :

    DEV_VENDOR_ID = 0x10C4
    DEV_PRODUCT_ID = 0xEA61
    DEV_INTERFACE_ID = 0
    DEV_BULK_IN_EP = 2
    DEV_BULK_OUT_EP = 2

    def __init__(self) :
        self.device_descriptor = DeviceDescriptor(SensembleUSBDevice.
             DEV_VENDOR_ID,
                                                  SensembleUSBDevice.
                                                       DEV_PRODUCT_ID,
                                                  SensembleUSBDevice.
                                                       DEV_INTERFACE_ID)
        self.device = self.device_descriptor.getDevice()
        self.handle = None

    def open(self) :
        self.device = self.device_descriptor.getDevice()
        self.handle = self.device.open()
        if sys.platform == 'darwin' :
            # XXX : For some reason, Mac OS X doesn't set the
            # configuration automatically like Linux does.
            self.handle.setConfiguration(1)
        self.handle.claimInterface(self.device_descriptor.interface_id)
  #USBXpress API message
  self.handle.controlMsg(64,2,0,value=2,index=0,timeout=100)

    def close(self) :
  self.handle.bulkWrite(SensembleUSBDevice.DEV_BULK_OUT_EP,
                              chr(0xFF)+chr(0xFF)+chr(1)+chr(0)+chr(0)+chr(0)
                                                                                ,
                              10)
        self.handle.releaseInterface()
  #USBXpress API message
  self.handle.controlMsg(64,2,0,value=4,index=0,timeout=100)
  self.handle.reset()

    def sendMessage(self) :
  self.handle.bulkWrite(SensembleUSBDevice.DEV_BULK_OUT_EP,
        chr(0xAF)+chr(0xFF)+chr(1)+chr(1)+chr(1)+chr(0),
        10)

    def getDataPacket(self, bytesToGet) :
        """
        Assume bytesToGet is two bytes wide.
        """
        return self.handle.bulkRead(SensembleUSBDevice.DEV_BULK_IN_EP,
                                    bytesToGet,
                                    1)

def logData(dev,
      filename="Sensors.dat",
      maxbytesPerBlock=2048,
      packetsToGet=-1) :

    f = file(filename, 'w')
    if packetsToGet == -1 :
        print "\n To stop data collection, type <CTRL> + c."
    #for now automatically start sampling
    starttime=time.time()
    while True :
        try :
     dev.sendMessage()
            data = dev.getDataPacket(maxbytesPerBlock)
     bytesReceived=len(data)
     #print bytesReceived
     #convert data from signed char to unsigned int
     data = struct.unpack('B'*bytesReceived,struct.pack('b'*bytesReceived,*
          data))
     mytime=int((time.time()-starttime)*1000)
     if data != None :
    if data[0]==31 :
         #unpack samples for each node received
     index = 1
     while index < bytesReceived-3 :
         if data[index] == 172 :
       thislength = data[index+2]
       if data[index+3]!=0 :
         samples = []
           #store node ID
         samples.append(data[index+3])
         #store host timestamp
         samples.append(mytime)
         #store node timestamp
         samples.append(data[index+1])
         index += 3
         if thislength <= bytesReceived-index+1 :
           j = 0
           while j < thislength-3 :
              samples.append(long((data[index+j+1]<<4)&4080) + long((data[
                   index+j+2]>>4)&15))
              samples.append(long((data[index+j+2]<<8)&3840) + long((data[
                   index+j+3])&255))
              j+=3
            format_string = ("%d"+"\t")*(len(samples)-1)+"%d\n"
            f.write(format_string % tuple(samples))
          else :
            break
          index += thislength
        else :
          index += thislength+3
      else :
        index += 1

        except KeyboardInterrupt :
              print "You have successfully logged data."
              f.close()
              return
        except Exception, e:
              print "An error has occurred."
              print e
              f.close()
              sys.exit(1)

def main(argv=None) :
```

```
    if argv is None :
        script_name = sys.argv[0]
        argv = sys.argv[1:]
    if len(argv) == 1 :
        option = argv[0]
        filename = "SensembleLog.dat"
    elif len(argv) == 2 :
        option = argv[0]
        filename = argv[1]
    else :
        option = None
        filename = None

    if option == 'log' :
        dev = SensembleUSBDevice()
        dev.open()
        logData(dev, filename)
        dev.close()
    else :
        print "Usage: python -i %s OPTION [FILENAME]" % script_name
        print "  where OPTION can only be 'log' at this time"

if __name__ == "__main__" :
    data = main()
```

Listing D.5: Source code for rawusb Max object.

```
#include "ext.h"
#include "usb.h"

void *rawusb_class;
//t_symbol *ps_list;
struct usb_device *my_device;
struct usb_dev_handle *my_handle;
unsigned char data_out[6]={0xAF,0xFF,0x01,2,2,0}; //default message
unsigned char dummy[1024];
#define BUFFER_SIZE 1024 //4096 bytes supports 40 nodes with usb buffer read
      every 50ms
#define NUM_OUTLETS 12
#define BLOCK_HEADER 31
#define NODE_HEADER 172
#define NODE_PACKET_SIZE 16

//data structure for rawusb
typedef struct {
  t_object r_ob;
  t_atom mylist[NUM_OUTLETS];
  void *r_list;
  void *r_clock;
} t_rawusb;

//data buffer
struct r_buffer{
  unsigned char data[BUFFER_SIZE];
  unsigned int outval[NUM_OUTLETS];
  unsigned int offset;
};
struct r_buffer BUFFER;

//void *rawusb_new(long interval); //object creation method
void *rawusb_new(void); //object creation method
void rawusb_bang(t_rawusb *x);  //receive bang method
void rawusb_tick(t_rawusb *x);  //clock tick method
void rawusb_stop(t_rawusb *x); //receive stop message method
void rawusb_free(t_rawusb *x); //free method
int find_my_device(void);
int open_my_device(void);
int close_my_device(void);
int send_device_message(void);
void get_data_block(int bytes_requested, int *bytes_received);

int main()
{
  //setup creates class definition
  setup((t_messlist **)&rawusb_class, (method)rawusb_new, (method)rawusb_free
      , (short)sizeof(t_rawusb), 0L,0L, 0);
  addbang((method)rawusb_bang); //binds rawusb_bang method to bang message
  addmess((method)rawusb_stop,"stop",0);
  return (0);
}


void rawusb_bang(t_rawusb *x)
{
  clock_fdelay(x->r_clock, 0.); //start the clock
}

void rawusb_tick(t_rawusb *x)
{
  int bytes_recv=0;
  int i=0;
  int j,k;
  char length;

  clock_fdelay(x->r_clock, 10.); //schedule next tick
  get_data_block(BUFFER_SIZE,&bytes_recv);
  if(bytes_recv > 0){
    if(BUFFER.data[i++] == BLOCK_HEADER){
      while(i<(bytes_recv-3)){
        if(BUFFER.data[i++] == NODE_HEADER){
          SETLONG(x->mylist+1,BUFFER.data[i++]); //timestamp
          length = BUFFER.data[i++]; //length
          SETLONG(x->mylist,BUFFER.data[i]); //node ID
          if(BUFFER.data[i] > 0){
            if(length <= (bytes_recv - i) && length == NODE_PACKET_SIZE){
              k = 2;
              j = 0;
              while(j<(length-3) && k<(NUM_OUTLETS-1)){
                SETLONG(x->mylist+k,((unsigned int)BUFFER.data[i+j+1]<<4)+((
                     unsigned int)BUFFER.data[i+j+2]>>4));
                SETLONG(x->mylist+k+1,((unsigned int)(BUFFER.data[i+j+2]&0x0F
                     )<<8)+(unsigned int)BUFFER.data[i+j+3]);
                k=k+2;
                j=j+3;
              }
            }
            else{
              break;
            }
            outlet_list(x->r_list,0L,NUM_OUTLETS,x->mylist);
          }
          i=i+length;
        }
      }
    }
  }
}

void rawusb_stop(t_rawusb *x)
{
  clock_unset(x->r_clock);
  //temporary dealing with device firmware
  if(!close_my_device()){
    post("device could not be closed");
    return;
  }
  if(!open_my_device()){
    post("device could not be opened");
    return;
  }
  post("device ready");
}

void rawusb_free(t_rawusb *x)
{
  int temp;
  freeobject((t_object *)x->r_clock); //free the clock
  temp = usb_bulk_read(my_handle,2,dummy,BUFFER_SIZE,2);
  if(temp > 0){
    post("flushed %d bytes from buffer",temp);
  }
  //if open, close usb device
  if(!close_my_device()){
    post("device could not be closed");
  }
  else{
    post("device closed nicely");
  }
}

int find_my_device(void)
{
  struct usb_bus *bus,*busses;

  usb_find_busses();
  usb_find_devices();
  busses = usb_get_busses();
  for(bus=busses;bus;bus=bus->next){
    struct usb_device *dev;
    for(dev=bus->devices;dev;dev=dev->next){
      if(dev->descriptor.idVendor == 0x10C4){
        if(dev->descriptor.idProduct == 0xEA61){
          my_device=dev;
          return 1;
        }
      }
    }
  }
  return 0;
}

int open_my_device(void)
{
  my_handle = usb_open(my_device);
  //set config for darwin ...
  if(usb_set_configuration(my_handle,1)){
    return 0;
  }
  if(usb_claim_interface(my_handle,0)){
    return 0;
  }
  //USB Xpress proprietary control messages
  usb_control_msg(my_handle,64,2,2,0,0,0,100);
  return 1;
}

int close_my_device(void)
{
  data_out[0] = 0xFF;
  data_out[3] = 0;
  data_out[4] = 0;
  if(send_device_message() <= 0){
    post("couldn't send message");
    return 0;
  }
  data_out[0] = 0xAF;
  data_out[3] = 2;
  data_out[4] = 2;

  //USB Xpress proprietary control messages
  usb_control_msg(my_handle,64,2,4,0,0,0,20);

  if(usb_release_interface(my_handle,0)){
    return 0;
  }
  if(usb_close(my_handle)){
    return 0;
  }
  return 1;
}

int send_device_message(void)
{
  int temp;

  temp=usb_bulk_write(my_handle,2,data_out,6,1);
  return temp;
}

void get_data_block(int bytes_requested, int *bytes_received)
{
  usb_bulk_write(my_handle,2,data_out,6,1); //this can be any message not
       beginning with 0xFF
  *bytes_received = usb_bulk_read(my_handle,2,BUFFER.data,bytes_requested,1);
}

void *rawusb_new(void)
{
  t_rawusb *x;
  //create new instance and return pointer to instance
  x = (t_rawusb *)newobject(rawusb_class);
  x->r_clock = clock_new(x,(method)rawusb_tick); //create new clock
  x->r_list=listout(x);

  //find and open usb device
  usb_init();
  if(!find_my_device()){
    post("device not found");
  }
  else if(!open_my_device()){
    post("device could not be opened");
  }

  return x;
}
```

Listing D.6: Source code for **xcov** Max object.

```
#include "ext.h"
#include "complex.h"
#include "math.h"
#include "fftw3.h"
```

```
void *xcov_class;

#define BUFFER_SIZE 4096
#define MAX_OUT 2049 //must be BUFFER_SIZE/2+1

//data structure for xcov
typedef struct {
  t_object r_ob;
  t_atom *fft_left_list;
  t_atom *fft_right_list;
  t_atom *result_list;
  int outlength;
  long lagtime;
  double peakval;
  double leftmean;
  double rightmean;
  double *fftleft;
  double *fftright;
  double *result;
  //buffer for two streams of synchronous incoming data
  long *leftdata;
  long *rightdata;
  long righttemp;
  unsigned int w_ptr;
  unsigned int leftflag;
  //settings
  int window_size;
  int step_size;
  //FFT
  double *fft_in;
  fftw_complex *fft_out;
  fftw_plan fft_plan;
  fftw_complex *ifft_in;
  double *ifft_out;
  fftw_plan ifft_plan;
  fftw_complex *Astore;
  fftw_complex *Bstore;
  //ports
  void *r_vector;
  void *r_fftleft;
  void *r_fftright;
  void *r_meanright;
  void *r_meanleft;
  void *r_peakval;
  void *r_lagtime;
} t_xcov;


void *xcov_new(long window, long step);
void xcov_free(t_xcov *x);
void xcov_int(t_xcov *x, long leftval);
void xcov_in1(t_xcov *x, long rightval);
void xcov_assist(t_xcov *x, Object *b, long msg, long arg, char *s);
void compute_xcov(t_xcov *x);

int main()
{
  //setup creates class definition
  setup((t_messlist **)&xcov_class, (method)xcov_new, (method)xcov_free, (
       short)sizeof(t_xcov), 0L,A_DEFLONG,A_DEFLONG, 0);
  addint((method)xcov_int);   //binds xcov_int method to int message in left
       inlet
  addinx((method)xcov_in1,1);   //binds xcov_in1 method to int message in
       right inlet
  addmess((method)xcov_assist,"assist",A_CANT,0); //binds xcov_assist method
       to assist message
  finder_addclass("All Objects","xcov"); //adds class to object browser
  return (0);
}

void xcov_int(t_xcov *x, long leftval)
{
  x->leftdata[x->w_ptr]=leftval;
  x->rightdata[x->w_ptr]=x->righttemp;
  x->leftflag++;
  if(x->leftflag==x->step_size){
    int i;

    compute_xcov(x);
    outlet_int(x->r_lagtime,x->lagtime);
    outlet_float(x->r_peakval,x->peakval);
    outlet_float(x->r_meanright,x->rightmean);
    outlet_float(x->r_meanleft,x->leftmean);
    for(i=0;i<x->outlength;i++){
      SETFLOAT(x->fft_left_list+i,x->fftleft[i]);
      SETFLOAT(x->fft_right_list+i,x->fftright[i]);
    }
    for(i=0;i<x->window_size;i++){
      SETFLOAT(x->result_list+i,x->result[i]);
    }
    outlet_list(x->r_fftright,0L,x->outlength,x->fft_right_list);
    outlet_list(x->r_fftleft,0L,x->outlength,x->fft_left_list);
    outlet_list(x->r_vector,0L,x->window_size,x->result_list);
    x->leftflag=0;
  }
  x->w_ptr = (x->w_ptr + 1)%BUFFER_SIZE;
}

void xcov_in1(t_xcov *x, long rightval)
{
  x->righttemp=rightval;
  //for every right val, we expect an incoming left val in order to register
       a sample!
}

void xcov_assist(t_xcov *x, Object *b, long msg, long arg, char *s)
{
  if(msg == ASSIST_INLET){
    switch(arg){
      case 0: sprintf(s,"%s","Left Value Triggers Sample Stored To Buffer");
          break;
      case 1: sprintf(s,"%s","Right Value Must Be Accompanied By Left Value")
           ;
    }
  }
  else if(msg == ASSIST_OUTLET){
    switch(arg){
      case 0: sprintf(s,"%s","Cross-Covariance Vector, As List");
          break;
      case 1: sprintf(s,"%s","FFT of Left Input Minus Mean, As List");
          break;
      case 2: sprintf(s,"%s","FFT of Right Input Minus Mean, As List");
          break;
      case 3: sprintf(s,"%s","Mean of the Left Input Over Current Window, As
            Float");
          break;
      case 4: sprintf(s,"%s","Mean of the Right Input Over Current Window, As
             Float");
          break;
      case 5: sprintf(s,"%s","Value of Peak Cross-Covariance, As Float");
          break;
      case 6: sprintf(s,"%s","Lag Index of Peak Cross-Covariance, As Long");
          break;
      default:sprintf(s,"%s","No Designated Output");
    }
  }
}

void compute_xcov(t_xcov *x)
{
  int i,j;
  complex *A;
  complex *B;
  double amean=0;
  double bmean=0;
  double aa0=0;
  double bb0=0;
  double scale;

  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    amean += x->leftdata[j];
    bmean += x->rightdata[j];
  }
  x->leftmean = amean/x->window_size;
  x->rightmean = bmean/x->window_size;

  //this will be fft(left input)
  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    x->fft_in[i]=x->leftdata[j]-x->leftmean;
    aa0 += x->fft_in[i]*x->fft_in[i]; //needed to compute autocrosscov at
          zero lag
  }
  fftw_execute(x->fft_plan);
  A = x->fft_out;
  for(i=0;i<x->outlength;i++){
    x->fftleft[i] = sqrt(creal(*(A+i))*creal(*(A+i)) + cimag(*(A+i))*cimag(*(
          A+i)));
    x->Astore[i]=*(x->fft_out+i);
  }

  //this will be conj(fft(right input)) for real input
  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr -  i)%BUFFER_SIZE;
    x->fft_in[i]=x->rightdata[j]-x->rightmean;
    bb0 += x->fft_in[i]*x->fft_in[i]; //needed to compute autocrosscov at
          zero lag
  }
  fftw_execute(x->fft_plan);
  B = x->fft_out;
  for(i=0;i<x->outlength;i++){
    //magnitude of conj(fft) = mag(fft) for real input
    x->fftright[i] = sqrt(creal(*(B+i))*creal(*(B+i))+cimag(*(B+i))*cimag(*(B
          +i)));
    x->Bstore[i]=*(x->fft_out+i);
  }

  //compute autocrosscov at zero lag
  scale = sqrt(aa0*bb0);
  //compute cross covariance
  for(i=0;i<x->outlength;i++){
    x->ifft_in[i]=x->Astore[i]*x->Bstore[i];
  }
  fftw_execute(x->ifft_plan);
  x->peakval = 0;
  x->lagtime = 0;
  for(i=0;i<x->window_size;i++){
    j=(i+(x->window_size>>1))%x->window_size;
    x->result[i]=x->ifft_out[j]/(x->window_size*scale);
    if(x->result[i]>=x->peakval){
      x->peakval=x->result[i];
      x->lagtime=i;
    }
  }

}

void *xcov_new(long window, long step)
{
  int temp=2;
  t_xcov *x;
  //create new instance and return pointer to instance
  x = (t_xcov *)newobject(xcov_class);
  //initialize buffer offsets and flags
  x->w_ptr = 0;
  x->leftflag = 0;
  //set window and step size
  if(window > BUFFER_SIZE){
    x->window_size = BUFFER_SIZE;
  }
  else{
    while(temp < window){
      temp=temp<<1;
    }
    x->window_size = temp;
  }
  post("using window size %ld",x->window_size);
  if(step > x->window_size){
    x->step_size = x->window_size;
  }
  else{
    x->step_size=(int)step;
  }
  post("using step size %ld",x->step_size);
  x->outlength=(x->window_size>>1)+1;
  if(x->outlength > MAX_OUT){
    x->outlength=MAX_OUT;
  }
  post("output will be size %ld",x->outlength);

  //set up fftw data structure
  x->fft_in=(double*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
```

```
    x->fft_out=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
    x->fft_plan=fftw_plan_dft_r2c_1d(x->window_size,x->fft_in,x->fft_out,
         FFTW_ESTIMATE);
    x->ifft_in=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
    x->ifft_out=(double*)fftw_malloc(sizeof(double)*x->window_size);
    x->ifft_plan=fftw_plan_dft_c2r_1d(x->window_size,x->ifft_in,x->ifft_out,
         FFTW_ESTIMATE);

    x->Astore=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->outlength);
    x->Bstore=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->outlength);

    //allocate other arrays
    x->leftdata=(long*)fftw_malloc(sizeof(long)*BUFFER_SIZE);
    x->rightdata=(long*)fftw_malloc(sizeof(long)*BUFFER_SIZE);
    x->fft_left_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
    x->fft_right_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
    x->result_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
    x->fftleft=(double*)fftw_malloc(sizeof(double)*x->outlength);
    x->fftright=(double*)fftw_malloc(sizeof(double)*x->outlength);
    x->result=(double*)fftw_malloc(sizeof(double)*x->window_size);

    //create outlets
    x->r_lagtime=intout(x);
    x->r_peakval=floatout(x);
    x->r_meanright=floatout(x);
    x->r_meanleft=floatout(x);
    x->r_fftright=listout(x);
    x->r_fftleft=listout(x);
    x->r_vector=listout(x);
    intin(x,1); //create the right inlet

    return x;
  }


  void xcov_free(t_xcov *x)
  {
    //free the fftw data structure
    fftw_destroy_plan(x->fft_plan);
    fftw_destroy_plan(x->ifft_plan);
    fftw_free(x->fft_in);
    fftw_free(x->fft_out);
    fftw_free(x->ifft_in);
    fftw_free(x->ifft_out);
    fftw_free(x->Astore);
    fftw_free(x->Bstore);
    fftw_free(x->leftdata);
    fftw_free(x->rightdata);
    fftw_free(x->fft_left_list);
    fftw_free(x->fft_right_list);
    fftw_free(x->result_list);
    fftw_free(x->fftleft);
    fftw_free(x->fftright);
    fftw_free(x->result);
  }
```

Listing D.7: Source code for **xcovlist** Max object.

```
#include "ext.h"
#include "complex.h"
#include "math.h"
#include "fftw3.h"

void *xcov_class;

#define BUFFER_SIZE 4096
#define MAX_OUT 2049 //must be BUFFER_SIZE/2+1
#define MAX_IN 10
#define XCOV_PROXY_GETINLET(x) (proxy_getinlet? proxy_getinlet((t_object *)x)
       : x->r_proxyinletnum)

typedef struct{
  long rightdata[MAX_IN];
  long leftdata[MAX_IN];
}t_list;

//data structure for xcov
typedef struct {
  t_object r_ob;

  t_atom *fft_left_list;
  t_atom *fft_right_list;
  t_atom *result_list;

  int outlength;
  long lagtime;
  double peakval;
  double leftmean;
  double rightmean;
  double *fftleft;
  double *fftright;
  double *result;

  //buffer for two streams of synchronous incoming data
  long *leftdata;
  long *rightdata;
  long righttemp;
  unsigned int w_ptr;
  unsigned int leftflag;
  //test for list input
  t_list *listdata;
  long listtemp[MAX_IN];
  unsigned int list_ptr;
  unsigned int listflag;
  //settings
  int window_size;
  int step_size;
  int listlength;
  //FFT
  double *fft_in;
  fftw_complex *fft_out;
  fftw_plan fft_plan;
  fftw_complex *ifft_in;
  double *ifft_out;
  fftw_plan ifft_plan;
  fftw_complex *Astore;
  fftw_complex *Bstore;
  //ports
  void *r_proxy;
  long r_proxyinletnum;
  void *r_vector;
  void *r_fftleft;
  void *r_fftright;
  void *r_meanright;
  void *r_meanleft;
  void *r_peakval;
  void *r_lagtime;
} t_xcov;


void *xcov_new(long window, long step, long listlen);
void xcov_free(t_xcov *x);
void xcov_int(t_xcov *x, long val);
void xcov_assist(t_xcov *x, Object *b, long msg, long arg, char *s);
void compute_xcov(t_xcov *x);
void xcov_list(t_xcov *x, t_symbol *s, short argc, t_atom *argv);

int main()
{
  setup((t_messlist **)&xcov_class, (method)xcov_new, (method)xcov_free, (
      short)sizeof(t_xcov), 0L,A_DEFLONG,A_DEFLONG,A_DEFLONG,0);
  //addint((method)xcov_int);   //binds xcov_int method to int message in
      left inlet
  addmess((method)xcov_assist,"assist",A_CANT,0); //binds xcov_assist method
      to assist message
  addmess((method)xcov_list, "list", A_GIMME, 0);
  finder_addclass("All Objects","xcov"); //adds class to object browser
  return (0);
}

void xcov_list(t_xcov *x, t_symbol *s, short argc, t_atom *argv){
  int i;

  if(XCOV_PROXY_GETINLET(x) == 0){
    for(i=0;i<x->listlength;i++){
      if(i<argc && argv[i].a_type == A_LONG){
        x->listdata[x->list_ptr].leftdata[i]=argv[i].a_w.w_long;
      }
      else{
        x->listdata[x->list_ptr].leftdata[i]=1;
      }
      x->listdata[x->list_ptr].rightdata[i]=x->listtemp[i];
    }
    x->listflag++;
    if(x->listflag==x->step_size){
      //make the calculation
      compute_xcov(x);
      outlet_int(x->r_lagtime,x->lagtime);
      outlet_float(x->r_peakval,x->peakval);
      outlet_float(x->r_meanright,x->rightmean);
      outlet_float(x->r_meanleft,x->leftmean);
      for(i=0;i<x->outlength;i++){
        SETFLOAT(x->fft_left_list+i,x->fftleft[i]);
        SETFLOAT(x->fft_right_list+i,x->fftright[i]);
      }
      for(i=0;i<x->window_size;i++){
        SETFLOAT(x->result_list+i,x->result[i]);
      }
      outlet_list(x->r_fftright,0L,x->outlength,x->fft_right_list);
      outlet_list(x->r_fftleft,0L,x->outlength,x->fft_left_list);
      outlet_list(x->r_vector,0L,x->window_size,x->result_list);
      x->listflag=0;
    }
    x->list_ptr = (x->list_ptr + 1)%BUFFER_SIZE;
  }//inlet was '0'
  else{
    //for every right val, we expect an incoming left val in order to
        register a sample!
    for(i=0;i<x->listlength;i++){
      if(i<argc && argv[i].a_type == A_LONG){
        x->listtemp[i]=argv[i].a_w.w_long;
      }
      else{
        x->listtemp[i]=1;
      }
    }
  }
}

void xcov_assist(t_xcov *x, Object *b, long msg, long arg, char *s)
{
  if(msg == ASSIST_INLET){
    switch(arg){
      case 0: sprintf(s,"%s","Left Value Triggers Sample Stored To Buffer");
          break;
      case 1: sprintf(s,"%s","Right Value Must Be Accompanied By Left Value")
          ;
    }
  }
  else if(msg == ASSIST_OUTLET){
    switch(arg){
      case 0: sprintf(s,"%s","Cross-Covariance Vector, As List");
          break;
      case 1: sprintf(s,"%s","FFT of Left Input Minus Mean, As List");
          break;
      case 2: sprintf(s,"%s","FFT of Right Input Minus Mean, As List");
          break;
      case 3: sprintf(s,"%s","Mean of the Left Input Over Current Window, As
           Float");
          break;
      case 4: sprintf(s,"%s","Mean of the Right Input Over Current Window, As
            Float");
          break;
      case 5: sprintf(s,"%s","Value of Peak Cross-Covariance, As Float");
          break;
      case 6: sprintf(s,"%s","Lag Index of Peak Cross-Covariance, As Long");
          break;
      default:sprintf(s,"%s","No Designated Output");
    }
  }
}

void compute_xcov(t_xcov *x)
{
  int i,j,k;
  complex *A;
  complex *B;
  double scale;
```

```c
x->leftmean = 0;
x->rightmean = 0;
x->peakval = 0;
x->lagtime = 0;
for(i=0;i < x->outlength;i++){
  x->fftleft[i]=0;
  x->fftright[i]=0;
}
for(i=0; i < x->window_size;i++){
  x->result[i]=0;
}

for(k=0;k < x->listlength;k++){
  double amean=0;
  double bmean=0;
  double aa0=0;
  double bb0=0;

  for(i=0;i < x->window_size;i++){
    j = (x->list_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    amean += x->listdata[j].leftdata[k];
    bmean += x->listdata[j].rightdata[k];
  }
  x->leftmean += amean/x->window_size;
  x->rightmean += bmean/x->window_size;

  //this will be fft(left input)
  for(i=0;i<(x->window_size);i++){
    j = (x->list_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    x->fft_in[i]=x->listdata[j].leftdata[k]-amean/x->window_size;
    aa0 += x->fft_in[i]*x->fft_in[i]; //needed to compute autocrosscov at
        zero lag
  }
  fftw_execute(x->fft_plan);
  A = x->fft_out;
  for(i=0;i<x->outlength;i++){
    x->fftleft[i] += sqrt(creal(*(A+i))*creal(*(A+i)) + cimag(*(A+i))*cimag
        (*(A+i)));
    x->Astore[i]=*(x->fft_out+i);
  }

  //this will be conj(fft(right input)) for real input
  for(i=0;i<(x->window_size);i++){
    j = (x->list_ptr -  i)%BUFFER_SIZE;
    x->fft_in[i]=x->listdata[j].rightdata[k]-bmean/x->window_size;
    bb0 += x->fft_in[i]*x->fft_in[i]; //needed to compute autocrosscov at
        zero lag
  }
  fftw_execute(x->fft_plan);
  B = x->fft_out;
  for(i=0;i<x->outlength;i++){
    //magnitude of conj(fft) = mag(fft) for real input
    x->fftright[i] += sqrt(creal(*(B+i))*creal(*(B+i))+cimag(*(B+i))*cimag
        (*(B+i)));
    x->Bstore[i]=*(x->fft_out+i);
  }

  //compute autocrosscov at zero lag
  scale = sqrt(aa0*bb0);
  //compute cross covariance
  for(i=0;i<x->outlength;i++){
      x->ifft_in[i]=x->Astore[i]*x->Bstore[i];
    }
    fftw_execute(x->ifft_plan);
    for(i=0;i<x->window_size;i++){
      j=(i+(x->window_size>>1))%x->window_size;
      x->result[i]+=x->ifft_out[j]/(x->window_size*scale);
    }
  }
  x->leftmean = x->leftmean/x->listlength;
  x->rightmean = x->rightmean/x->listlength;
  x->peakval = x->peakval/x->listlength;
  x->lagtime = x->lagtime/x->listlength;
  for(i=0;i < x->outlength;i++){
    x->fftleft[i] = x->fftleft[i]/x->listlength;
    x->fftright[i] = x->fftright[i]/x->listlength;
  }
  for(i=0;i < x->window_size;i++){
    x->result[i] = x->result[i]/x->listlength;
    if(x->result[i]>=x->peakval){
      x->peakval=x->result[i];
      x->lagtime=i;
    }
  }
}

void *xcov_new(long window, long step, long listlen)
{
  int temp=2;
  t_xcov *x;
  //create new instance and return pointer to instance
  x = (t_xcov *)newobject(xcov_class);
  //set up proxy inlet
  x->r_proxy = proxy_new(x,1,&x->r_proxyinletnum);
  //initialize buffer offsets and flags
  x->w_ptr = 0;
  x->leftflag = 0;
  x->list_ptr = 0;
  x->listflag = 0;
  //set window, step, list input sizes
  if(window > BUFFER_SIZE){
    x->window_size = BUFFER_SIZE;
  }
  else{
    while(temp < window){
      temp=temp<<1;
    }
    x->window_size = temp;
  }
  post("using window size %ld",x->window_size);
  if(step > x->window_size){
    x->step_size = x->window_size;
  }
  else if(step == 0){
    x->step_size = 1;
  }
  else{
    x->step_size=(int)step;
  }
  post("using step size %ld",x->step_size);
  x->outlength=(x->window_size>>1)+1;
  if(x->outlength > MAX_OUT){
    x->outlength=MAX_OUT;
  }
  post("output will be size %ld",x->outlength);
  if(listlen > MAX_IN){
    x->listlength = MAX_IN;
  }
  else if(listlen == 0){
    x->listlength = 1;
  }
  else{
    x->listlength=(int)listlen;
  }
  post("list length will be %ld",x->listlength);

  //set up fftw data structure
  x->fft_in=(double*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
  x->fft_out=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
  x->fft_plan=fftw_plan_dft_r2c_1d(x->window_size,x->fft_in,x->fft_out,
       FFTW_ESTIMATE);
  x->ifft_in=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->window_size);
  x->ifft_out=(double*)fftw_malloc(sizeof(double)*x->window_size);
  x->ifft_plan=fftw_plan_dft_c2r_1d(x->window_size,x->ifft_in,x->ifft_out,
       FFTW_ESTIMATE);

  x->Astore=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->outlength);
  x->Bstore=(fftw_complex*)fftw_malloc(sizeof(fftw_complex)*x->outlength);

  //allocate other arrays
  x->leftdata=(long*)fftw_malloc(sizeof(long)*BUFFER_SIZE);
  x->rightdata=(long*)fftw_malloc(sizeof(long)*BUFFER_SIZE);
  x->fft_left_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
  x->fft_right_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
  x->result_list=(t_atom*)fftw_malloc(sizeof(t_atom)*MAX_OUT);
  x->fftleft=(double*)fftw_malloc(sizeof(double)*x->outlength);
  x->fftright=(double*)fftw_malloc(sizeof(double)*x->outlength);
  x->result=(double*)fftw_malloc(sizeof(double)*x->window_size);

  x->listdata=(t_list*)fftw_malloc(sizeof(t_list)*BUFFER_SIZE);

  //create outlets
  x->r_lagtime=intout(x);
  x->r_peakval=floatout(x);
  x->r_meanright=floatout(x);
  x->r_meanleft=floatout(x);
  x->r_fftright=listout(x);
  x->r_fftleft=listout(x);
  x->r_vector=listout(x);

  return x;
}


void xcov_free(t_xcov *x)
{
  //free the fftw data structure
  fftw_destroy_plan(x->fft_plan);
  fftw_destroy_plan(x->ifft_plan);
  fftw_free(x->fft_in);
  fftw_free(x->fft_out);
  fftw_free(x->ifft_in);
  fftw_free(x->ifft_out);
  fftw_free(x->Astore);
  fftw_free(x->Bstore);
  fftw_free(x->leftdata);
  fftw_free(x->rightdata);
  fftw_free(x->fft_left_list);
  fftw_free(x->fft_right_list);
  fftw_free(x->result_list);
  fftw_free(x->fftleft);
  fftw_free(x->fftright);
  fftw_free(x->result);
  fftw_free(x->listdata);
  freeobject(x->r_proxy);
}
```

Listing D.8: Source code for **Rmean** Max object.

```
#include "ext.h"

void *Rmean_class;

#define BUFFER_SIZE 4096

//data structure for Rmean
typedef struct {
  t_object r_ob;
  double mean;
  //buffer for incoming data
  long data[BUFFER_SIZE];
  long righttemp;
  unsigned int w_ptr;
  unsigned int flag;
  //settings
  int window_size;
  int step_size;
  //ports
  void *r_mean;
} t_Rmean;


void *Rmean_new(long window, long step);
void Rmean_free(t_Rmean *x);
void Rmean_int(t_Rmean *x, long val);
void Rmean_assist(t_Rmean *x, Object *b, long msg, long arg, char *s);
void compute_mean(t_Rmean *x);

int main()
{
  //setup creates class definition
  setup((t_messlist **)&Rmean_class, (method)Rmean_new, (method)Rmean_free, (
       short)sizeof(t_Rmean), 0L,A_DEFLONG,A_DEFLONG, 0);
  addint((method)Rmean_int);    //binds Rmean_int method to int message in
       left inlet
  addmess((method)Rmean_assist,"assist",A_CANT,0); //binds Rmean_assist
       method to assist message
  finder_addclass("All Objects","Rmean"); //adds class to object browser
  return (0);
}
```

```
void Rmean_int(t_Rmean *x, long val)
{
  x->data[x->w_ptr]=val;
  x->flag++;
  if(x->flag==x->step_size){
    compute_mean(x);
    outlet_float(x->r_mean,x->mean);
    x->flag=0;
  }
  x->w_ptr = (x->w_ptr + 1)%BUFFER_SIZE;
}


void Rmean_assist(t_Rmean *x, Object *b, long msg, long arg, char *s)
{
  if(msg == ASSIST_INLET){
    switch(arg){
      case 0: sprintf(s,"%s","Int inlet");
          break;
    }
  }
  else if(msg == ASSIST_OUTLET){
    switch(arg){
      case 0: sprintf(s,"%s","Mean of input with specified window");
          break;
      default:sprintf(s,"%s","No Designated Output");
    }
  }
}

void compute_mean(t_Rmean *x)
{
  int i,j;
  double amean=0;

  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    amean += x->data[j];
  }
  x->mean = amean/x->window_size;
}

void *Rmean_new(long window, long step)
{
  t_Rmean *x;
  //create new instance and return pointer to instance
  x = (t_Rmean *)newobject(Rmean_class);
  //initialize buffer offsets and flags
  x->w_ptr = 0;
  x->flag = 0;
  //set window and step size
  if(window > BUFFER_SIZE){
    x->window_size = BUFFER_SIZE;
  }
  else if(window < 2){
    x->window_size = 2;
  }
  else{
    x->window_size = window;
  }
  post("using window size %ld",x->window_size);
  if(step > x->window_size){
    x->step_size = x->window_size;
  }
  else if(step == 0){
    x->step_size=1;
  }
  else{
    x->step_size=(int)step;
  }
  post("using step size %ld",x->step_size);
  //create outlets
  x->r_mean=floatout(x);
  return x;
}

void Rmean_free(t_Rmean *x){
  return;
}
```

Listing D.9: Source code for `Rvar` Max object.

```
#include "ext.h"

void *Rvar_class;

#define BUFFER_SIZE 1024

//data structure for Rvar
typedef struct {
  t_object r_ob;
  double var;
  //buffer for incoming data
  double data[BUFFER_SIZE];
  long righttemp;
  unsigned int w_ptr;
  unsigned int flag;
  //settings
  int window_size;
  int step_size;
  //ports
  void *r_var;
} t_Rvar;


void *Rvar_new(long window, long step);
void Rvar_free(t_Rvar *x);
void Rvar_int(t_Rvar *x, long val);
void Rvar_float(t_Rvar *x, double val);
void Rvar_assist(t_Rvar *x, Object *b, long msg, long arg, char *s);
void compute_var(t_Rvar *x);

int main()
{
  //setup creates class definition
  setup((t_messlist **)&Rvar_class, (method)Rvar_new, (method)Rvar_free, (
       short)sizeof(t_Rvar), 0L,A_DEFLONG,A_DEFLONG, 0);
  addint((method)Rvar_int);   //binds Rvar_int method to int message in left
       inlet
  addfloat((method)Rvar_float);
  addmess((method)Rvar_assist,"assist",A_CANT,0); //binds Rvar_assist method
       to assist message
  finder_addclass("All Objects","Rvar"); //adds class to object browser
  return (0);
}

void Rvar_int(t_Rvar *x, long val)
{
  x->data[x->w_ptr]=(double)val;
  x->flag++;
  if(x->flag==x->step_size){
    compute_var(x);
    outlet_float(x->r_var,x->var);
    x->flag=0;
  }
  x->w_ptr = (x->w_ptr + 1)%BUFFER_SIZE;
}

void Rvar_float(t_Rvar *x, double val)
{
  x->data[x->w_ptr]=val;
  x->flag++;
  if(x->flag==x->step_size){
    compute_var(x);
    outlet_float(x->r_var,x->var);
    x->flag=0;
  }
  x->w_ptr = (x->w_ptr + 1)%BUFFER_SIZE;
}


void Rvar_assist(t_Rvar *x, Object *b, long msg, long arg, char *s)
{
  if(msg == ASSIST_INLET){
    switch(arg){
      case 0: sprintf(s,"%s","Int/Float inlet");
          break;
    }
  }
  else if(msg == ASSIST_OUTLET){
    switch(arg){
      case 0: sprintf(s,"%s","Variance of input with specified window");
          break;
      default:sprintf(s,"%s","No Designated Output");
    }
  }
}

void compute_var(t_Rvar *x)
{
  int i,j;
  double amean=0;
  double avar=0;

  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    amean += x->data[j];
  }
  amean = amean/x->window_size;
  for(i=0;i<(x->window_size);i++){
    j = (x->w_ptr - x->window_size + i + 1)%BUFFER_SIZE;
    avar += (x->data[j]-amean)*(x->data[j]-amean);
  }
  x->var = avar/x->window_size;
}

void *Rvar_new(long window, long step)
{
  t_Rvar *x;
  //create new instance and return pointer to instance
  x = (t_Rvar *)newobject(Rvar_class);
  //initialize buffer offsets and flags
  x->w_ptr = 0;
  x->flag = 0;
  //set window and step size
  if(window > BUFFER_SIZE){
    x->window_size = BUFFER_SIZE;
  }
  else if(window < 2){
    x->window_size = 2;
  }
  else{
    x->window_size = window;
  }
  post("using window size %ld",x->window_size);
  if(step > x->window_size){
    x->step_size = x->window_size;
  }
  else if(step == 0){
    x->step_size=1;
  }
  else{
    x->step_size=(int)step;
  }
  post("using step size %ld",x->step_size);


  //create outlets
  x->r_var=floatout(x);

  return x;
}

void Rvar_free(t_Rvar *x){
  return;
}
```

Appendix E

Max/MSP Patches

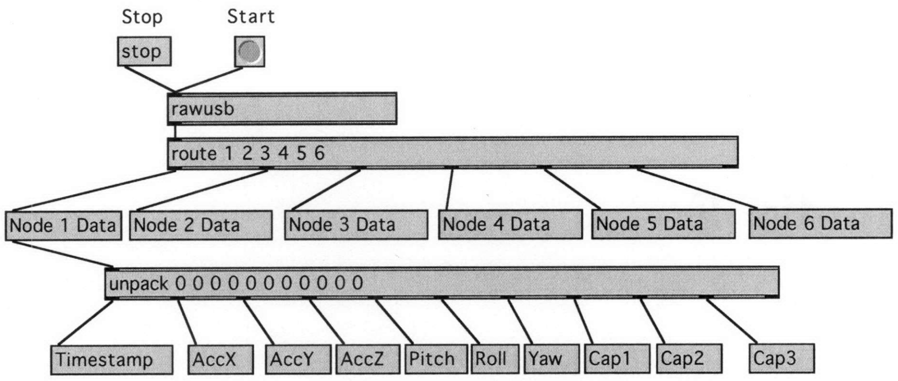


Figure E-1: Patch showing `rawusb` external used to extract data from the USB basestation.

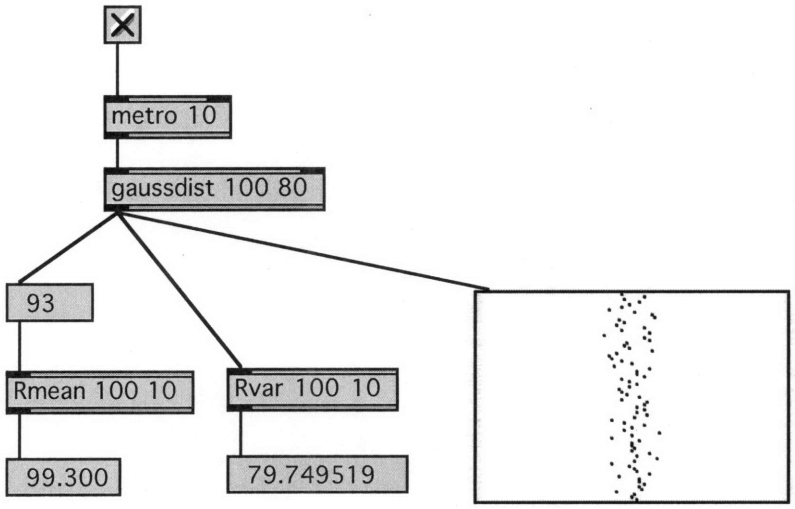


Figure E-2: Externals `Rmean` and `Rvar` usage.

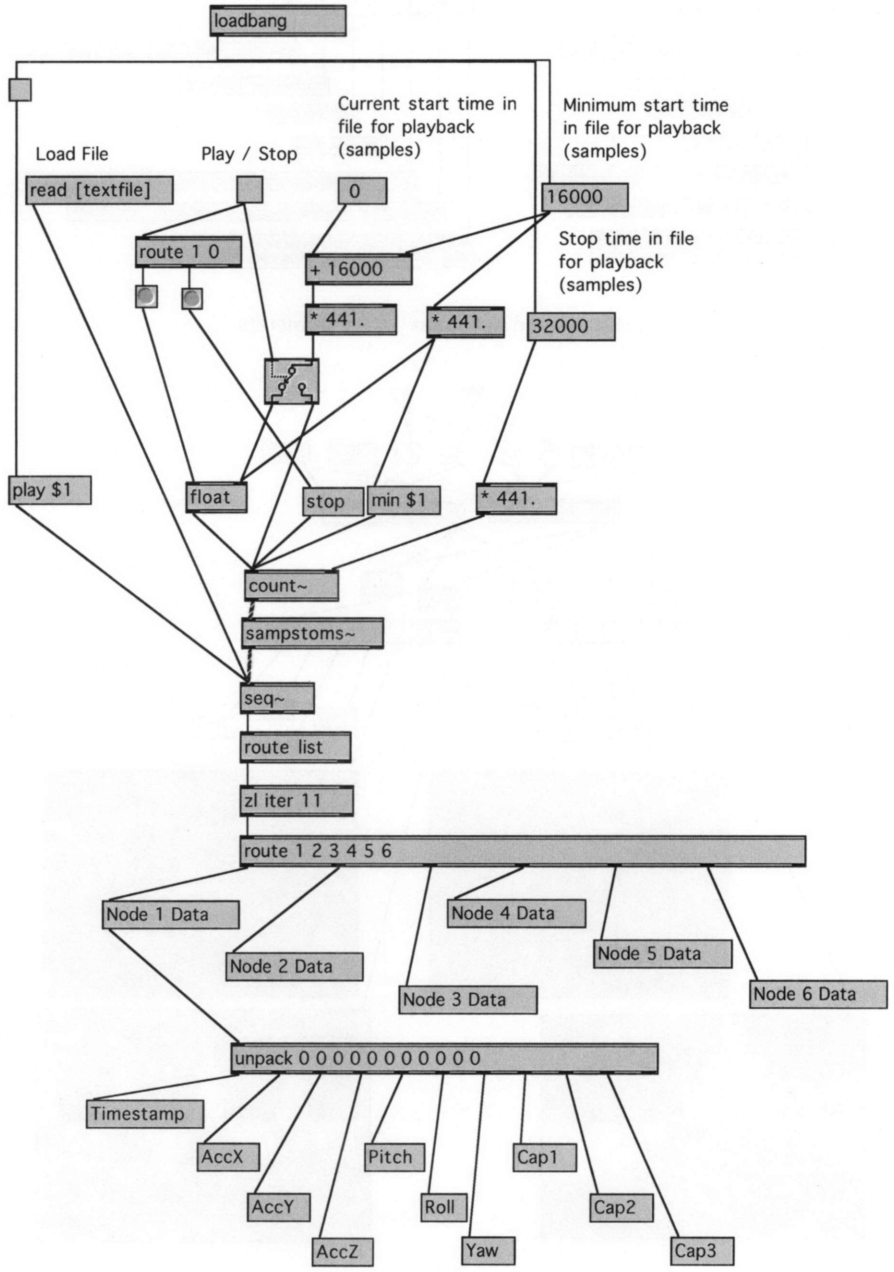
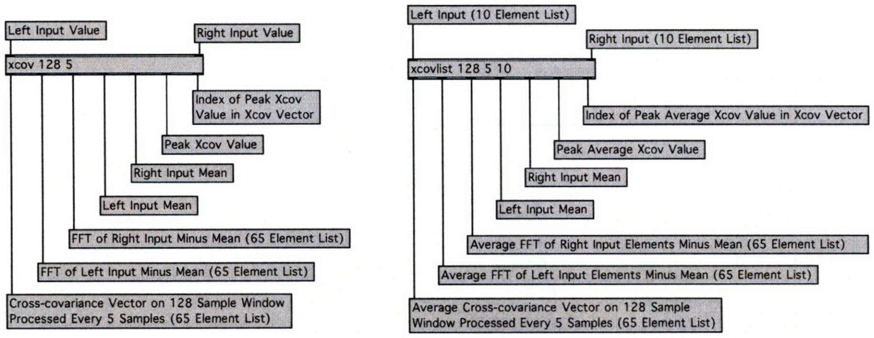
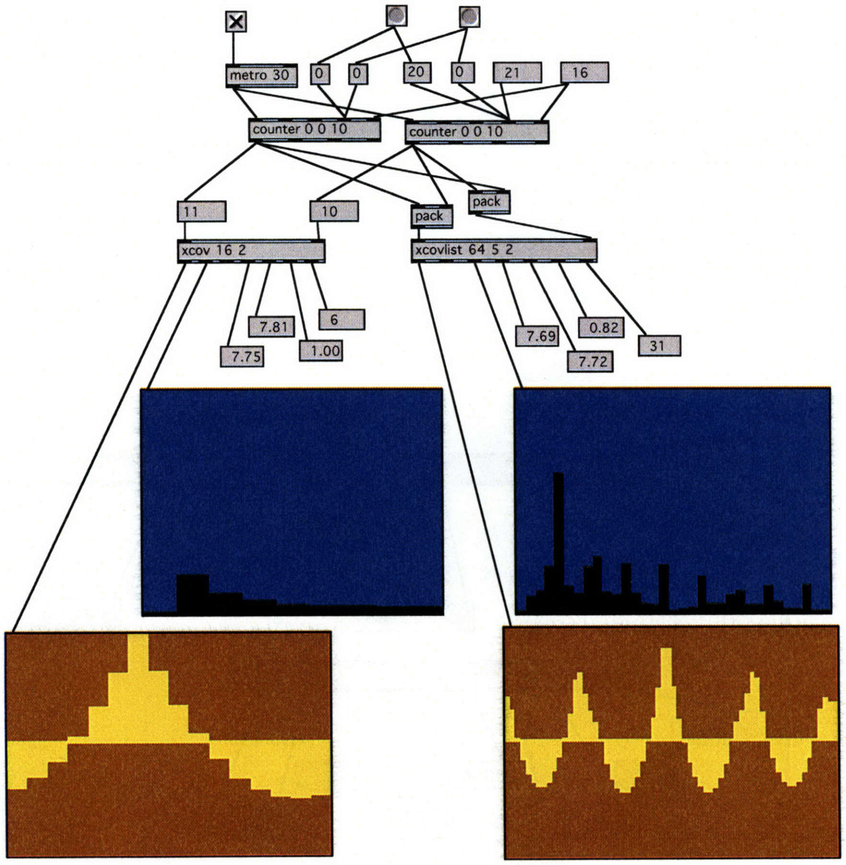


Figure E-3: Patch showing usage of seq~ for loading and playing back data from a text file.


Left Input Value
Right Input Value
xcov 128 5
Index of Peak Xcov Value in Xcov Vector
Peak Xcov Value
Right Input Mean
Left Input Mean
FFT of Right Input Minus Mean (65 Element List)
FFT of Left Input Minus Mean (65 Element List)
Cross-covariance Vector on 128 Sample Window Processed Every 5 Samples (65 Element List)
Left Input (10 Element List)
Right Input (10 Element List)
xcovlist 128 5 10
Index of Peak Average Xcov Value in Xcov Vector
Peak Average Xcov Value
Right Input Mean
Left Input Mean
Average FFT of Right Input Elements Minus Mean (65 Element List)
Average FFT of Left Input Elements Minus Mean (65 Element List)
Average Cross-covariance Vector on 128 Sample Window Processed Every 5 Samples (65 Element List)

(a) `xcov` and `xcovlist` I/O Parameters



(b) Example Patch

Figure E-4: Externals `xcov` and `xcovlist` usage.

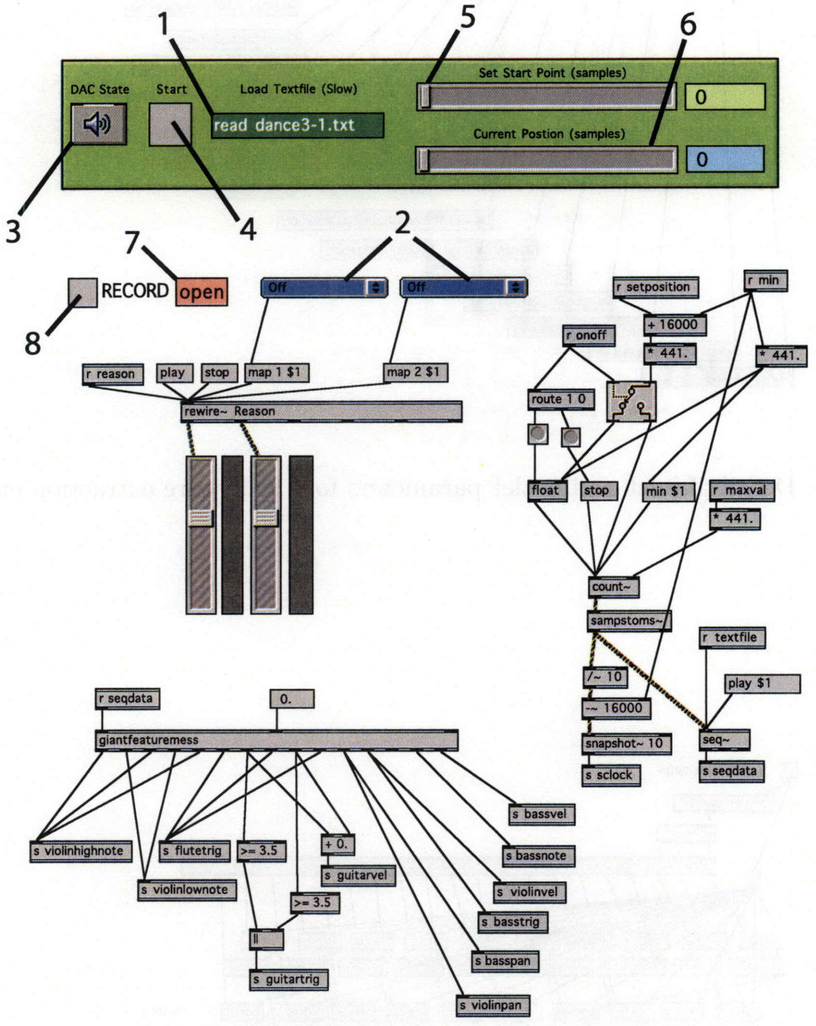


Figure E-5: Interface for streaming recorded text files Into Max/MSP.

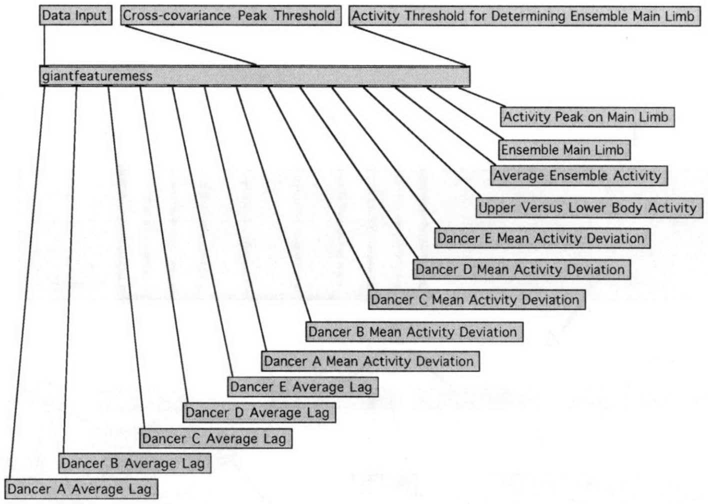

Data Input
Cross-covariance Peak Threshold
Activity Threshold for Determining Ensemble Main Limb
giantfeaturemess
Activity Peak on Main Limb
Ensemble Main Limb
Average Ensemble Activity
Upper Versus Lower Body Activity
Dancer E Mean Activity Deviation
Dancer D Mean Activity Deviation
Dancer C Mean Activity Deviation
Dancer B Mean Activity Deviation
Dancer A Mean Activity Deviation
Dancer E Average Lag
Dancer D Average Lag
Dancer C Average Lag
Dancer B Average Lag
Dancer A Average Lag

Figure E-6: Detail of inlet and outlet parameters for the feature extraction engine.

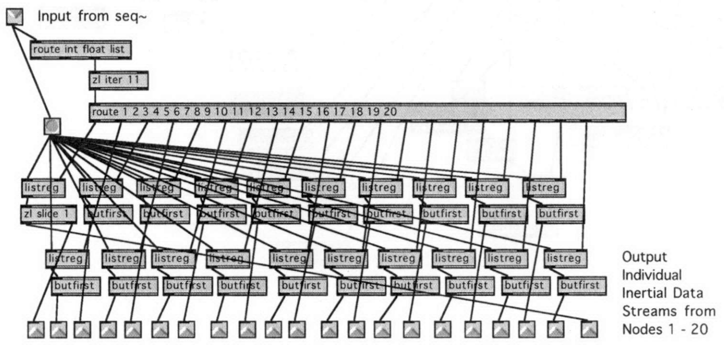
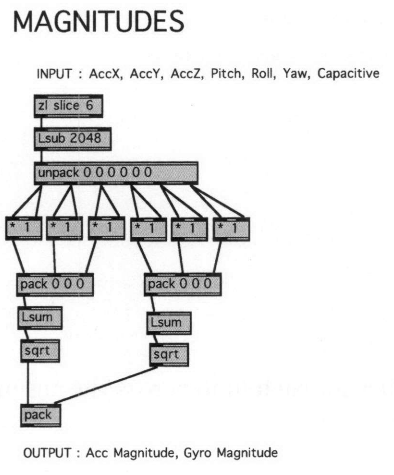
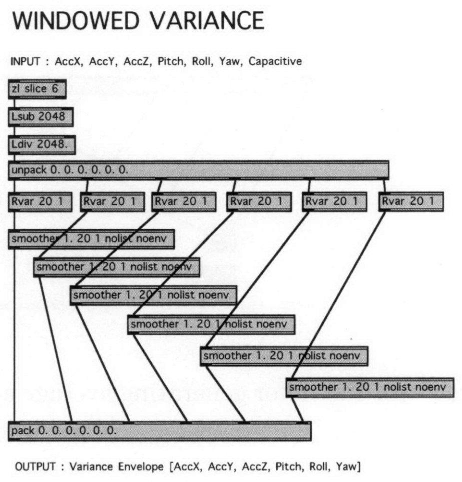

Input from seq~
route int float list
zl iter 11
route 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
listreg
zl slide 1
butfirst
listreg
butfirst
Output Individual Inertial Data Streams from Nodes 1 - 20

Figure E-7: Patch for parsing bulk data stream by node.

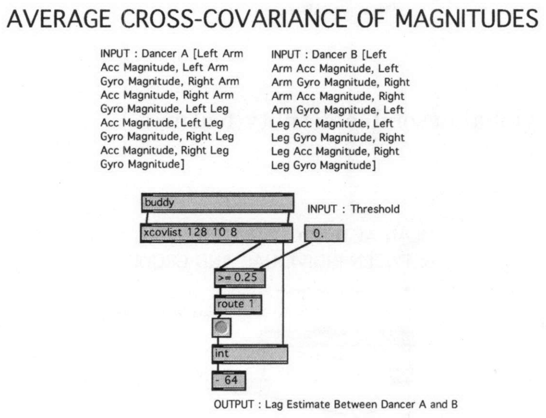


(a) Magnitude.

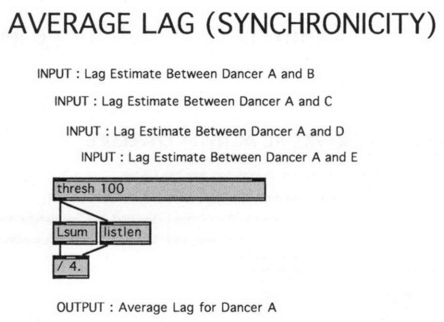

WINDOWED VARIANCE
INPUT : AccX, AccY, AccZ, Pitch, Roll, Yaw, Capacitive
zl slice 6
Lsub 2048
Ldiv 2048.
unpack 0. 0. 0. 0. 0. 0.
Rvar 20 1
Rvar 20 1
Rvar 20 1
Rvar 20 1
Rvar 20 1
Rvar 20 1
smoother 1. 20 1 nolist noenv
smoother 1. 20 1 nolist noenv
smoother 1. 20 1 nolist noenv
smoother 1. 20 1 nolist noenv
smoother 1. 20 1 nolist noenv
smoother 1. 20 1 nolist noenv
pack 0. 0. 0. 0. 0. 0.
OUTPUT : Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]

(b) Windowed variance.

Figure E-8: Example patches for extracting basic features in Max/MSP.


AVERAGE CROSS-COVARIANCE OF MAGNITUDES
INPUT : Dancer A [Left Arm Acc Magnitude, Left Arm Gyro Magnitude, Right Arm Acc Magnitude, Right Arm Gyro Magnitude, Left Leg Acc Magnitude, Left Leg Gyro Magnitude, Right Leg Acc Magnitude, Right Leg Gyro Magnitude]
INPUT : Dancer B [Left Arm Acc Magnitude, Left Arm Gyro Magnitude, Right Arm Acc Magnitude, Right Arm Gyro Magnitude, Left Leg Acc Magnitude, Left Leg Gyro Magnitude, Right Leg Acc Magnitude, Right Leg Gyro Magnitude]
buddy
INPUT : Threshold
xcovlist 128 10 8
0.
>= 0.25
route 1
int
- 64
OUTPUT : Lag Estimate Between Dancer A and B

(a) Computing a lag estimate between two dancers using cross-covariance.


AVERAGE LAG (SYNCHRONICITY)
INPUT : Lag Estimate Between Dancer A and B
INPUT : Lag Estimate Between Dancer A and C
INPUT : Lag Estimate Between Dancer A and D
INPUT : Lag Estimate Between Dancer A and E
thresh 100
Lsum
listlen
/ 4.
OUTPUT : Average Lag for Dancer A

(b) Accumulating and averaging lag estimates with respect to all dancers.

Figure E-9: Example patches for describing synchronicity.

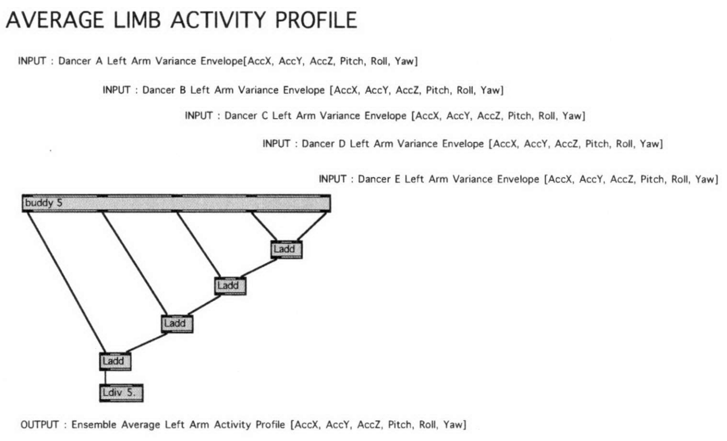

AVERAGE LIMB ACTIVITY PROFILE
INPUT : Dancer A Left Arm Variance Envelope[AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer B Left Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer C Left Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer D Left Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer E Left Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
buddy 5
Ladd
Ladd
Ladd
Ladd
Ldiv 5.
OUTPUT : Ensemble Average Left Arm Activity Profile [AccX, AccY, AccZ, Pitch, Roll, Yaw]

Figure E-10: Patch for generating average activity profiles for each limb across the ensemble.

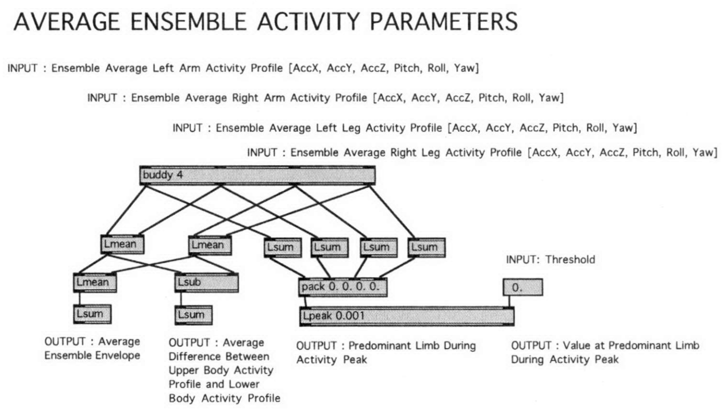
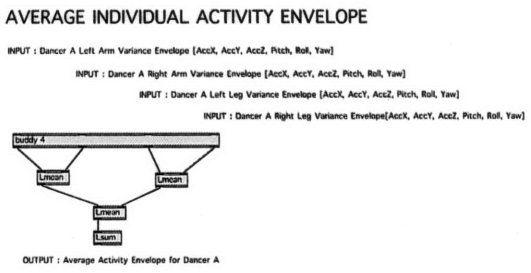
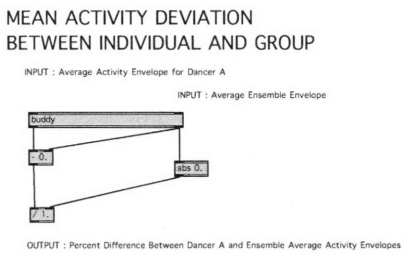

AVERAGE ENSEMBLE ACTIVITY PARAMETERS
INPUT : Ensemble Average Left Arm Activity Profile [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Ensemble Average Right Arm Activity Profile [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Ensemble Average Left Leg Activity Profile [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Ensemble Average Right Leg Activity Profile [AccX, AccY, AccZ, Pitch, Roll, Yaw]
buddy 4
Lmean
Lmean
Lsum
Lsum
Lsum
Lsum
INPUT: Threshold
Lmean
Lsub
pack 0. 0. 0. 0.
0.
Lsum
Lsum
Lpeak 0.001
OUTPUT : Average Ensemble Envelope
OUTPUT : Average Difference Between Upper Body Activity Profile and Lower Body Activity Profile
OUTPUT : Predominant Limb During Activity Peak
OUTPUT : Value at Predominant Limb During Activity Peak

Figure E-11: Patch for generating global ensemble activity features.


AVERAGE INDIVIDUAL ACTIVITY ENVELOPE
INPUT : Dancer A Left Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer A Right Arm Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer A Left Leg Variance Envelope [AccX, AccY, AccZ, Pitch, Roll, Yaw]
INPUT : Dancer A Right Leg Variance Envelope[AccX, AccY, AccZ, Pitch, Roll, Yaw]
buddy 4
Lmean
Lmean
Lmean
Lsum
OUTPUT : Average Activity Envelope for Dancer A

(a) Computing an individual activity level.


MEAN ACTIVITY DEVIATION
BETWEEN INDIVIDUAL AND GROUP
INPUT : Average Activity Envelope for Dancer A
INPUT : Average Ensemble Envelope
buddy
- 0.
abs 0.
/ 1.
OUTPUT : Percent Difference Between Dancer A and Ensemble Average Activity Envelopes

(b) Computing the percent difference.

Figure E-12: Comparing individual activity to group activity.

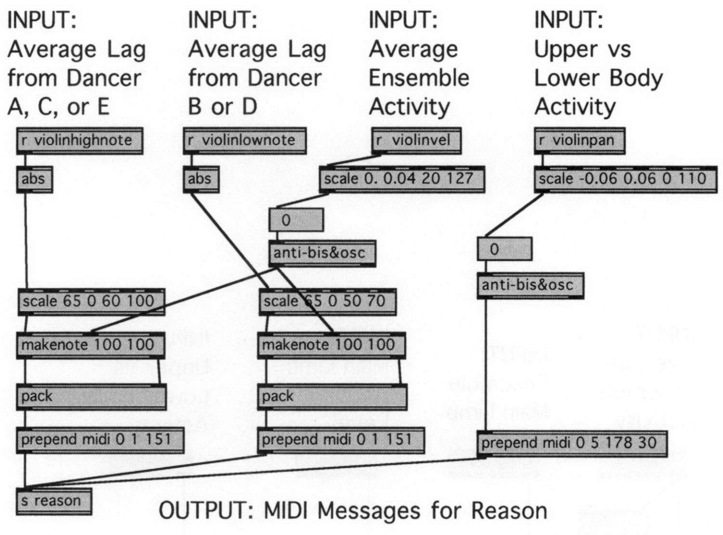
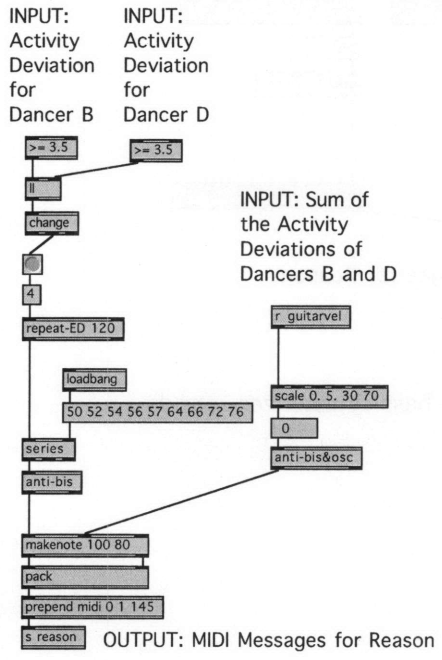
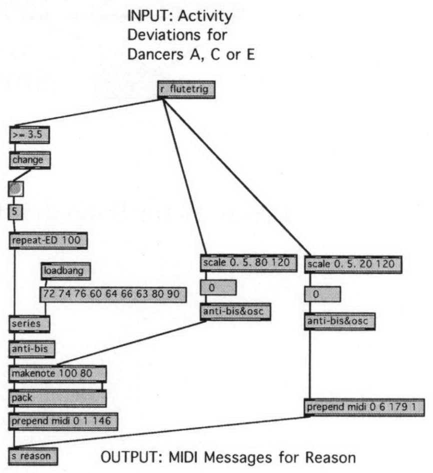


Figure E-13: Subpatch for controlling violin sounds.



(a) Guitar.



(b) Flute.

Figure E-14: Subpatches for controlling guitar and flute sounds.

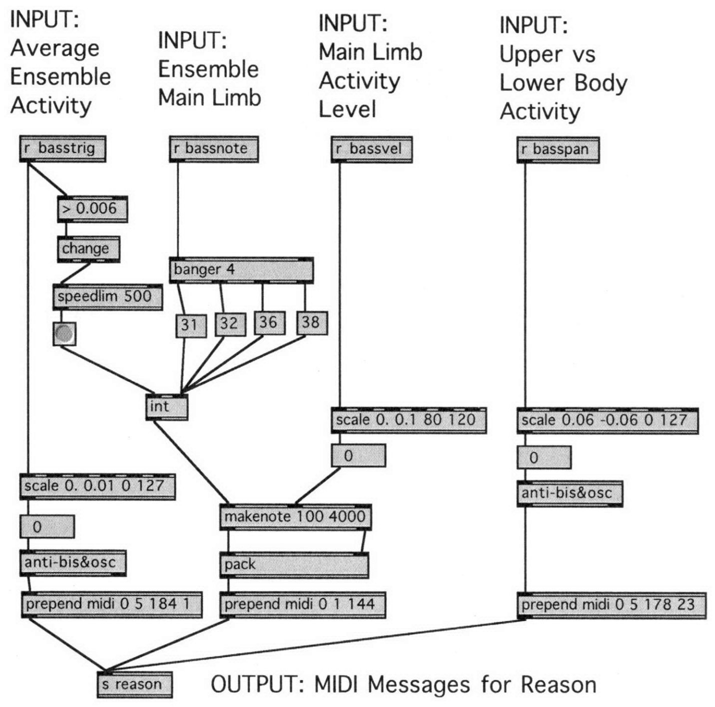


Figure E-15: Subpatch for controlling bass synthesizer sounds.

Appendix F

Matlab Scripts

Listing F.1: Importing recorded data from a text file into Matlab and repairing timing discrepancies.

```
%Parse USB text file generated by sensemble data logging app
clear all
close all

maxnumnodes = 25;

fid = fopen('data.txt');
RawData = fscanf(fid,'%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d'
      ,[13,inf]);
fclose(fid);

%Temporal correction
hostinit = RawData(2,1); %host starting timestamp
baseinit = RawData(3,1); %basestation starting timestamp
timetotal = round((max(RawData(2,:))-hostinit)/10); %total time processing on
      host in units of 10ms, assuming the text file was logged correctly
unrolledtime = RawData(3,:); %initialize array for unrolled basestation time
gtime = round((RawData(2,:)-hostinit)./10); %host time in intervals of 10ms
for i = 1:length(RawData)-1,
    d = unrolledtime(i+1)-unrolledtime(i);
    if d<0,
        unrolledtime(i+1:length(RawData))=unrolledtime(i+1:length(RawData))
             +256;
    end
end
unrolledtime=unrolledtime-baseinit;
[U,V]=unique(unrolledtime);
hostskew = gtime(V) - U;
newtime = unrolledtime;
gapsizes = diff(hostskew);
gaplocations = find(gapsizes > 3);
if length(gaplocations) > 1
    for j = 1:length(gaplocations),
        newtime(V(gaplocations(j))+1:length(newtime))=newtime(V(gaplocations(
             j))+1:length(newtime)) + gapsizes(gaplocations(j));
    end
end
newtime=newtime+1; %newtime started at zero previously

%Create time corrected dataset with interpolation over missing samples
Data = {};
for i = 1:maxnumnodes
    index = find(RawData(1,:)==i);
    Data{i}(newtime(index),:)=RawData(4:9,index)';
    if length(index) > 0
        for j = 1:length(index)-1,
            x=newtime(index(j));
            y=newtime(index(j+1));
            if y - x > 1
                Data{i}(x+1:y-1,:)=ones(y-x-1,1)*Data{i}(x,:);
            end
        end
    end
end
```

Listing F.2: Assembling data into a seq~ format for streaming into Max/MSP.

```
%reformat text file for Max/MSP seq~ object
%run newUSBparse on desired data first, or import the workspace

start=8000;
stop=32000;
foo=fopen('data.txt','w');
[seqtimes,indices,mapping]=unique(newtime);
seqtimes=(seqtimes-1)*10;
fprintf(foo,'0, id 0;\n');
for i = start:stop,
    selection=find(mapping==i);
    fprintf(foo,'%d, %f ',i,seqtimes(i));
    for j = 1:length(selection),
        thisline=[RawData(1,selection(j)); RawData(3:12,selection(j))];
        fprintf(foo,'%d %d %d %d %d %d %d %d %d %d %d %d ',thisline);
    end
    fprintf(foo,';\n');
end
fclose(foo);
```

Listing F.3: Running average cross-covariance.

```
function [tx,ty,C] = runxcov(Xdata,Ydata,window,step,fs)
%Calculates the running cross-covariance of two data segments whose time
%vector is row aligned. The data segments should have the same size.
%If Xdata and Ydata contain more than one column,the running xcovar of
%each column is summed to obtain the output. The function returns scaled
%axis vectors with units in seconds given sampling rate fs.

[xrows xcols]=size(Xdata);
[yrows ycols]=size(Ydata);

if xrows ~= yrows | xcols ~= ycols
    error('Inputs must be the same size')
end

tx = [0:ceil((xrows-window)/step)-1]*step/fs;
ty = [-window:window]/fs;

i=1;
for n = 1:step:xrows-window,
    for thiscolumn = 1:xcols,
        temp(:,thiscolumn)=xcov(Xdata(n:n+window,thiscolumn),Ydata(n:n+window
             ,thiscolumn),'coeff');
    end
    C(:,i)=sum(temp,2);
    i=i+1;
end
```

Listing F.4: Collection of scripts for feature extraction.

```
%Here you must import block 1
close all

Rmean={};
Rvar={};

start = 92000;
stop = 97000;
window = 100;
step = 25;
%load data segment,center with ideal center value,scale by ideal max
c = 2048;
for thisnode = 1:6,
    NormData{thisnode} = (Node{thisnode}(start:stop-1,:) - c)/c;
end
window=50;
step=10;

%plots the x accel and roll gyro of three legs during quick synchronous leg
     lifts
figure(1)
subplot 311
plot([0:length(NormData{2})-1]/100,NormData{2}(:,[1 5]))
subplot 312
plot([0:length(NormData{3})-1]/100,NormData{3}(:,[1 5]))
subplot 313
plot([0:length(NormData{5})-1]/100,NormData{5}(:,[1 5]))
xlabel('Elapsed Time (Seconds)')
%plots the x accel and roll gyro of the arms during the leg lifts
figure(2)
subplot 311
plot([0:length(NormData{2})-1]/100,NormData{4}(:,[1 5]))
subplot 312
plot([0:length(NormData{3})-1]/100,NormData{6}(:,[1 5]))
subplot 313
plot([0:length(NormData{5})-1]/100,NormData{1}(:,[1 5]))
xlabel('Elapsed Time (Seconds)')

%calculate running cross-covariances between legs and then between arms
[tx,ty,BuffyAshleyLeg]=runxcov(NormData{2},NormData{3},window,step,100);
figure(3)
subplot 311
imagesc(tx,ty,BuffyAshleyLeg)
[tx,ty,BuffySarahLeg]=runxcov(NormData{2},NormData{5},window,step,100);
subplot 312
imagesc(tx,ty,BuffySarahLeg)
[tx,ty,AshleySarahLeg]=runxcov(NormData{3},NormData{5},window,step,100);
subplot 313
imagesc(tx,ty,AshleySarahLeg)
xlabel('Elapsed Time (Seconds)')
[tx,ty,BuffyAshleyArm]=runxcov(NormData{4},NormData{1},window,step,100);
figure(4)
subplot 311
imagesc(tx,ty,BuffyAshleyArm)
[tx,ty,BuffySarahArm]=runxcov(NormData{4},NormData{6},window,step,100);
subplot 312
imagesc(tx,ty,BuffySarahArm)
[tx,ty,AshleySarahArm]=runxcov(NormData{1},NormData{6},window,step,100);
subplot 313
imagesc(tx,ty,AshleySarahArm)
xlabel('Elapsed Time (Seconds)')

%short scale running mean and variance
for thisnode = 1:6,
    i=1;
    for n = 1:step:length(NormData{thisnode})-window
        Rmean{thisnode}(i,:)=mean(NormData{thisnode}(n:n+window,:));
        Rvar{thisnode}(i,:)=var(NormData{thisnode}(n:n+window,:));
        i=i+1;
    end
end

figure(5)
subplot 311
plot([0:length(NormData{5})-1]/100,NormData{5}(:,5))
subplot 312
plot([0:length(NormData{3})-1]/100,NormData{1}(:,5))
subplot 313
hold on
plot([0:step:length(NormData{5})-window-1]/100,sum(medfilt1(Rvar{5}(:,1:3)
     ,20),2))
plot([0:step:length(NormData{5})-window-1]/100,sum(medfilt1(Rvar{5}(:,4:6)
     ,20),2),'r')
plot([0:step:length(NormData{2})-window-1]/100,sum(medfilt1(Rvar{1}(:,1:3)
     ,20),2),'g')
plot([0:step:length(NormData{2})-window-1]/100,sum(medfilt1(Rvar{1}(:,4:6)
     ,20),2),'k')
hold off

start = 115000;
stop = 124000;
c = 2048;
for thisnode = 1:6,
    NormData{thisnode} = (Node{thisnode}(start:stop-1,:) - c)/c;
end
window=50;
step=10;
%Running mean and variance
for thisnode = 1:6,
    i=1;
    for n = 1:step:length(NormData{thisnode})-window
        Rmean{thisnode}(i,:)=mean(NormData{thisnode}(n:n+window,:));
        Rvar{thisnode}(i,:)=var(NormData{thisnode}(n:n+window,:));
        i=i+1;
    end
end

figure(6)
subplot 311
hold on
plot([0:step:length(NormData{2})-window-1]/100,sum(medfilt1(Rvar{2}(:,1:3)
     ,20),2))
```

```
plot([0:step:length(NormData{2})-window-1]/100,sum(medfilt1(Rvar{2}(:,4:6)
      ,20),2),'r')
hold off
subplot 312
hold on
plot([0:step:length(NormData{3})-window-1]/100,sum(medfilt1(Rvar{3}(:,1:3)
      ,20),2))
plot([0:step:length(NormData{3})-window-1]/100,sum(medfilt1(Rvar{3}(:,4:6)
      ,20),2),'r')
hold off
subplot 313
hold on
plot([0:step:length(NormData{5})-window-1]/100,sum(medfilt1(Rvar{5}(:,1:3)
      ,20),2))
plot([0:step:length(NormData{5})-window-1]/100,sum(medfilt1(Rvar{5}(:,4:6)
      ,20),2),'r')
hold off

%axes of predominant activity
Activity = {};
for thisnode = 1:6,
    Sum = sum(Rvar{thisnode},2);
    flag = 0;
    for i = 1:length(Sum),
        if flag == 0
            if i == length(Sum)
                Activity{thisnode}(i,1:2)=0;
            elseif Sum(i)>=0.03
                index=i;
                flag = 1;
            end
        else
            if Sum(i)<=0.01 | i==length(Sum)
                [maxval maxidx]=max(max(Rvar{thisnode}(index:i,4:6)));
                Activity{thisnode}(index:i,1)=maxidx;
                [maxval maxidx]=max(max(Rvar{thisnode}(index:i,1:3)));
                Activity{thisnode}(index:i,2)=maxidx;
                flag = 0;
            end
        end
    end
end
```

Listing F.5: Analysis for results section.

```
% Figures for results section of thesis

%run newUSBdataparse first to load desired data block
close all
clear C;
clear Lag;

Suelin = [1 11 15 5];
Jessica = [12 6 17 2];
Allison = [19 4 7 16];
Vala = [10 18 9 13];
Adie = [3 14 20 8];
RWrist = [1 12 19 10 3];
LWrist = [11 6 4 18 14];
RLeg = [15 17 7 9 20];
LLeg = [5 2 16 13 8];
Uppers = [RWrist LWrist];
Lowers = [RLeg LLeg];
People = [Suelin;Jessica;Allison;Vala;Adie];
Limbs = [RWrist; LWrist; RLeg; LLeg];
center = 2048;

%segment 3:18 to 3:40
start = 19800;
stop = 22000;
Markers = [1 100 367 400 500 700 750 800 867 973 1043 1200 1327 1430 1573
      1700 1867 2000 2167 2200];

% %segment 4:52 to 5:13
% start = 29200;
% stop = 31300;
% Markers = [1 40 100 140 170 200 263 293 440 777 880 1117 1230 1493 2060
      2100];

seg = [1:stop-start+1];
segment =[start:stop];
time = [0:length(segment)-1]/100;

%Video Frame Markers
Mark = zeros(length(time),1)-127;
Mark(Markers) = 127;

%Calculate Basic Features
window=20;
step=1;
NodeVar=zeros(length(time),6,20);
NodeMag=zeros(length(segment),2,20);
NodeActivity=zeros(length(time),6,20);
PersonMag=zeros(length(segment),8,5);
FullActivity=zeros(length(time),6,4,5);
LimbActivity=zeros(length(time),6,4);
PersonActivity=zeros(length(time),6,5);

for n = 1:20,
    NodeMag(:,1,n) = sqrt(sum(((Data{n}(start:stop,1:3)-center)/center).^2,2)
        );
    NodeMag(:,2,n) = sqrt(sum(((Data{n}(start:stop,4:6)-center)/center).^2,2)
        );
    for i = 1+window:length(time),
        j=(i-1)*step+start;
        NodeVar(i,:,n)=var((Data{n}(j-window:j,:)-center)/center);
    end
    NodeActivity(:,:,n)=medfilt1(NodeVar(:,:,n),20);
end

%Calculate Ensemble Features
FullActivity = cat(4,NodeActivity(:,:,Limbs(:,1)),NodeActivity(:,:,Limbs(:,2)
      ),NodeActivity(:,:,Limbs(:,3)),NodeActivity(:,:,Limbs(:,4)),NodeActivity
      (:,:,Limbs(:,5)));
LimbActivity = mean(FullActivity,4);
EnsembleActivity = mean(mean(LimbActivity,3),2);
Upper = mean(LimbActivity(:,:,[1 2]),3);
Lower = mean(LimbActivity(:,:,[3 4]),3);
```

```
UpperVersusLower = mean((Upper - Lower),2);
[LimbPeakValue MainLimb] = max(mean(LimbActivity,2),[],3);
LimbMask=zeros(length(LimbPeakValue),1);
LimbMask(find(LimbPeakValue > 0.001))=1;
MainLimb=MainLimb.*LimbMask;

%Personal - Average Lag Time
window=128;
step=10;
C={};
Lag={};
PersonMag=[NodeMag(:,:,People(:,1)),NodeMag(:,:,People(:,2)),NodeMag(:,:,
     People(:,3)),NodeMag(:,:,People(:,4))];
[tx,ty,C{1}]=runxcov(PersonMag(:,:,1),PersonMag(:,:,2),window,step,100);
[tx,ty,C{2}]=runxcov(PersonMag(:,:,1),PersonMag(:,:,3),window,step,100);
[tx,ty,C{3}]=runxcov(PersonMag(:,:,1),PersonMag(:,:,4),window,step,100);
[tx,ty,C{4}]=runxcov(PersonMag(:,:,1),PersonMag(:,:,5),window,step,100);
[tx,ty,C{5}]=runxcov(PersonMag(:,:,2),PersonMag(:,:,3),window,step,100);
[tx,ty,C{6}]=runxcov(PersonMag(:,:,2),PersonMag(:,:,4),window,step,100);
[tx,ty,C{7}]=runxcov(PersonMag(:,:,2),PersonMag(:,:,5),window,step,100);
[tx,ty,C{8}]=runxcov(PersonMag(:,:,3),PersonMag(:,:,4),window,step,100);
[tx,ty,C{9}]=runxcov(PersonMag(:,:,3),PersonMag(:,:,5),window,step,100);
[tx,ty,C{10}]=runxcov(PersonMag(:,:,4),PersonMag(:,:,5),window,step,100);


Peaks = zeros(length(tx),10);
RIndices = zeros(length(tx),10);
Indices = zeros(length(tx),10);
Mask = zeros(length(tx),10);
for n = 1:10,
    [Peaks(:,n) RIndices(:,n)] = max(C{n}',[],2);
    RIndices(:,n)= RIndices(:,n) - 128;
    Mask(find(Peaks(:,n) > 2),n)=1;
    Indices(:,n)=RIndices(:,n).*Mask(:,n);
end

LagMat = zeros(length(ty),length(tx),5);
%Person 1
AccumMask=sum(Mask(:,[1 2 3 4]),2);
Valid = find(AccumMask > 0);
Lag{1}(:,1)=abs(sum(Indices(Valid,[1 2 3 4]),2)./AccumMask(Valid));
Lag{1}(:,2)=Valid;
RLag=round(mean(RIndices(:,[1 2 3 4]),2))+128;
RPeak=mean(Peaks(:,[1 2 3 4]),2);
for i = 1:length(tx),
    LagMat(RLag(i),i,1)=RPeak(i);
end
%Person 2
AccumMask=sum(Mask(:,[1 5 6 7]),2);
Valid = find(AccumMask > 0);
Lag{2}(:,1)=abs(sum( [Indices(Valid,[5 6 7]) -Indices(Valid,1)] ,2)./
     AccumMask(Valid));
Lag{2}(:,2)=Valid;
RLag=round(mean([RIndices(:,[5 6 7]) -RIndices(:,1)],2))+128;
RPeak=mean(Peaks(:,[1 5 6 7]),2);
for i = 1:length(tx),
    LagMat(RLag(i),i,2)=RPeak(i);
end
%Person 3
AccumMask=sum(Mask(:,[2 5 8 9]),2);
Valid = find(AccumMask > 0);
Lag{3}(:,1)=abs(sum( [Indices(Valid,[8 9])  -Indices(Valid,[2 5])] ,2)./
     AccumMask(Valid));
Lag{3}(:,2)=Valid;
RLag=round(mean([RIndices(:,[8 9]) -RIndices(:,[2 5])],2))+128;
RPeak=mean(Peaks(:,[2 5 8 9]),2);
for i = 1:length(tx),
    LagMat(RLag(i),i,3)=RPeak(i);
end
%Person 4
AccumMask=sum(Mask(:,[3 6 8 10]),2);
Valid = find(AccumMask > 0);
Lag{4}(:,1)=abs(sum( [Indices(Valid,10)  -Indices(Valid,[3 6 8])] ,2)./
     AccumMask(Valid));
Lag{4}(:,2)=Valid;
RLag=round(mean([RIndices(:,10) -RIndices(:,[3 6 8])],2))+128;
RPeak=mean(Peaks(:,[3 6 8 10]),2);
for i = 1:length(tx),
    LagMat(RLag(i),i,4)=RPeak(i);
end
%Person 5
AccumMask=sum(Mask(:,[4 7 9 10]),2);
Valid = find(AccumMask > 0);
Lag{5}(:,1)=abs(sum( -Indices(Valid,[4 7 9 10])  ,2)./AccumMask(Valid));
Lag{5}(:,2)=Valid;
RLag=round(mean(-RIndices(:,[4 7 9 10]),2))+128;
RPeak=mean(Peaks(:,[4 7 9 10]),2);
for i = 1:length(tx),
    LagMat(RLag(i),i,5)=RPeak(i);
end

%Personal - Average Deviation
PersonActivity=squeeze(mean(FullActivity,3));
Deviation = zeros(length(time),5);
for n = 1:5,
    Deviation(:,n)= (mean(PersonActivity(:,:,n),2) - EnsembleActivity)./
        EnsembleActivity;
end

%Violin
%Scale Lags to MIDI range (high violin notes and low violin notes)
for n=[1 3 5],
    Lag{n}(:,1)=round(-0.4615*Lag{n}(:,1)+100);
end
for n=[2 4],
    Lag{n}(:,1)=round(-0.3077*Lag{n}(:,1)+70);
end

%Scale Ensemble Activity to MIDI range for violin velocity
ViolinVel = round(2675*EnsembleActivity + 20);
%Scale Upper Versus Lower for Violin Pan
ViolinPan = round(5200*UpperVersusLower)+64;

%Flute
%Compute Triggers
FluteMask = zeros(length(time),1);
FluteMask(union(union(find(Deviation(:,1) >= 3.5),find(Deviation(:,3) >= 3.5)
     ),find(Deviation(:,5) >= 3.5)))=1;
FluteInput = max([Deviation(:,1)'; Deviation(:,3)';Deviation(:,5)']);
Flute = zeros(length(time),1);
Flute(find(diff(FluteMask)>0)+1)=1;
%Velocity
```

```
FluteVel = round(8*mean(Deviation(:,[1 3 5]),2) + 80);
%Mod Wheel
FluteMod = round(20*mean(Deviation(:,[1 3 5]),2) + 20);

%Guitar
%Compute Triggers
GuitarMask = zeros(length(time),1);
GuitarMask(union(find(Deviation(:,2) >= 3.5),find(Deviation(:,4) >= 3.5)))=1;
GuitarInput = max([Deviation(:,2)'; Deviation(:,4)']);
Guitar = zeros(length(time),1);
Guitar(find(diff(GuitarMask)>0)+1)=1;
%Velocity
GuitarVel = round(8*mean(Deviation(:,[2 4]),2) + 30);

%Bass
%Compute Triggers and Notes
BassNotes = [0 31 32 36 38];

BassMask = zeros(length(time),1);
BassMask(find(EnsembleActivity > 0.006))=1;
Bass = zeros(length(time),1);
SparseBass = zeros(length(time),1);
Bass(find(diff(BassMask)>0)+1)=1;
for i=[51:50:length(time)],
    if mean(Bass([i-50:i])) > 0
        SparseBass(i)=BassNotes(MainLimb(i)+1);
    end
end
%Mod Wheel
BassMod = round(12700*EnsembleActivity);
%Velocity
BassVel = round(400*LimbPeakValue);
%Pan
BassPan = round(-1058.3*UpperVersusLower + 0.06)+64;



figure(1)
set(gcf,'Renderer','zbuffer')
hold on
plot((Lag{1}(:,2)+12.8)*(step/100),Lag{1}(:,1),'b.')
plot((Lag{2}(:,2)+12.8)*(step/100),Lag{2}(:,1),'g.')
plot((Lag{3}(:,2)+12.8)*(step/100),Lag{3}(:,1),'r.')
plot((Lag{4}(:,2)+12.8)*(step/100),Lag{4}(:,1),'k.')
plot((Lag{5}(:,2)+12.8)*(step/100),Lag{5}(:,1),'m.')
plot(time,ViolinVel)
plot(time,ViolinPan,'r')
%plot(time(seg),Mark(seg),'k')
hold off

figure(2)
set(gcf,'Renderer','zbuffer')
subplot(211)
hold on
plot(time,EnsembleActivity)
%plot(time(seg),Mark(seg)*0.001,'k')
title('Ensemble Activity Envelope')
hold off
subplot(212)
hold on
plot(time,UpperVersusLower,'r')
%plot(time(seg),Mark(seg)*0.001,'k')
xlabel('Time Elapsed'),title('Upper Versus Lower Activity')
hold off

figure(3)
subplot(211)
imagesc(tx,ty,sum(LagMat(:,:,[1 3 5]),3))
title('Violin High')
subplot(212)
imagesc(tx,ty,sum(LagMat(:,:,[2 4]),3))
xlabel('Time Elapsed (Seconds)'),title('Violin Low')

figure(8)
subplot(211)
hold on
plot(time,GuitarInput)
%plot(time(seg),Mark(seg),'k')
title('Input to Guitar and Flute Triggers')
hold off
subplot(212)
hold on
plot(time,FluteInput,'r')
%plot(time(seg),Mark(seg),'k')
xlabel('Time Elapsed (Seconds)')
hold off

figure(9)
set(gcf,'Renderer','zbuffer')
hold on
plot(time,GuitarVel)
plot(time,FluteVel,'r')
plot(time,FluteMod,'g')
plot(time(find(Guitar>0)),Guitar(find(Guitar>0))*50,'k.')
plot(time(find(Flute>0)),Flute(find(Flute>0))*50,'c.')
%plot(time(seg),Mark(seg),'k')
hold off
title('Guitar and Flute Tracks'),xlabel('Time Elapsed (Seconds)')

%plot ensemble activity and personal activity
figure(10)
set(gcf,'Renderer','zbuffer')
subplot(311)
hold on
plot(time,EnsembleActivity,'k');
%plot(time(seg),Mark(seg),'k')
title('Ensemble')
hold off
subplot(312)
hold on
plot(time,squeeze(mean(PersonActivity(:,:,[1 3 5]),2)));
%plot(time(seg),Mark(seg),'k')
ylabel('Windowed Variance Envelope'),title('Dancers Driving Flute Track')
hold off
subplot(313)
hold on
plot(time,squeeze(mean(PersonActivity(:,:,[2 4]),2)));
%plot(time(seg),Mark(seg),'k')
xlabel('Time Elapsed (Seconds)'),title('Dancers Driving Guitar Track')
hold off

figure(11)
```

```
set(gcf,'Renderer','zbuffer')
hold on
plot(time(seg),BassPan(seg))
plot(time(seg),BassVel(seg),'r')
plot(time(seg),BassMod(seg),'g')
```

```
plot(find(SparseBass(seg) > 0)/100,SparseBass(find(SparseBass(seg) > 0)),'k.'
        )
%plot(time(seg),Mark(seg),'k')
hold off
```

Bibliography

[1] C.P. Mason. Terpsitone. a new electronic novelty. *Radio Craft*, page 335, December 1936.

[2] R.N. Current and M.E. Current. *Loie Fuller, Goddess of Light.* Northeastern University Press, Boston, 1997.

[3] A. di Perna. Tapping into MIDI. *Keyboard Magazine*, page 27, July 1988.

[4] J. Paradiso et al. Design and implementation of expressive footwear. *IBM Systems Journal*, 39:511–529, October 2000.

[5] W. Siegel and J. Jacobsen. The challenges of interactive dance: An overview and case study. *Computer Music Journal*, 22(4):29–43, 1998.

[6] M. Coniglio. The MidiDancer system. `http://www.troikaranch.org/mididancer.html`.

[7] Measurand. `http://www.measurand.com`.

[8] Polhemus. `http://www.polhemus.com`.

[9] Ascension. `http://www.ascension-tech.com`.

[10] A. Marion. Images of human motion: Changing representation of human identity. Master's thesis, Massachusetts Institute of Technology, 1982.

[11] R. Zacks. Dances with machines. *Technology Review*, pages 58–62, May/June 1999.

[12] A. Camurri et al. Eyes Web - towards gesture and affect recognition in dance/music interactive systems. *Computer Music Journal*, 24(1):57–69, 2000.

[13] Big eye. `http://www.steim.org/steim/bigeye.html`.

[14] Jitter. `http://www.cycling74.com/products/jitter`.

[15] M. Downie. *Choreographic the Extended Agent: Performance Graphics for Dance Theater.* PhD thesis, MIT Media Laboratory, September 2005.

[16] R. Wechlser et al. EyeCon - a motion sensing tool for creating interactive dance, music, and video projections. In *Proc. of the SSAISB Convention*, Leeds, England, March 2004.

[17] L. Carpenter. Video imaging method and apparatus for audience participation. US Patents #5210604 (1993) and #5365266 (1994). Cinematrix.

[18] A.Y. Benbasat and J.A. Paradiso. A compact modular wireless sensor platform. In *Proc. of the 2005 Symposium on Information Processing in Sensor networks (IPSN)*, pages 410–415, Los Angeles, CA, April 2005.

[19] S.J. Morris and J.A. Paradiso. A compact wearable sensor package for clinical gait monitoring. *Offspring*, 1(1):7–15, January 2003.

[20] S.J.M. Bamberg et al. Gait analysis using a shoe-integrated wireless sensor system. to appear in IEEE Transactions on Information Technology in Biomedicine, 2007.

[21] Xbow. `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[22] L.E. Holmquist et al. Building intelligent environments with Smart-Its. *Computer Graphics and Applications, IEEE*, pages 56–64, January/February 2004.

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

[24] E. Flety. The WiSe box: A multi-performer wireless sensor interface using WiFi and OSC. In *Proc. of NIME 05*, pages 266–267, Vancouver, Canada, May 2005.

[25] M. Barry, J. Gutknecht, I. Kulka, P. Lukowicz, and T. Stricker. Multimedial enhancement of a butoh dance performance - mapping motion to emotion with a wearable computer system. In *Proc. of MoMM*, 2004.

[26] K. Kunze et al. Towards recognizing tai chi - an initial experiment using wearable sensors. In *Proc. of IFAWC*, 2006.

[27] H. Junker, P. Lukowicz, and G. Troester. PadNET: Wearable physical activity detection network. In *Proc. of the Seventh IEEE International Symposium on Wearable Computers (ISWC 03)*, page 244, 2003.

[28] Xsens. `http://www.xsens.com`.

[29] K. Laurila et al. Wireless motion bands. In *Ubicomp 2005 Workshops: Ubiquitous computing to support monitoring, measuring, and motivating exercise*, 2005. `http://seattleweb.intel-research.net/projects/ubifit/papers/w10-p3.pdf`.

[30] E. Munguia Tapia et al. MITes: Wireless portable sensors for studying behavior. In *Proc. of Extended Abstracts Ubicomp 2004: Ubiquitous Computing*, 2004.

[31] J. Lester, B. Hannaford, and G. Borriello. "Are You With Me?" - using accelerometers to determine if two devices are carried by the same person. In *2nd International Conference on Pervasive Computing*, pages 33–50, Linz, Austria, April 2004.

[32] L.E. Holmquist et al. Smart-Its Friends: A technique for users to easily establish connections between smart artefacts. In *Proc. of UBICOMP 2001*, pages 116–122, Atlanta, GA, September 2001.

[33] K. 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, Vancouver, Canada, 2003.

[34] M. Feldmeier and J.A. Paradiso. An interactive music environment for large groups with giveaway wireless motion sensors. to appear in Computer Music Journal, 2007.

[35] M. Feldmeier and J.A. Paradiso. Giveaway wireless sensors for large-group interaction. In *Proc. of the ACM Conference on Human Factors and Computing Systems (CHI 2004)*, pages 1291–1292, Vienna, Austria, 2004.

[36] C. Ramakrishnan, J. Freeman, and K. Varnik. The architechture of Auracle: a real-time, distributed, collaborative instrument. In *Proc. of the 2004 International Conference on New Interfaces for Musical Expression (NIME 04)*, pages 100–103, Hamamatsu, Japan, June 2004.

[37] M. Gurevich. JamSpace: designing a collaborative networked music space for novices. In *Proc. of the 2006 International Conference on New Interfaces for Musical Expression (NIME 06)*, pages 118–123, Paris, France, June 2006.

[38] N. Kinns-Bryan and P.G.T. Healey. Decay in collaborative music making. In *Proc. of the 2006 International Conference on New Interfaces for Musical Expression (NIME 06)*, pages 114–117, Paris, France, June 2006.

[39] G. Wang, A. Misra, and P.R. Cook. Building collaborative graphical interfaces in the audicle. In *Proc. of the 2006 International Conference on New Interfaces for Musical Expression (NIME 06)*, pages 49–52, Paris, France, 2006.

[40] T. Blaine and S. Fels. Collaborative musical experiences for novices. *Journal of New Music Research*, 32(4):411–428, December 2003.

[41] B.R. Knapp and P.R. Cook. Creating a network of Integral Music Controllers. In *Proc. of the 2006 International Conference on New Interfaces for Musical Expression (NIME 06)*, pages 124–128, Paris, France, June 2006.

[42] D.A. Winters. *Biomechanics and motor control of human movement.* John Wiley & Sons, 2 edition, 1990.

[43] D. MacKenzie. *Inventing Accuracy, A Historical Sociology of Nuclear Missle Guidance.* MIT Press, 1993.

[44] Northrop grumman. `http://www.nsd.es.northropgrumman.com/Automated/categories/Navigation_Systems.html`.

[45] A.Y. Benbasat. An inertial measurement unit for user interfaces. Master's thesis, MIT Media Laboratory, September 2000.

[46] S.D. Lovell. A system for real-time gesture recognition and classification of coordinated motion. Master's thesis, MIT EECS, January 2005.

[47] Rowley cross works. `http://www.rowley.co.uk/msp430/index.htm`.

[48] L.K. Baxter. *Capacitive Sensors: Design and Applications.* John Wiley and Sons, Ltd., Canada, 1996.

[49] J.A. Paradiso and N. Gershenfeld. Musical applications of electric field sensing. *Computer Music Journal*, 21(2):69–89, 1997.

[50] J. Smith et al. Electric field sensing for graphical interfaces. *IEEE Computer Graphics and Applications*, 18(3):54–60, 1998.

[51] O.D. Grace and S.P. Pitt. Sampling and interpolation of bandlimited signals by quadrature methods. *J of the Acoustical Society of America*, 48(6A):1311–1318, December 1970.

[52] J.R. Smith. *Electric Field Imaging.* PhD thesis, MIT Media Laboratory, February 1999.

[53] Texas Instruments. *MSP430x1xxx Family User's Guide.*

[54] G.J. Welk et al. A comparative evaluation of three accelerometry-based physical activity monitors. *Med Sci Sports Exerc*, 31:171–5, 1999.

[55] M. Sung and A. Pentland. Minimally invasive physiological sensing for human-aware interfaces. In *HCI International 2005*, July 2005.

[56] Polar. `http://www.polarusa.com`.

[57] Zigbee. `http://www.zigbee.org/en/index.asp`.

[58] Bekaert Fabric Technologies. Bekaert. `http://www.bekaert.com/bft`.

[59] Libusb. `http://libusb.sourceforge.net/`.

[60] Pyusb. `http://pyusb.berlios.de/`.

[61] R. Aylward, S.D. Lovell, and J.A. Paradiso. A compact, wireless, wearable sensor network for interactive dance ensembles. In *Proc. of the International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2006)*, pages 65–70, April 2006.

[62] R. Aylward and J.A. Paradiso. Sensemble: A wireless, compact, multi-user sensor system for interactive dance. In *Proc. of the 2006 International Conference on New Interfaces for Musical Expression (NIME 06)*, pages 134–139, Paris, France, June 2006.

[63] T.G. Zimmerman. Personal area networks (PAN): near-field intrabody communication. Master's thesis, MIT, June 1995.

[64] S. Conte, R.K. Requa, and J.G. Garrick. Disability days in major league baseball. *Am J Sports Med*, 29(4):431–6, 2001.

[65] G.S. Fleisig, D.S. Kingsley, et al. Kinetic comparison among the fastball, curveball, change-up, and slider in collegiate baseball pitchers. *Am J Sports Med*, 34:423–430, 2006.

[66] H.S. Tullos and J. King. Throwing mechanism in sports. *Orthop Clin North Am*, 4:709–20, 1973.

[67] G.S. Fleisig, J.R. Andrews, et al. Kinetics of baseball pitching with implications about injury mechanisms. *Am J Sports Med*, 23(2):233–9, 1995.

[68] L. Rizio and J.W. Uribe. Overuse injuries of the upper extremity in baseball. *Clin Sports Med*, 20(3):453–68, 2001.

[69] J. Andrews and K. Wilk. *The Athlete's Shoulder*, chapter Shoulder injuries in baseball, pages 369–89. Churchhill Livingstone, New York, 1994.

[70] T. Seaver. *The Art of Pitching*. Hearst Books, New York, 1984.

[71] S.L. Werner et al. Relationships between throwing mechanisms and shoulder distraction in professional baseball pitchers. *Am J Sports Med*, 29(3):354–8, 2001.

[72] C.J. Dillman, G.S. Fleisig, and J.R. Andrews. Biomechanics of pitching with emphasis upon shoulder kinematics. *J Orthop Sports Phys Ther*, 18(2):402–8, 1993.

[73] Werner S.L., G.S. Fleisig, et al. Biomechanics of the elbow during baseball pitching. *J Orthop Sports Phys Ther*, 17(6):274–8, 1993.

[74] M. Feltner and J. Dapena. Dynamics of the shoulder and elbow joints of the throwing arm during a baseball pitch. *Int J Sport Biomech*, 2:235–59, 1986.

[75] A.M. Pappas, R.M. Zawacki, and T.J. Sullivan. Biomechanics of baseball pitching. A preliminary report. *Am J Sports Med*, 13(4):216–22, 1985.

[76] Y.S. Hang et al. Biomechanical study of the pitching elbow. *Int Orthop*, 3(3):217–23, 1979.

[77] E. Berkson et al. IMU arrays: The biomechanics of baseball pitching. to appear in the Orthopaedic Journal at Harvard Medical School, October 2006.

[78] Fftw. `http://www.fftw.org/`.

[79] M. Niedermayer et al. Miniaturization platform for wireless sensor nodes based on 3D-packaging technologies. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN 2006)*, pages 391–398, Nashville, 2006.

[80] B. O'Flynn et al. The development of a novel miniaturized modular platform for wireless sensor networks. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 370–375, Los Angeles, 2005.