AN INERTIAL MEASUREMENT SYSTEM FOR HAND AND FINGER TRACKING

by

Edward Nelson Henderson

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

December 2011

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Edward Nelson Henderson

Thesis Title: An Inertial Measurement System for Hand and Finger Tracking

Date of Final Oral Examination: 14 October 2011

The following individuals read and discussed the thesis submitted by student Edward Nelson Henderson, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Dr. Thad Welch, Ph.D., P.E.	Chair, Supervisory Committee
Dr. Elisa Barney Smith, Ph.D.	Member, Supervisory Committee
Dr. Nadar Rafla, Ph.D.	Member, Supervisory Committee
Dr. Hao Chen, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Dr. Thad Welch, Ph.D., P.E., Chair, Supervisory Committee. The thesis was approved for the Graduate College by Dr. John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

This thesis would not have been possible without the generous support of the professors and staff of the Boise State University College of Engineering, my thesis committee members, and my advisor Dr. Thad Welch.

I would like to thank Dr. Thad Welch for his encouragement, advice, and wisdom during my time at Boise State University. Dr. Welch has helped me through some difficult times and he has been a true inspiration to me during my academic pursuits. I greatly appreciate his willingness to take on this thesis even when the topic was outside of his realm of focus.

I am grateful for the financial support that I received from Dr. Jim Browning and the Crossed Field Amplifier project. I could not have afforded to stay in school and finish this work without it.

I would like to thank Dr. Elisa Barney Smith for her tireless dedication to the students of the Boise State College of Engineering. Dr. Barney Smith has been an inspiration throughout my undergraduate and graduate career.

I greatly appreciate the efforts of Dr. Nadar Rafla and Dr. Hao Chen for being on my thesis committee and for their time and efforts in reviewing this document.

A very special thank you goes to Mr. Paul G. Savage of *Strapdown Analytics*. Mr. Savage very kindly offered his support in the area of Inertial Navigation Systems. He answered my novice questions and provided practical techniques for implementing the algorithms required for this thesis.

I am deeply indebted to a wonderful group of people that supported this effort by reviewing drafts of this thesis. First and foremost, I must thank Mrs. Donna Welch who offered valuable insight and spent countless hours reviewing my drafts. My brothers David and Daniel and my wife Vicki all took time to review and comment on portions of this work. Finally, Tanner Moore assisted me in cleaning up what I hope are the last remnants of missing commas, weak sentences, and typos.

My family has been very supportive during this process. I cannot thank them enough for their understanding and patience. They are the true inspirations in my life.

I would like to finish by saying that prayer and thanks to God are a regular part of my life. God has opened many doors for me and provided opportunities where none existed. At one point I thought I would never finish a bachelors degree; now I am nearly done with a masters. Thank you, God.

And whatsoever ye do in word or deed, do all in the name of the Lord Jesus, giving thanks to God and the Father by him. Colossians 3:17, KJV.

ABSTRACT

The primary Human Computer Interfaces (HCI) today are the keyboard and mouse. These interfaces do not facilitate a fluid flow of thought and intent from the operator to the computer. A computer mouse provides only 2 Degrees of Freedom (2DOF). Touch interfaces also provide 2DOF, but with multiple points, making the touch interface far more expressive. The hand has 6 Degrees of Freedom (6DOF) by itself. Combined with the motion of the fingers, the hand has the potential to represent a vast array of differing gestures. Hand gestures must be captured before they can be used as a HCI. Fortunately, advances in device manufacturing now make it possible to build a complete Inertial Measurement Unit (IMU) the size of a fingernail.

This thesis documents the design and development of a glove outfitted with six IMUs. The IMUs are used to track the finger and hand positions. The glove employs a controller board for capturing IMU data and interfacing with the host computer. Python™ software on the host computer captures data from the glove. MATLAB™ is used to perform IMU calculations of the incoming data. The calculated data drives a 3D visualization of the glove rendered in Panda3D™.

Future work using the glove would include improved IMU algorithms and development of gesture pattern recognition.

TABLE OF CONTENTS

ABSTRACT v

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF CODE LISTINGS xiii

LIST OF ABBREVIATIONS xv

1 INTRODUCTION 1

1.1 The Human Computer Interface 2

1.2 History of Computer Interfaces 5

1.3 Motion Capture 10

1.3.1 Video Motion Capture Techniques 11

1.3.2 Inertial Measurement Motion Capture 12

1.3.3 Hand and Finger Motion Capture 14

1.4 Glove History 15

1.4.1 Early Gloves 15

1.4.2 Accelerometer Based Gloves 15

1.5 Inertial Based Glove Motion Capture 20

1.5.1 The GyroGlove 23

2 INERTIAL MEASUREMENT SYSTEMS 24
2.1 A Brief History of Inertial Measurement 24
2.2 Inertial Measurement Key Concepts 27
2.2.1 Comparison of Global Positioning System and Inertial Measurement Units 28
2.2.2 Inertial Measurement Unit Types 28
2.2.3 Coordinate Systems 29
2.2.4 Navigation Reference Frames 32
2.3 The Mathematics of Inertial Navigation 36
2.3.1 Reference Frame Transformations 36
2.3.2 Direction Cosine Matrix Updates 41
2.3.3 Accelerometer Updates 43
2.3.4 Inertial Measurement Unit Initial Alignment 43
2.3.5 Error Sources for Inertial Measurement 46
2.4 Summary 46

3 HARDWARE AND SOFTWARE TOOLS 47
3.1 Board Design Tools 47
3.2 Board Assembly Tools 49
3.3 System Testing Equipment 50
3.4 Firmware Development Tools 51
3.4.1 Processor Configuration and Debug 51
3.5 Software Development Tools 52
3.5.1 Python™ Language 52
3.5.2 MATLAB™ Environment 54
3.5.3 Panda3D™ Library 55

4 GYROGLOVE SYSTEM DEVELOPMENT 57
4.1 GyroGlove Design 57
4.1.1 Data Handling 58
4.1.2 Glove Versions 59
4.2 Hardware Design 60
4.2.1 Inertial Measurement Units 60
4.2.2 Controller Board 64
4.3 Firmware Design 68
4.3.1 Gyro Command Processor 71
4.3.2 I^2C Transaction Primer 72
4.3.3 I2C_Master 75
4.3.4 IMU and IMUManager 75
4.3.5 Packet Data Rate Calculations 81
4.4 Software Design 81
4.4.1 Python™ 82
4.4.2 Panda3D™ 84
4.4.3 MATLAB™ 85
4.4.4 MATLAB™ Mex Functions 88

5 RESULTS 90
5.1 Hardware Results 91
5.2 Software Results 91

6 CONCLUSION AND FUTURE WORK 93

REFERENCES 95

A Python™ GLOVESERVER SOURCE CODE 99

B Panda3D™ Python™ SOURCE CODE 123

C MATLAB™ CODE 126

D MATLAB™ MEX CODE 154

E FIRMWARE CODE DOXYGEN OUTPUT 159

LIST OF TABLES

4.1 IMU Class Run Method Initial State Table 77

LIST OF FIGURES

1.1 ENIAC Computer at the University of Pennsylvania[13] 6

1.2 The UNIVAC 1 - First Commercial Computer[3] 6

1.3 The First Computer Mouse and Screen 7

1.4 The Apple™ Macintosh™[2] 7

1.5 Early Trackball Device[6] 8

1.6 Space Navigator three-dimensional Mouse[7] 9

1.7 Xsense™ Biomech™ Suit[9] 13

1.8 Xsense™ MTx™ Sensor[8] 13

1.9 Acceleration Sensing Glove[24] 16

1.10 Early and Commercial AcceleGlove 17

1.11 Vietnamese Sign Language Glove[11] 17

1.12 KHU-1 Korean Accelerometer Glove[23] 18

1.13 Glove Proposed by P. Asare[10] 19

1.14 Body Reference Frame for Finger 21

2.1 A simplified drawing of the ship's inertial navigation system[15]. 26

2.2 An illustration of a frame translated relative to the reference frame. . . . 30

2.3 An illustration of frame translated and rotated relative to the reference frame. 31

2.4 An illustration of the body frame axes for the right hand. 31

2.5 An illustration of the ECI coordinate system. 33

2.6 An illustration of the ECEF coordinate system. 34
2.7 An illustration of rotation and translation of coordinate system. 37
2.8 Coordinate system rotation about the z-axis with angle ψ. 38
2.9 Coordinate system rotation about the z-axis with angle ψ and the y'-axis with angle θ. 39
2.10 Coordinate system rotation about the z-axis with angle ψ, the y'-axis with angle θ and the x''-axis with angle ϕ. 39

4.1 Block Diagram of GyroGlove 58
4.2 The GyroGlove 59
4.3 3D PCB Render of the IMU Board 61
4.4 GyroGlove IMU 61
4.5 Controller Board 65
4.6 Controller Board Plane Layers Detail 65
4.7 Block Diagram of GyroGlove Controller and IMU Boards 66
4.8 GyroGlove Firmware Architecture Block Diagram 69
4.9 I^2C Start and Stop Transactions 73
4.10 I^2C Write Transaction 73
4.11 I^2C Read Transaction 73
4.12 I^2C Combined Transaction 74
4.13 Partial State Machine Diagram for the IMU Class 76
4.14 GloveServer GUI 82
4.15 UML Diagram of the Python™ GloveServer 83
4.16 Google Sketchup™ Drawings Used in the Panda3D™ 3D Visualizer 84
4.17 GloveGui UML Diagram 86
4.18 MATLAB™ Glove GUI 87

LIST OF CODE LISTINGS

4.1 I^{2}C Class Creation . . . 69

4.2 Initialization of the IMUManager Class and the IMU Classes . . . 70

4.3 Initialization of the HardwareSerial and GyroCommandProcessor Classes 70

4.4 GyroAcc Main Loop . . . 71

4.5 GyroCmdProcessor Loop Example . . . 72

4.6 IMU Class ProcessTransaction Method . . . 77

4.7 IMU_Manager Loop Method . . . 78

4.8 IMU_Manger SendPacket Method . . . 79

A.1 Top Level . . . 100

A.2 IMU Manager . . . 105

A.3 Application Programming Interface . . . 106

A.4 Packet Handling . . . 116

A.5 User Interface . . . 121

B.1 3D Server . . . 124

C.1 GloveGui Top . . . 127

C.2 GloveGui Base . . . 130

C.3 GUI Base . . . 134

C.4 Platform IMU . . . 137

C.5 Glove IMU . . . 141

C.6 IMU Core . . . 147

C.7 Hand Kinematics . . . 151

D.1 GyroGlove Main 155
D.2 GyroGlove Client 156

LIST OF ABBREVIATIONS

ADC	analog-to-digital converter
API	application programming interface
ASG	acceleration sensing glove
ASL	American sign language
BOM	bill of materials
CAD	computer aided design
DPS	degrees per second
DCM	direction cosine matrix
DOF	degree of freedom
ECEF	earth centered earth fixed
ECI	earth centered inertial
ENIAC	Electronic Numerical Integrator And Computer
ESGN	electrostatic gyro navigator
GUI	graphical user interface
GPS	global positioning system
HCI	human computer interface

Hz Hertz

IMU inertial measurement unit

ICE in-circuit emulator

IFOG interferometric fiber optic gyro

INS inertial navigation system

IR infrared

I^{2}C inter-integrated communications

JTAG joint test action group

KBPS kilobits per second

MEMS microelectromechanical systems

NED north east down

PARC Palo Alto Research Center

PC personal computer

PCB printed circuit board

PDA personal data assistant

POS point-of-sale

QFN quad flat-pack no lead

SIGGRAPH Special Interest Group on Computer Graphics and Interactive Techniques

SINS ship's inertial navigation system

SRI	Stanford Research Institute
SPI	serial peripheral interface
TQFP	thin quad flat pack
UNIVAC I	UNIVersal Automatic Computer I
USB	universal serial bus
UML	unified modeling language
2D	two-dimensional
3D	three-dimensional
6DOF	6 degrees of freedom

CHAPTER 1

INTRODUCTION

Recent advances in microelectromechanical systems (MEMS) technology have reduced the size of accelerometer and gyroscope devices to the point where it is practical to construct an inertial measurement unit (IMU) with 6 degrees of freedom (6DOF) the size of a fingernail. A glove designed with a complete IMU placed at the tip of each finger can capture the motion of the fingers, and be used for gesture recognition. This thesis outlines the design and implementation of such a glove. Hand gestures tracked using such a glove can be used to control computer functions currently accessed using a keyboard and mouse. Gesture capture has the potential to significantly advance the current state of human computer interfaces (HCI).

A set of IMUs were designed along with a controller for managing them. A total of six IMUs were then assembled into a glove, mounted one per finger, one on the thumb, and one on the back of the hand. Data from this glove was captured, processed, and visualized on a personal computer (PC).

This thesis begins with some background information and relevant history. Chapter 1 includes a review of the history of HCI and some background information about inertial navigation. The chapter presents several types of motion capture. The chapter concludes with an introduction to the capture technique used for this thesis.

Chapter 2 presents the topic of inertial navigation systems. The chapter begins with a history of inertial navigation leading up to the current state-of-the-art in the field. Section 2 defines key concepts needed to understand inertial navigation. The chapter concludes with the theoretical and mathematical background required for such systems.

Chapter 3 is a reference for the hardware and software used on this project, including a summary of the test equipment, software development tools, and assembly tools.

Chapter 4 presents the GyroGlove and provides an overview of this project, along with discussions of key topics. The hardware design is presented with block diagrams and descriptions of major components. The chapter provides high level details about the microprocessor firmware, MATLAB™ , Python™ , and C++ software.

Chapter 5 presents the results of the GyroGlove project and discusses those aspects that were successful, as well as those that were not. Chapter 6 presents conclusions and a discussion about the possible future for the GyroGlove project.

1.1 The Human Computer Interface

The idea behind this thesis stems from an interest in inertial navigation systems (INSs) and a desire to use this technology to improve how users interact with their computers. Computers are useful in a wide range of applications, but the mechanisms used to interact with them have changed very little over the past decade or two.

Engineers use computers for the design and modeling of mechanical, electrical, and architectural systems. Artists use computers to create two-dimensional (2D) and three-dimensional (3D) still graphics, as well as animated 3D scenes. Computers manage point-of-sale (POS) transactions in many industries. In the medical field, computers schedule patient visits, monitor vital signs, and facilitate face-to-face interaction between patient and doctor via remote presence. Auto mechanics use computers to perform vehicle diagnostic tests and to search for the proper repair procedure. Aircraft cockpits, once dominated by dials and needles, are filled with computer-generated displays. It is difficult to find an area in modern society where computers do not play some type of role. As ubiquitous as computers are, the mechanisms that we use to interact with them have barely changed over the past two decades.

Various techniques for computer interaction have been developed and those techniques continue to evolve. Unfortunately, the currently available mechanisms still present a huge bottleneck when interacting with modern computers capable of performing many tasks simultaneously. A computer will spend over 90% of its processor time waiting for user input. Experienced computer users can think about a sequence of tasks much quicker than they can initiate them. The need to point to menu items, click, drag, and select hinders the potential efficiency of many day-to-day tasks.

Computer operating systems and most programs support shortcut keys. Such combinations involve sequences of keys combined with the CONTROL, ALT, and SHIFT keys. Shortcuts can speed experienced users through common operations, but the learning curve is generally quite long and many users never gain the experience to use more than a few.

More expressive interface devices would allow the user to access relevant menus more quickly, be more precise when selecting options, and offer a more natural means of interacting with programs. An ideal HCI would become a natural extension of our bodies and allow for a wide range of expressions to rapidly and effectively indicate the user's intentions to the computer system.

The technologies available for interacting with the computer are an important driver of the way that software is written. Most software uses menus, drop-down lists, and pop-up menus. Such mechanisms are far from ideal, but they are the best options available given the current state of HCI. Software tools designed for 3D modeling often use a combination of key presses and mouse moves for rotating an object. A more natural way to interact with a 3D object would be to reach out and grab the object with your hand. Such a natural gesture would be easy to learn. New developments in the state-of-the-art for HCI devices will bring associated changes to the software written for those devices. Big changes in the HCI would have a major impact on how we work with computers, but the industry is still waiting for the next big breakthrough.

Some devices have made great strides in the right direction. The iPhone™ introduced a new interface mechanism. When the iPhone™ was released, touch screens were not new, but they had yet to be utilized beyond a few niche application areas. Nonetheless, the trend in smart phones is definitely toward the touch screen interface.

The new interface mechanisms for portable devices have been a large step in the right direction. The touch interfaces on the iPhone™, iPad™, and Android™ devices are vastly superior to the interface used on previous handheld interfaces, such as the stylus. The new touch interfaces have revolutionized the handheld device marketplace. These handheld devices are making an impact in areas traditionally dominated by the PC. iPad™s have blurred the lines between a handheld device and a computer, with many of the common PC tasks easily accessible on these new devices. Small computer devices like the iPad™ have begun to replace the PC for tasks such as web surfing, basic e-mail, gaming, reading, and similar applications. However, the PC will continue to be the preferred platform for more intensive applications such as engineering, 3D mechanical design, graphics design, and similar applications.

There have been some attempts to use the touch interfaces on more traditional computers. Several computer manufacturers have produced computers with a touch panel in additional to the typical mouse and keyboard. It is not clear that these improvements have been particularly successful in the market place. One challenge is user familiarity and comfort. It would take time for users to accept and embrace these new touch panel interfaces. Another challenge is that most programs are not written specifically for such an interface. Programs for touch devices such as the iPhone™ have been written specifically to use the touch interface while programs on the typical computer have not. The combination of the program interfaces and user familiarity make the introduction of these interfaces a huge marketing challenge. Widespread market penetration for new technologies like touch panels would require a significant increase in usability — enough of an increase to offset the typically slow adoption rates of such technology innovations. Current touch technologies do not offer enough of an increase to drive such adoption.

Computers have made huge leaps in processing power in the past decade. Unfortunately, the HCI mechanisms have failed to maintain the same pace of innovation. The powerful computers of today are unable to truly augment our thought processes. The computer is hampered by the fact that it must wait patiently while the user moves the mouse, types on the keyboard, and clicks away wildly, attempting to transmit the free-flowing ideas in his or her mind into something that the computer can respond to. The HCI is arguably the single largest factor hindering the free flow of ideas from the mind to the computer.

1.2 History of Computer Interfaces

The history of the computer has seen many changes in the HCI. The earliest computing devices used punch cards and plug boards [28] .

The world's first digital computer was the Electronic Numerical Integrator And Computer (ENIAC)[18]. The ENIAC, shown in Figure 1.1, was developed at the University of Pennsylvania in the mid 1940s and dedicated in 1946 [18]. The ENIAC used punch cards for programming, as did many of its successors [5].

Cards were replaced with paper tape, and eventually magnetic tape. Keyboards were used for interaction starting in the early 1950s [5]. The UNIVersal Automatic

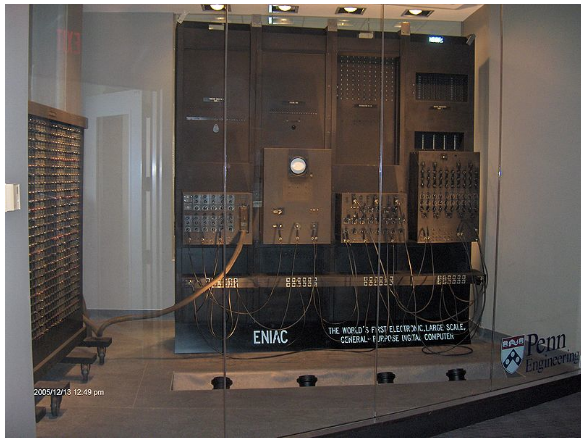


Figure 1.1: ENIAC Computer at the University of Pennsylvania[13]

Computer I (UNIVAC I), shown in Figure 1.2, was the first commercial computer. The UNIVAC I was used by the Census Bureau in 1951.

Figure 1.2: The UNIVAC 1 - First Commercial Computer[3]

Amazingly, the first mouse, shown in Figure 1.3a, was actually invented back in the late 1960s. A paper published in 1967 described the use of a joystick, a Grafacon, and a mouse [16]. The devices were developed at Stanford Research Institute (SRI) by Douglas Engelbart. The mouse did not become a part of everyday life until almost 20 years later, however. While Engelbart did develop something of a graphical interface, shown in Figure 1.3b, the modern graphical user interface (GUI) did not come about until many years later.

(a) **First Mouse**[12] (b) **First Mouse and Screen**[4]

Figure 1.3: The First Computer Mouse and Screen

There were no significant advances in computer interfaces until the 1980s when Apple™ computer released the Macintosh™. The predecessor to the modern GUI was developed at Xerox Palo Alto Research Center (PARC). These developments were communicated to Apple™ and Microsoft™, both of which began work on the first widely used GUI platforms [26]. Apple™ released its first Macintosh™, shown in Figure 1.4, in 1984, which is credited as being the first GUI computer available to the public. The keyboard and mouse combination, together with graphical elements on the screen that were controlled by the mouse, revolutionized the computer. These enhancements made the computer accessible to a wider audience.

Figure 1.4: The Apple™ Macintosh™[2]

In the ensuing decades, the primary computer interface did not change fundamentally. The keyboard and mouse were still by far the dominant method of interacting with the computer. Other methods were tried, and some worked for specific applications or user tastes. Trackballs, such as the one shown in Figure 1.5, turned the mouse upside down. Instead of moving the mouse, which rolled the ball on the table, you moved the ball. The goals were the same, but some users found the ergonomics more comfortable.

Figure 1.5: Early Trackball Device[6]

Some industrial or process control applications used special monitors to provide a touch screen interface. This reduced or eliminated the need for the keyboard and mouse, especially in industrial environments with dust and other contaminants. The early touch screens could detect a single touch at one point on the screen. The single touch screens were limited, and worked much more like a mouse. They have little in common with the multi-touch screens found on modern devices.

Specific application areas often drive a need for a custom interface mechanism. Engineers or graphics designers that work with 3D models manipulate the computer representation of their work in complex ways. A standard mouse and keyboard can be used for this purpose, but they are not very efficient. As an example, the SolidWorks™ mechanical engineering package uses a complex set of keyboard, mouse click, and button combinations to select between pan, zoom, and 3D rotate modes.

Figure 1.6: Space Navigator three-dimensional Mouse[7]

Such programs benefit from a special type of mouse that provides three degrees of freedom that make manipulation of 3D objects easier. One such device is the Space Navigator™ 3D mouse, shown in Figure 1.6.

Palm™ computer developed personal data assistant (PDA) devices with unique interfaces. These devices used a stylus for interaction. The devices were simple, as there was no need for a keyboard, but the interface was generally quite slow. Users could learn a *stroke* language and increase their efficiency, but the learning curve was steep. These devices were successful for a number of years, but as more advanced devices arrived, the stylus interface began to find fewer adherents.

In 2007, Apple™ released the iPhone™. This device uses a multi-touch screen for interaction with the phone. The multi-touch screen allows for much more expressive input from the user. Now it is not only possible to point at an item, it is possible to manipulate that item in very easy and intuitive ways. Squeeze two fingers together and the image shrinks, spread them apart and zoom in. Using gestures that feel natural make them easily accessible and allow anyone to learn them quickly. The multi-touch interface is in many ways the first major shift in the computer user interface since the mouse and GUI. In a few short years, the interface has moved from a novelty to the de facto standard for all such portable devices.

While the multi-touch screens are well suited for portable devices, they are not ideal for all applications or situations. The multi-touch screens provide the ability to track multiple finger touches and movements. Five finger gestures are possible in advanced devices. The limit, however, is that all of the gestures are constrained to a 2D plane. Gestures can swipe across, squeeze together, and rotate. The multi-touch screen, while still a great improvement over traditional HCI methods, still locks the user into a 2D world when we actually live in a 3D world.

While gesturing in 2D is helpful, gesturing in 3D would be significantly more expressive. In 2D, one might raise a finger and tap on the surface to indicate a "click." In 3D, the height of the finger, and other aspects of the finger movement may have significance. The velocity of the movement, the maximum height, even a finger waving gesture could infer a specific operation. The desire to capture the 3D movements of the hand and fingers is not new, however recent advances in sensor and electronic technology are bringing 3D gesture capture closer to a practical reality.

1.3 Motion Capture

Every mechanism that we use to interact with the computer is in effect some form of motion capture. A keyboard captures the motion of the fingertips on a 2D plane surface with specific functions assigned to each position on the plane — the J key for example. We use touch cues — the raised bumps on the F and J keys — to orient ourselves to the environment. With training, we learn to pinpoint specific locations on the keyboard without looking. The fastest touch-typists can type well in excess of 100 words per minute, while many people resort to a hunt-and-peck method. There is a great deal of variability in keyboarding skills among computer users. For most, however, the keyboard is a painfully slow means of computer interaction.

The mouse is in essence just another means of motion capture. We grip the mouse, and move our hands. The mouse tracks the position of our hand by sensing the surface upon which it rests. The sensing method and accuracy have improved over the years, but the basic technique has remained the same. Buttons and wheels on the mouse allow for other types of motion capture — the wheel captures an up-down or in-out (depending on how the motion is interpreted) motion of our finger. Buttons capture a clicking motion.

The mouse and keyboard are both very archaic means of motion capture. The keyboard requires extensive training to master. The mouse is limited to a 2D plane, and may add some buttons or wheels. Moving the mouse requires our entire hand, and often results in substantial user fatigue or even injury. Neither of these methods come close to capturing the full expressiveness of the human body. Our fingers, hands, and arms exhibit many degrees of freedom. The more degrees of freedom a system can capture, the more able it is to provide a truly expressive interface to the computer.

1.3.1 Video Motion Capture Techniques

Video processing is a well-established motion capture technique. With this technique, multiple cameras are required in order to achieve some amount of depth perception. Some motion capture systems require the subject to wear a suit with target dots that mark key spots on the person being tracked. The dots allow the camera tracking software to have a clear and unambiguous target. Multiple camera angles are generally required to determine position in a 3D frame of reference[19].

Camera video tracking systems generally require an extensive setup. Cameras must be placed precisely, and special suits are often required. Improvements in processing power and some novel techniques have driven recent improvements in video tracking. The Microsoft™ Kinect™ system was introduced on November 4th, 2010. The Kinect™ paints the environment with an infrared (IR) laser. The laser dots are then tracked by a pair of cameras mounted on a boom. The laser dot pattern deforms in predictable ways as the objects in the field move. The current Kinect™ systems track major body parts such as the hands, legs, and head. It is likely that future enhancements will be able to track more detail, such as the motion of the fingers and hands.

Video motion capture is not suitable for all environments. Mobile applications would be poor candidates for a video tracking system with fixed cameras. Outdoor applications would make use of an IR laser virtually impossible. Not every application would be practical with a camera system facing the user. More flexible and portable systems are needed.

1.3.2 Inertial Measurement Motion Capture

Inertial Measurement techniques for motion capture are gaining in popularity and capability. Inertial Measurement has been around for decades, but the size of such units has limited their use in tracking small items. Recent advances in technology have reduced the size of such devices to the point that they are small enough to be mounted on a circuit board the size of a fingernail. Smaller devices, however, generally lead to sacrifices in accuracy, limiting their effectiveness for motion capture systems. Body motion capture devices are larger and designed to be as accurate as possible.

The motion capture industry has been driven by athletics and Hollywood, due in large part to the funds available in these industries. Motion capture benefits a great deal from unconstrained systems. Systems that require precision camera setups constrain the actors or athletes. Video capture systems might be impractical for capturing a skier on a giant slalom run, but inertial suits have been used to capture motion during such events in great detail.

Several companies manufacture commercial motion systems. One such system is the Biomech™ suit from Xsens™, shown in Figure 1.7. The Biomech™ utilizes 17 inertial motion trackers located at key points on the body. This suit captures the movement of major body parts and records them for further processing and analysis.

Figure 1.7: Xsense™ Biomech™ Suit[9]

Figure 1.8: Xsense™ MTx™ Sensor[8]

The motion trackers used on the Biomech™ suite are the Xsens™ MTx™, shown in Figure 1.8. The MTx™ units measure 38 x 52 x 21 mm and are highly accurate.

The Biomech™ suit works great for tracking the entire body. The tracking units, however, are much too large to track smaller body parts, such as a finger.

1.3.3 Hand and Finger Motion Capture

The most expressive parts of our bodies are the hands and the face. Some work has been done on using facial features as a computer interface [30]. Such a facial recognition system would be usable by an individual with upper limbs amputated. Such systems are not likely solutions for day-to-day work. The hands are a far more natural way to interact, since we already use them every day to interact with the world around us.

The hands are able to point, gesture, manipulate, and grasp. The fingers have great range of motion in some directions with limited motion in others. The thumb works in opposition to the fingers for gripping of objects. Tracking the hand and fingers in 3D would allow the capture of a large number of expressions and gestures. Gestures could be used for a variety of tasks that we currently perform on our computers:

* Virtual Keyboard Interface.
* Finger movements to scroll windows.
* Gestures to switch applications or windows.
* Manipulation of 3D objects for computer aided design (CAD) or Graphic Design work.

Currently available applications could make use of a gesture-based input. However, real improvements in HCI will require a new breed of application, written specifically with gesture interaction in mind.

1.4 Glove History

A literature and internet review shows that there is a significant amount of activity in the IMU-based motion capture industry. Motion capture specifically for hands and fingers is less well represented. However, there have been some university and commercial developments. Because there is no definitive source on the history of such gloves, it is not possible to know for sure that all previous developments have been identified.

1.4.1 Early Gloves

One of the first gloves designed to capture hand gestures was built for the Nintendo™ game system. The Power Glove™ was designed as a game controller and released in 1989 [25]. This glove uses sensors to determine the finger bend angles.

A current commercial glove that uses bend sensors is the CyberGlove II™, manufactured by CyberGlove™ Systems [14]. The CyberGlove™ tracks finger position by monitoring bend angles of the finger joints with the bend sensors. The position data is transmitted wirelessly to a host computer where it can be used for a range of applications.

In 2001, Sony Computer Science Laboratories, Inc. designed a tracking wrist that uses capacitive sensors for capturing gestures and touch [27]. The GestureWrist™ uses a device that capacitively measures the changes in wrist shape and movements of the forearm [27].

1.4.2 Accelerometer Based Gloves

There have been several gloves developed that use accelerometers for motion tracking. The first such glove was developed by a group of undergraduate engineers at the University of California, Berkeley in 1999. The glove was called the acceleration

Figure 1.9: Acceleration Sensing Glove[24]

sensing glove (ASG) [24]. The ASG was well ahead of its time. The glove, shown in Figure 1.9, used 2-axis accelerometers on each finger and the thumb.

The ASG used an Atmel processor and analog accelerometers. The analog accelerometers, common during that period, must be digitized using an analog-to-digital converter (ADC). One challenge faced by the engineers at that time would have been noise. Today, 3-axis devices are common. At that time, the 2-axis device was probably state-of-the-art. The team at Berkeley oriented the x-axis perpendicular to the plane of the fingernails, and the y-axis parallel to the axis of the fingers. The orientation allowed the glove to sense the curling motion of the fingers, but not left or right motion. Given the limitations of the technology existing at that time, the ASG was a very impressive accomplishment.

A paper presented in 2002 to the Special Interest Group on Computer Graphics and Interactive Techniques (SIGGRAPH) showed another acceleration glove designed to recognize American sign language (ASL) [22]. This glove, shown in Figure 1.10a, also used 2-axis accelerometers, again placing one on each finger, the thumb, and the hand. The glove described in the 2002 paper is quite rudimentary, however there is currently a commercial company that bears the name of AcceleGlove™, and it is quite

(a) Early AcceleGlove[22] (b) Commercial AcceleGlove[1]

Figure 1.10: Early and Commercial AcceleGlove

possible that the developers writing the 2002 paper went on to found the commercial AcceleGlove™ entity.

While the 2002 AcceleGlove™ used 2-axis accelerometers, the more modern commercial AcceleGlove™, seen in Figure 1.10b, uses 3-axis accelerometers[1]. The commercial glove does not appear to use gyroscopes in the current production design.

Figure 1.11: Vietnamese Sign Language Glove[11]

The team of Duy Bui and Long Thang Nguyen designed a glove in 2007, shown in Figure 1.11. Their glove was used to perform gesture recognition of Vietnamese Sign Language[11]. Like the ASG, this glove used six 2-axis accelerometers located on the fingers, thumb, and the back of the hand.

The sign language glove used digital accelerometers that generated a continuous sequence of pulses. The duty cycle of the pulse train varied based on the acceleration detected by the device, allowing the host processor to calculate the acceleration by measuring pulse width. The glove used a Parallax™ BASIC Stamp micro controller for the onboard processor. One unique aspect of this project was the use of fuzzy logic for performing the gesture recognition.

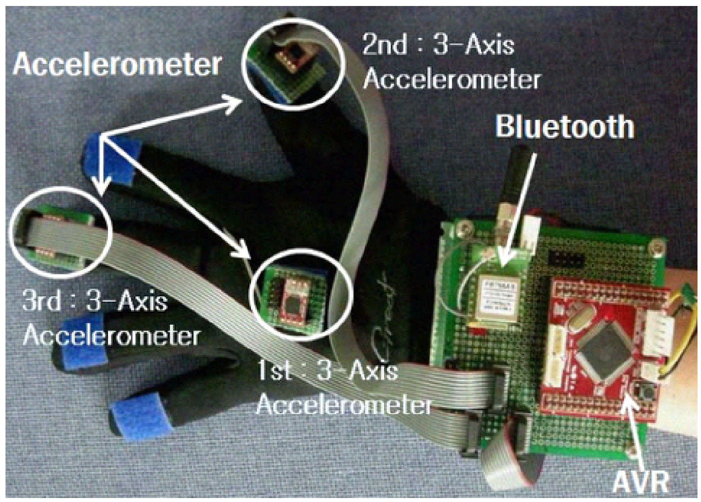


Figure 1.12: KHU-1 Korean Accelerometer Glove[23]

A paper published in 2009 at Kyung Hee University, South Korea, demonstrated 3-D motion tracking of the hand and fingers using accelerometers [23]. The students built a working glove. The glove was called the KHU-1 and is shown in Figure 1.12. The KHU-1 was the first glove[1] known to have used 3-axis accelerometers. The authors of this paper noted that the use of 3-axis accelerometers would expand the possible uses of such a glove to three dimensions, whereas the previous versions were only suitable for 2D gesture captures.

The KHU-1 glove used analog sensors. These sensors required an ADC to capture and digitize the accelerometer outputs. The KHU-1 performed ADC measurements at a 20 Hertz (Hz) rate. The ADC captured the accelerometer values with 10 bits of

[1]The author understands that some gloves may have been developed privately with no published papers to document their existence.

precision. The sample rate and ADC precision of this glove would be much too slow and inaccurate for most purposes. Motion capture for biomechanics would generally require a sample rate in the 150 Hz to 200 Hz range. 10 bits of ADC precision would be very limiting, and the algorithms used to compute motion would be difficult to implement. Nevertheless, the KHU-1 was able to successfully recognize some basic gestures.

The final glove found during the literature search has not been built yet. The glove is proposed in Asare[10] and shown in Figure 1.13. The proposed glove consists of 11 3-axis accelerometers and 14 pressure sensors [10]. The purpose of the pressure sensors is to detect hand positions and movements that the accelerometers cannot detect.

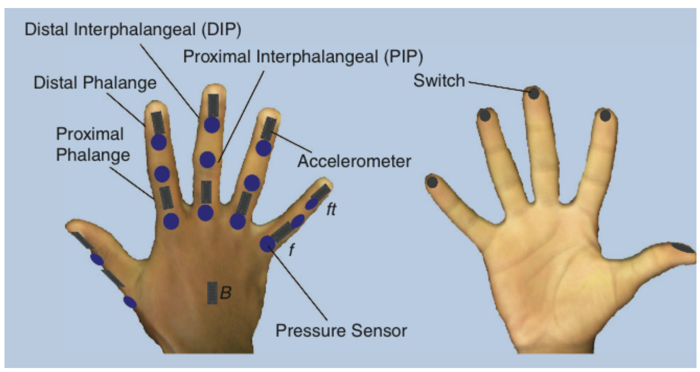


Figure 1.13: Glove Proposed by P. Asare[10]

The glove proposed in [10] would use the accelerometer to detect motion and the orientation of the glove. The gravity vector, which causes a fixed acceleration[2] of 1 G, is used to determine the orientation of the glove.

No other gloves were found during the literature review. It is quite possible that private or commercial firms have developed gloves but refrained from documenting their development in order to maintain competitive advantage. It is also possible that new gloves are under development but have yet to be announced.

[2]G is the mathematical symbol that represents the acceleration due to gravity, which is $9.8\frac{m}{s^2}$.

Gyroscopes are the one thing missing from all of the gloves identified during this review. Accelerometers can be used to determine the orientation of a body part by tracking the gravity vector. Gravity vector tracking is difficult or impossible when acceleration due to motion is also present. Gyroscopes add the ability to track changes in the orientation even when rapid motion is present. Tracking rapid motion would be required in order to capture gestures and signals from a hand moving at natural speeds.

1.5 Inertial Based Glove Motion Capture

According to Farrell [17], an inertial measurement unit is a device that measures both the acceleration and the rotation of a vehicle in 6 degrees of freedom. Measuring just acceleration is not adequate to accurately reconstruct the position of the vehicle[3] during typical hand movements. Consider an attempt to use just a 3-axis accelerometer for this purpose. Consider further that this accelerometer is located on the back of the right hand with the z-axis perpendicular to the hand, the y-axis directed to the right, and the x-axis directed toward the fingers. Figure 1.14 shows the coordinate frame described on the hand.

With the palm facing downward, the x- and y-axes will read zero. The z-axis will read -1G, which indicates that the force of gravity is directed along the negative z-axis. Consider what happens as the hand is rotated slowly clockwise through 90°. The -z-axis reading will decrease from -1G to 0 and the +x-axis reading will increase from 0 to +1G. The angle θ, can be determined from the formula $\theta = \cos^{-1}(\frac{Z}{1G})$. Any motion of the hand will cause an acceleration in the axis of motion. This motion may be along more than one axis, so the actual motion vector must be calculated based on the acceleration in all three axes. Once the hand begins to move it becomes more difficult, if not impossible, to accurately determine where the gravity vector is. The

[3] *Vehicle* in this context represents the entity that is being tracked, such as a fingertip, hand, etc.

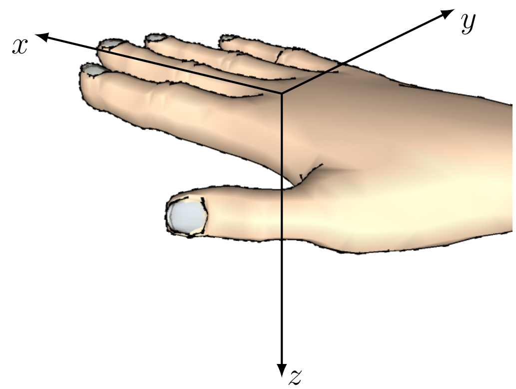


Figure 1.14: Body Reference Frame for Finger

motion of the hand will add acceleration to the measurement, making separation of the gravity vector and the motion vectors challenging.

The purpose of tracking the gravity vector is to maintain knowledge of the orientation of the hand. In the previous example, if the hand initially moved upward, then the accelerometers would register acceleration in the -z-axis. If the hand then rotated 90° clockwise and moved right, the acceleration would still be in the -z-axis, since this axis has been turned to point to the right. If the processor can maintain a value that accurately represents the current orientation, then the acceleration values will be applied in the proper direction at all times. Also, the gravity vector will be known relative to the orientation, and this acceleration can be removed from the motion acceleration. Maintaining an accurate orientation of the vehicle is probably the most critical factor determining the accuracy of an IMU.

Adding gyroscopes to the vehicle allows the system to monitor the rotation rate of the vehicle. Any change in orientation will involve rotation in one or more axes. Integrating the rotation rate will yield the number of degrees of rotation. Tracking the rotation versus time will provide a current orientation of the hand. When the vehicle motion is slow or constant, then the gravity vector can again be used, this time to ensure that the orientation determined by the gyroscopes has not drifted too far from the actual orientation. A key system parameter is how accurately the processor's calculated orientation matches the actual value.

Devices used to build real world IMUs have real world noise and real world offsets. Offsets, or bias, are fixed deviations of the measured value from the real value. An accelerometer that is oriented perpendicular to the gravity vector should read zero, but the probability that a real device reads exactly zero is very low. The device will read a value near zero, but all practical devices will have some offset. Fortunately, offset values are static and fixed. They will usually change with temperature variations, but within a practical time frame they should not vary significantly. Static offset values can therefore be calculated and compensated for.

Practical devices have noise and the values change from time to time. Some of this noise can be filtered out using low-pass filters, but it is not possible to eliminate all noise without eliminating useful information. Newer and better gyros and accelerometers reduce the amount of noise, but will never be able to eliminate it. Very expensive gyro systems using lasers and light rings can reduce the noise values to extremely low values, but the small MEMS devices used in portable systems have much higher noise levels.

All IMUs will deviate in their calculated position versus their actual position over time. A vehicle using an IMU will set an initial position, and use the IMU to track the motion relative to this initial position. In every case, the calculated position will drift away from the actual position over time. The quality of the sensors determines how long the calculated position will stay within an acceptable range. IMUs built with MEMS devices can maintain an accurate position for only a short period of time, after which their position will deviate from the actual position by a significant amount. Some method is required to re-set or compensate for this positional drift.

For this project, the hand position relative to the physical environment is not critical. The important information is the relative motion and position of the fingers and hand. For example, it is not critical to know if the hand is 10 cm above the table or 20 cm, only that the fingers are forming a particular gesture at this point in time. The theory is that the inaccuracies inherent in a practical IMU will not adversely affect the goals of this project.

1.5.1 The GyroGlove

The remainder of this document describes the design and development of a glove capable of IMU-based capture of hand and finger motion data. The GyroGlove was completed as part of this thesis project. The GyroGlove includes 6 complete IMUs mounted on the hand, fingers, and thumb. The completed glove includes the GyroGlove hardware, firmware, and software.

CHAPTER 2

INERTIAL MEASUREMENT SYSTEMS

Inertial measurement uses no sensors outside of the vehicle. There is no direct measurement of either position or velocity. The only measurements available are the acceleration and angular rate of the vehicle, both of which are measured with vehicle-mounted sensors. This chapter will take a brief look at the history of inertial navigation, including a look at some modern devices. Next, some key concepts of IMUs will be examined. The chapter will conclude with a presentation of the math required in an IMU.

2.1 A Brief History of Inertial Measurement

Nearly all of the information in this first section was gleaned from a paper written by Charles Stark Draper, titled "Origins of Inertial Navigation" [15]. Dr. Draper was born in 1901. He received a Ph.D. in physics from the Massachusetts Institute of Technology (MIT) in 1938. Dr. Draper is considered by many to be the "father of inertial navigation." He had a leading roll in the development of many of the technologies we use today. The Charles Stark Draper Laboratory at MIT bears his name.

Work on the first inertial navigation systems started soon after WWII [15]. During the war, gyroscopes were used to stabilize guns on ships. The work on these stabilizers eventually led to work on inertial navigation systems that would provide aircraft on bombing runs the ability to navigate to and from a target without making radio transmissions, allowing the aircraft to remain undetected.

Work on these inertial systems progressed through the 1950s with various levels of success [15]. None of the early units met the requirements for accuracy set by the Air Force. Encouraged by the results from the development of airborne inertial systems, the Navy commissioned the development of submarine-based inertial navigation systems. These early units were large and expensive, and required precise machining during manufacture. The complexity and size made them impractical for smaller vehicles. However, the unique operational needs of submarines were well served by these units. Submarines are unable to navigate using traditional methods while submerged. Traditional methods at the time required some type of radio communication or celestial navigation. A submarine is intended to operate covertly and going to periscope depth in order to get a navigation fix could potentially reveal the submarine's position. Therefore, inertial navigation techniques are the only viable solutions for submarine navigation.

The early submarine navigation system was called the ship's inertial navigation system (SINS). A simplified drawing of this system is shown in Figure 2.1. This system was extremely accurate. SINS used a set of gyroscopes mounted on a movable platform. The gyroscopes maintained the platform level as the submarine maneuvered. This type of IMU is called a platform IMU. These IMUs required large spinning gyroscopes to accurately measure acceleration and rotation of the submarine. The first SINS was deployed onboard U.S. submarines during the 1950s [21]. The SINS was improved in the early 1980s by reducing the friction on the spinning gyros, thus allowing them to spin at very high speeds — 216,000 RPM [21]. These improved electrostatic gyro navigator (ESGN) units were initially deployed in 1983. New techniques for gyro measurement are currently under development. Gyros using interferometric fiber optic gyro (IFOG) technology currently cannot match the

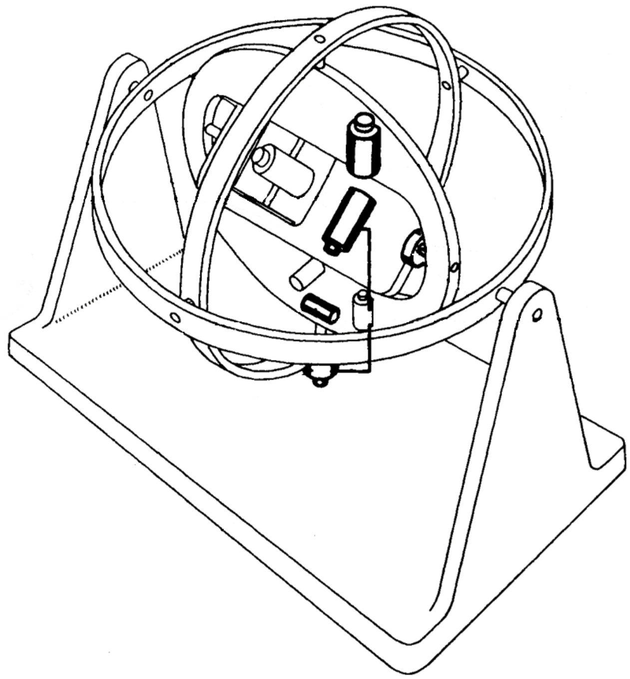


Figure 2.1: A simplified drawing of the ship's inertial navigation system[15].

accuracy of the spinning gyros, but IFOG has many other advantages. Gyros spinning at 216,000 RPM are difficult to manufacture, require high tolerances, and are very expensive [21]. IFOG technology has no moving parts. This makes the units smaller and easier to maintain.

The large, expensive measurement units and spinning gyros of the early INSs prohibited their use in many applications. Small, inexpensive, and reasonably accurate devices to measure acceleration and angular rate would be required before IMUs could be used in a wide range of applications. The development of such devices started around 1968 at Honeywell with the development of the first selectively etched silicon sensors [20]. By the 1990s, these MEMS devices were being developed and deployed in many areas, including inertial navigation.

The Wii controller is perhaps the most widely known use of IMU technology in a small consumer device. Today, IMUs are found in handheld phones, game controllers, and more. Commercial uses include all types of aircraft, vehicle control, and airbag deployment. Accelerometers are even used to detect if a laptop is in free-fall and prepare the hard disk in order to avoid damage to the disk surface on impact.

This thesis would not be practical without the development of MEMS devices. These devices provide accurate and fast acceleration and gyro measurements in a circuit package that is millimeters square. These newer devices include onboard processing and much higher levels of integration. The most cutting edge developments of 2011 include 3-axis accelerometers, 3-axis gyros, and onboard calculation of IMU characteristics, all in a single device. The potential market for such devices is huge, and it is certain that many new applications will be found for these highly integrated units.

The MEMS devices pack a lot into a small package, but the tradeoff is the noise level and accuracy of these devices. The gyro and accelerometer errors require calibration and removal of constant fixed-bias values. System level noise is high in the MEMS devices, limiting the accuracy of an IMU using them.

2.2 Inertial Measurement Key Concepts

Understanding inertial measurement techniques requires some background knowledge in a few key concepts. It is important to understand the two different types of IMUs available, platform IMUs and strapdown IMUs, and how they differ. Reference frames describe the various coordinate systems used in a typical INS. The measurement devices used in an IMU exhibit noise and bias effects that detract from the accuracy of the unit. Exact compensation is often not possible, so practical systems must often trade size and cost for overall accuracy.

2.2.1 Comparison of Global Positioning System and Inertial Measurement Units

Aircraft and vehicles can use the global positioning system (GPS) to track their position in real-time. Why would such a vehicle require an IMU? The answer to this question relates to how rapidly each system updates. The faster GPS devices update about 10 times per second. This seems fast until you consider the update rate required during high performance maneuvers, which can be hundreds or thousands of times per second. The required update rate of course depends upon the type of application. A typical IMU can update 8,000 times per second. In most cases, such high update rates are not required, but faster updates allow for filtering of the data to reduce noise, and actual updates to the vehicle control system may occur several hundred times per second.

2.2.2 Inertial Measurement Unit Types

There are two basic types of IMU: the platform IMU and the strapdown IMU.

The IMU used on submarines is a platform IMU. Platform IMUs use a set of gimbals, one for each axis, to keep the IMU platform fixed relative to the inertial frame of reference. What this means is that as the vehicle turns, dives, or rolls, the platform of the IMU stays flat. The advantage of the platform IMU is that the acceleration measured by the IMU is always in reference to the navigation frame.

In a strapdown IMU, the gyroscopes and accelerometers are fixed to the frame of the vehicle. A strapdown IMU is the most common type of system in use today. The reason is that they are easy to build, inexpensive, and small. All of the expensive and complex gimbals associated with a platform IMU are eliminated. The ramifications of this decision are that the orientation of the vehicle, relative to the navigation frame, must be tracked.

This project uses strapdown IMUs. The sensor readings in the body frame must therefore be translated to the local frame in order to track the position and orientation of vehicle. These frame conversions are typical of INSs and will be discussed in more detail in Section 2.3.1.

2.2.3 Coordinate Systems

Motion must be measured relative to some other point. An object sitting on the ground may not be moving relative to the earth's surface, but it is hurtling through space quite rapidly, and rotating at approximately one revolution per 24 hours. Relative to a particular point, an object may have motion that changes its position, also referred to as translation. An object may also have motion that changes its orientation, such that it rotates about one or more axes.

Practical navigation systems need to track position and orientation in 3D space. While it would be possible to use any available coordinate system, INS systems most commonly use cartesian coordinates. Position is given as a tuple of values, which represent the X, Y, and Z offsets from the reference origin. This frame of reference is described with a reference origin point and a cartesian coordinate system centered at that point.

The position and orientation of an object or vehicle is described with a body reference frame. This frame is also defined with a cartesian coordinate system. The origin of the coordinate system is located at the object center of mass. The axes are defined and fixed relative to the object axes.

2.2.3.1 Translation and Orientation

Two reference frames can be translated in position relative to each other. Given two reference frames A and B, the origin of frame B would be described as a tuple of position values in frame A. Similarly, the origin of frame A would be described as a tuple of position values in frame B. Figure 2.2 illustrates translation of a frame. A simple translation of frame B relative to frame A would maintain each of the three coordinate axes parallel. If frame B is also rotated relative to frame A, then a tuple of rotation angles is required.

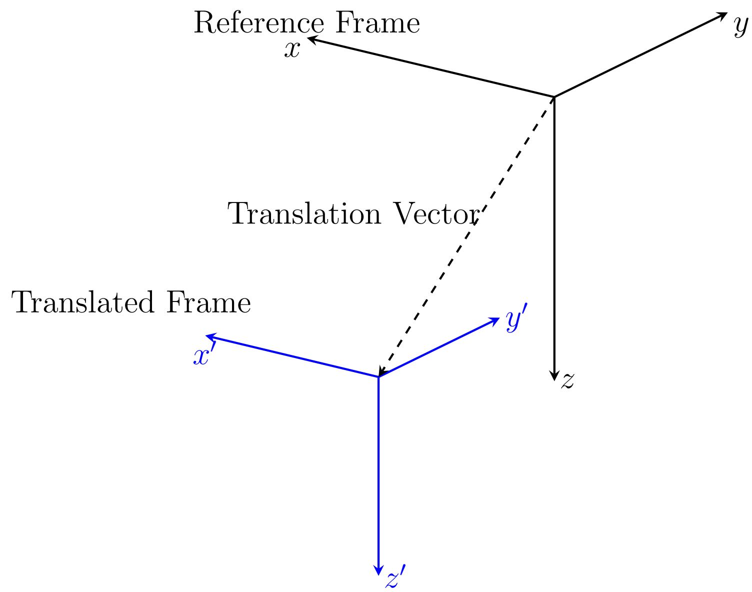


Figure 2.2: An illustration of a frame translated relative to the reference frame.

The orientation of frame B relative to frame A can be described as a sequence of rotations about the coordinate axes. The rotations are referred to as roll, pitch, and yaw. Roll is defined as rotation about the x-axis, pitch is defined as rotation about the y-axis, and yaw is rotation about the z-axis. Figure 2.3 illustrates angles used to describe a change in orientation.

For a body frame of reference, the x-, y-, and z-axes are defined to align with the vehicle axes in such a way that the roll, pitch, and yaw of the body frame make intuitive sense for the vehicle. Consider the right hand as the vehicle under consideration. The origin is defined as the center of the hand. The x-axis extends along the middle finger, the z-axis points downward away from the palm, and the y-axis completes the right-hand rule, pointing to the right. The body frame axes, superimposed on the right hand, are shown in Figure 2.4.

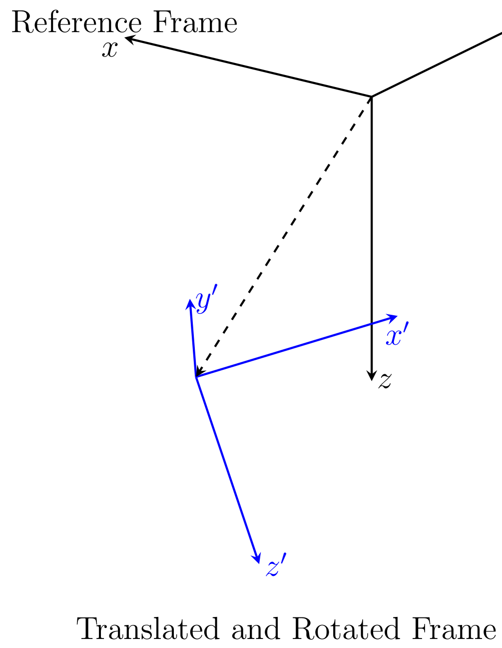


Figure 2.3: An illustration of frame translated and rotated relative to the reference frame.

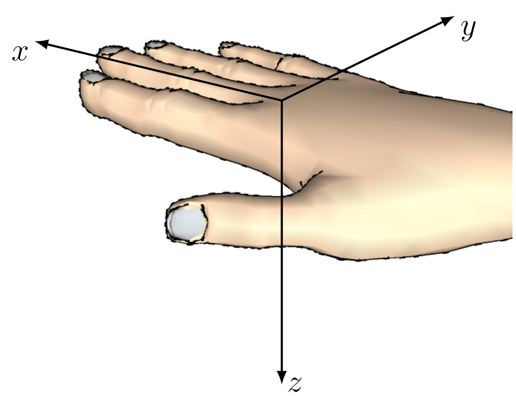


Figure 2.4: An illustration of the body frame axes for the right hand.

In order to illustrate reference frame orientation concepts, hold your right hand in front of you with the palm facing downward. According to the previous definitions, roll is defined as rotation of the wrist. Positive roll moves the pinkie downward, while negative roll moves the thumb down. Pitch is tilting of the hand upward (positive pitch) or downward, and yaw is a left or right flexing of the wrist. Mathematically, the roll, pitch, and yaw are defined by the three angles ϕ, θ, and ψ. When applying a set of orientation angles to a reference frame, it is typical to consider the angles applied in reverse order. Firstly, rotate frame B about the z-axis by the yaw angle ψ. Next, rotate B about the y-axis by the pitch angle θ. Finally, rotate the B frame about the x-axis by the roll angle ϕ. The angles ϕ, θ, and ψ are called Euler angles.

A complete specification of translation and orientation requires three values each, for a total of six values. These six values define the number of degrees of freedom (DOF) of frame B relative to frame A. The 6DOF thus defined are translation in three axes, and rotation in three axes.

2.2.4 Navigation Reference Frames

Navigation systems require well-defined reference frames. Reference frames associate the origin of a coordinate system with a well-defined location. The location may be fixed, as in the center of the earth, or it may be movable. The orientation of a reference frame must also be specified. The axes of a frame may be defined to always point to a particular location, such as the earth's prime meridian. A frame axis may also be defined to match the physical dimensions of a vehicle, or lie along a particular axis, such as the earth's spin axis. Navigation systems use a set of frames to measure and track the translation and orientation of frames relative to each other. Some frames also provide the basis for common navigation values, such as longitude and latitude.

An inertial reference frame is a frame that is not accelerating. The inertial frame may, however, experience uniform linear motion [17]. The reference frame origin may be placed at any location. The axes of the reference frame will follow the right-hand rule of the standard 3-axis coordinate system. The earth centered inertial (ECI) reference frame is defined with the origin at the center of the earth. The z-axis extends along the center of rotation of the earth, the x-axis points to the vernal equinox at a specified initial time, and the y-axis completes the right-hand rule[17]. This ECI reference frame does not rotate with the earth, hence a point on the earth's surface will exhibit a constant angular rotation and velocity relative to this frame. Figure 2.5 illustrates the ECI frame.

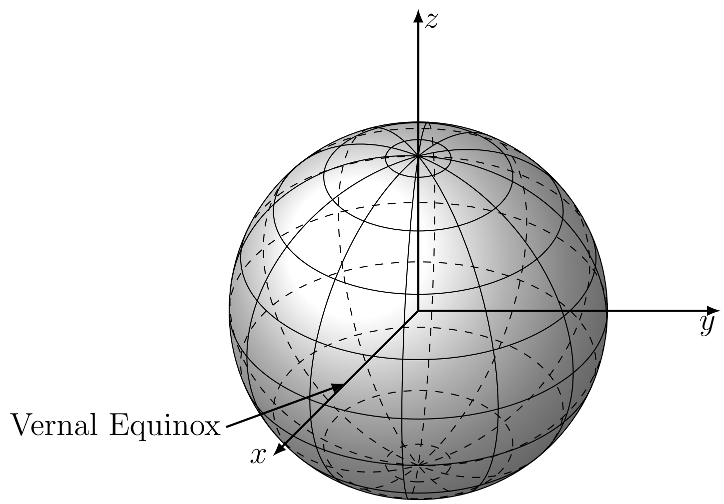


Figure 2.5: An illustration of the ECI coordinate system.

The astute reader may note that the center of the earth undergoes constant motion, but this motion is NOT linear. The earth actually rotates around the sun once every 365 $\frac{1}{4}$ days. The requirements of a particular system will determine how much motion is relevant, and how much can be ignored. A spacecraft traveling to the outer reaches of the solar system may need to take the rotation of the solar system into account. In this paper, the ECI frame can be considered as a true inertial frame.

The earth centered earth fixed (ECEF) reference frame is also defined with the origin at the center of the earth. This reference frame, however, will be defined such that the z-axis extends along the rotation axis, but the x-axis crosses the prime meridian. The y-axis is again defined to complete the right-hand rule. This reference frame is illustrated in Figure 2.6. A point on the surface of the earth will not move relative to the ECEF reference frame.

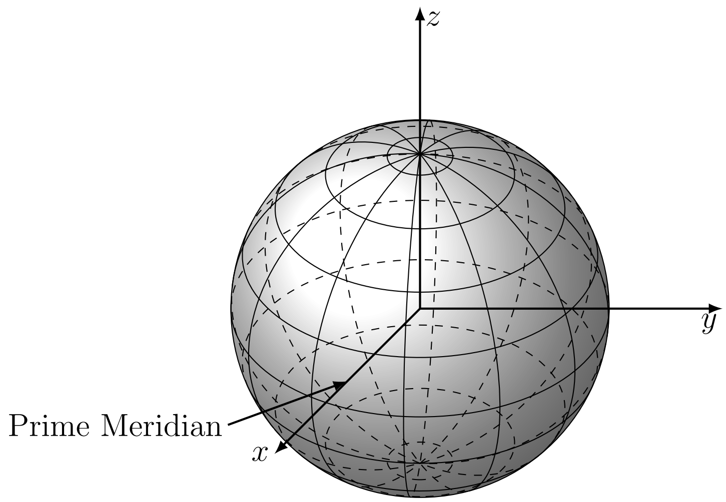


Figure 2.6: An illustration of the ECEF coordinate system.

The ECEF reference frame exhibits a constant angular rate relative to the ECI reference frame. This angular rate is defined as $\vec{\omega}^i_{ie}$. The subscript indicates that this is the rotation of the ECEF frame relative to the ECI frame. The superscript indicates that the value is represented in the ECI frame. The vector representation for this vector would be $\vec{\omega}^i_{ie} = [\begin{array}{ccc} 0 & 0 & \omega_{ei} \end{array}]^T$.

We generally think of navigation in terms of north, east, south, or west. The ECEF frame is not very convenient for calculating these navigation values. A vehicle traveling east in the ECEF frame would have a constant angular velocity around the z-axis, and varying amounts of translation in the X and Y axes, depending on the longitudinal position at a given time. Navigation frames are defined to better support more typical north, east navigation parameters.

2.2.4.1 Geographic and Geocentric Frames

The geographic and geocentric frames are closely related. Both move with the navigating vehicle. The geographic frame is aligned such that the origin sits on the surface of the earth's geoid directly below the navigating vehicle. The z-axis points down and is normal to the earth's surface. The geocentric frame also follows the navigating vehicle, but the z-axis points to the center of the earth. For both frames, the x-axis points toward true north, and the y-axis points east [17].

2.2.4.2 Local Navigation Frames

For local navigation, a local geodetic or tangent frame is defined. This frame differs from the geographic frame in that its origin is fixed at some point on the earth's surface. The frame serves as a convenient reference point for the system. The exact reference point may be selected according to the application. The point may be the end of a runway, a particular city in North America, or it may be the center of the desk you are working on. Since the tangent frame does not move, vehicle motion and position may be measured relative to this fixed point. The axes of the tangent frame are generally defined to be suitable for navigation. As such, the z-axis points downward, the x-axis points to magnetic north, and the y-axis points east. This frame is often referred to as the north east down (NED) reference frame. Common navigational thinking would mean that we travel north in the positive direction, or east in the positive direction. The choice of z-axis direction is used to make the x-axis and y-axis consistent with common navigational thinking and to maintain the right-hand rule.

2.2.4.3 Instrument and Body Frames

A strapdown IMU measures acceleration and angular rate relative to the physical instrument. The axis of the measurement is the instrument axis, and is referred to as the instrument frame. The vehicle reference frame is called the body frame. Ideally, the instrument frame would be perfectly aligned with the vehicle axes, but generally some variation in the alignment is present. The offset between the instrument frame and the body frame is usually a fixed X,Y,Z coordinate offset, and a fixed set of ϕ, θ, and ψ orientation angles. This offset is based on the placement of the instrument frame relative to the vehicle's center of mass. The vehicle's center of mass is the origin of the body frame.

2.3 The Mathematics of Inertial Navigation

A full presentation of inertial navigation systems requires a background in advanced mathematics including calculus, linear algebra, numerical methods, stochastic processes, and statistics. Most textbooks that cover inertial navigation systems provide several chapters of review on these topics, as well as an appendix or two for reference. A detailed review of these topics is beyond the scope of this thesis, therefore this document will present only the subset of INS techniques used in this project.

2.3.1 Reference Frame Transformations

The reference frames used in this project are the body and local navigation frames. The local navigation frame will be used as the reference frame and the body frame will move relative to this frame. A transformation from body frame to the local reference frame involves two fundamental operations. The first operation is a rotational transformation of the body frame coordinates to the reference frame, then a translation to the reference frame origin.

There are several methods that exist for transforming coordinates in one frame to a new frame. Two of these are the direction cosine matrix (DCM) and quaternions. The DCM transformation is much easier to visualize than the quaternion. Unfortunately, the DCM suffers from a singularity when the x-axis is pointed vertically. Recall that θ is the pitch angle. The DCM transformation is undefined when the x-axis is vertical since $cos(\theta) = 0$, and it appears in the denominator of a fraction. Quaternions resolve this singularity problem, but at the expense of clarity. Numerous texts are available on the subject of quaternions, however, the DCM was used during this project and is the only technique that will be described.

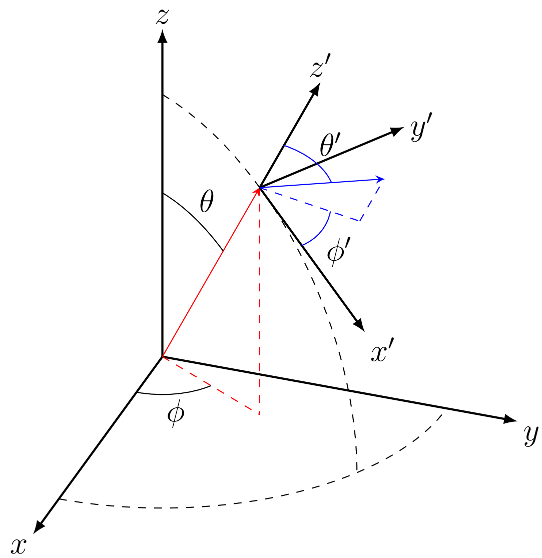


Figure 2.7: An illustration of rotation and translation of coordinate system.

2.3.1.1 Vector Notation

Let a point in frame B be described by the tuple $\vec{P}_B = [\ x \quad y \quad z\ ]^T$, where the subscript indicates that this point is relative to frame B. The superscript T denotes a vector transpose, and $\vec{P}_B$ describes a column vector

$$[\ x \quad y \quad z\ ]^T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{2.1}$$

Let $\vec{P}_A = [\ x' \quad y' \quad z'\ ]^T$ describe the identical point in frame A coordinates. We seek a transform matrix such that $\vec{P}_A = R\vec{P}_B$. The dimensions of matrix R must be 3x3. The matrix that we seek is the DCM. The DCM matrix to transform frame B coordinates to frame A coordinates is defined as R_B^A. The transform is then written as $\vec{P}_A = R_B^A \vec{P}_B$. The subscript and superscript indicate that R is a rotation matrix that rotates vectors in B to vectors in A.

2.3.1.2 Direction Cosine Matrix Derivation

We derive the DCM by performing coordinate transforms on the frame axes one at a time. Let us begin with frame A and frame B coincident and sharing the same origin. Let frame B rotate about the common z-axis by a yaw angle ψ. We call this new intermediate frame B1 and the vector $\vec{P}_{B1}$. The transformation matrix to move a point from $\vec{P}_{B1}$ to $\vec{P}_A$ is

$$R_{B1}^{A} = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.2}$$

The transformation is illustrated in Figure 2.8, where the frame B1 axes are represented by the single primes.

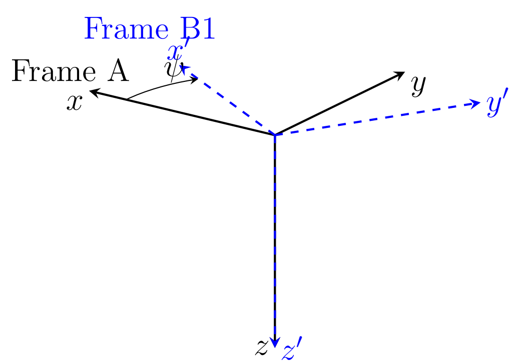


Figure 2.8: Coordinate system rotation about the z-axis with angle ψ.

Next, we rotate the frame B1 about the y'-axis through the pitch angle θ using

$$R_{B2}^{B1} = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}. \tag{2.3}$$

The transformation matrix from frame B2 to frame B1 is illustrated in Figure 2.9.

Finally, we can rotate the frame B2 about the x''-axis through the roll angle ϕ using

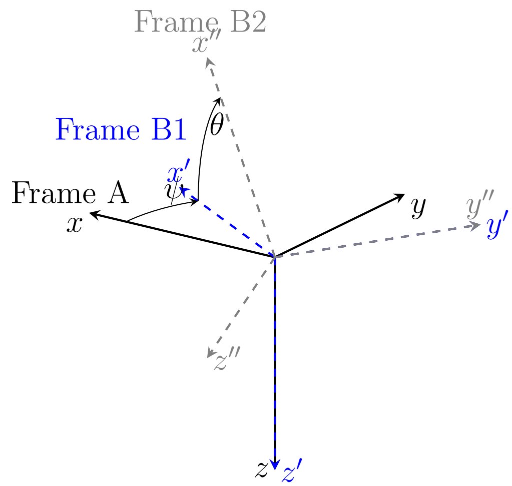


Figure 2.9: Coordinate system rotation about the z-axis with angle ψ and the y'-axis with angle θ.

$$R_B^{B2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix}. \tag{2.4}$$

This final rotation is illustrated in Figure 2.10.

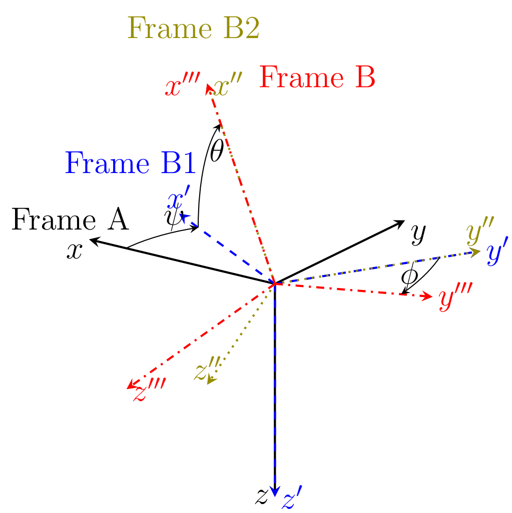


Figure 2.10: Coordinate system rotation about the z-axis with angle ψ, the y'-axis with angle θ and the x''-axis with angle ϕ.

Fortunately, the matrix operations can be chained together, so the three matrices can be applied in sequence to perform all three transformations in one series of operations:

$$\vec{P}_A = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \vec{P}_B. \tag{2.5}$$

Using elementary matrix operations, the three transformation matrices can be combined into a single matrix:

$$R_B^A = \begin{bmatrix} c(\psi)\,c(\theta) & -s(\psi)\,c(\phi) + c(\psi)\,s(\theta)\,s(\phi) & s(\psi)\,s(\phi) + c(\psi)\,s(\theta)\,c(\phi) \\ s(\psi)\,c(\theta) & c(\psi)\,c(\phi) + s(\psi)\,s(\theta)\,s(\phi) & -c(\psi)\,s(\phi) + s(\psi)\,s(\theta)\,c(\phi) \\ -s(\theta) & c(\theta)\,s(\phi) & c(\theta)\,c(\phi) \end{bmatrix} \tag{2.6}$$

where cos and sin operators are represented by c and s respectively.

The DCM matrix is referred to as a rotation matrix, since it rotates coordinates in one frame, along all three axes, to a new frame. The rotation matrix is valid for a particular set of roll, pitch, and yaw angles. For moving systems, the rotation matrix elements will be changing, possibly at a very high rate.

2.3.1.3 Direction Cosine Matrix Properties

The DCM matrix has several useful properties. These properties are defined according to matrix and linear algebra. The properties are stated here without proof.

The DCM matrix is an orthogonal matrix. A square matrix is orthogonal if the matrix transpose equals the matrix inverse. Hence, ${R_B^A}^{-1} = {R_B^A}^T$, where the superscript T indicates matrix transpose.

The DCM matrix can be applied in sequence to transform between multiple reference frames. In mathematical terms, $R_D^A = R_B^A R_D^B$. This is a most convenient feature of the DCM, and can be used to translate between multiple frames of reference, as long as the DCM for each pair of frames is known.

2.3.2 Direction Cosine Matrix Updates

The DCM must be continuously updated as the body frame moves relative to the reference frame. In an inertial system, it is not possible or practical to measure the roll, pitch, and yaw angles directly. The navigation system must perform an initial calibration of the angles, and then the inertial system must continuously update the angle values. The INS uses gyroscope and accelerometer inputs to update these values.

The rotation of frame B relative to frame A as projected onto the frame A axis is defined as the vector value $\vec{\omega}_{AB}^A$. This term is a vector quantity that represents the *rate of change* of the three Euler angles ϕ, θ, and ψ, respectively

$$\vec{\omega}_{AB}^A = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}. \tag{2.7}$$

The rotation matrix given in Equation 2.7 can be represented as a skew-symmetric square matrix

$$\vec{\omega}_{AB}^A \times = \begin{bmatrix} 0 & -\omega_{ABz}^A & \omega_{ABy}^A \\ \omega_{ABz}^A & 0 & -\omega_{ABx}^A \\ -\omega_{ABy}^A & \omega_{ABx}^A & 0 \end{bmatrix}. \tag{2.8}$$

The skew-symmetric matrix defined in Equation 2.8 is particularly well suited for updating the DCM matrix. For small values of the ω matrix, the relationship

$$\dot{R}_B^A = R_B^A \vec{\omega}_{AB}^A \times \tag{2.9}$$

holds true.

The rate of change of the rotation matrix is easily calculated by multiplying the current rotation matrix by the skew-symmetric form of the rotation rate vector. The rotation rate vector is easily constructed from the INS gyroscope outputs.

Equation 2.9 is a continuous time equation, but the sensors for a typical INS provide updated readings at a periodic rate. The update rate is generally a fixed time period, usually much less than one second. For example, a system that updates 100 times per second will have an update period of 10 ms. The sampled nature of the measured values requires that we use a discrete time approximation to the continuous time equations.

Gyroscope sensors measure angular rate. In order to determine the change in the angle, the rate must be integrated over the sensor time period. Since we only know the angular rate values at distinct points in time, we must estimate the rate value between the time periods.

There are a number of well-known techniques for estimating the intermediate values of distinct measurements. The simplest is the average. Given two measured values, ω_1 and ω_2 taken at times t_1 and t_2, respectively, then $\omega_{avg} = \frac{(\omega_1 - \omega_2)}{2}$. If the ω_n values represent the rate of change of angle θ, then $\Delta\theta = \omega_{avg} * \Delta t$. More complex estimating algorithms may be used to improve the performance of a system, but that topic is beyond the scope of this paper.

The quantity

$$\vec{\Delta\Omega} = \begin{bmatrix} \Delta\phi & \Delta\theta & \Delta\psi \end{bmatrix}^T \tag{2.10}$$

is a vector quantity that represents the change in the Euler angles along all three of the body frame axes, ϕ, θ, and ψ. The DCM is updated using the skew-symmetric form of the matrix with

$$R_B^{A^m} = R_B^{A^{(m-1)}} * \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & -\Delta\psi & \Delta\theta \\ \Delta\psi & 0 & -\Delta\phi \\ -\Delta\theta & \Delta\phi & 0 \end{bmatrix} \right). \tag{2.11}$$

The superscript on $R_B^{A^{(m-1)}}$ indicates the previous value of the DCM matrix, while $R_B^{A^m}$ represents the updated value. The DCM matrix is updated at each time period of the IMU.

2.3.3 Accelerometer Updates

Basic Newtonian physics declares that velocity is the time rate of change of position, and acceleration is the time rate of change of velocity. Equivalently, acceleration is the second derivative of position. In mathematical terms, $a = \frac{dv}{dt}$, $v = \frac{dp}{dt}$ and $a = \frac{d^2p}{dt^2}$.

The navigation system seeks to track position along the local navigation frame. The body acceleration values are first rotated to the local navigation frame using the DCM rotation matrix. The rotated acceleration values are used to update the position of the system in the local frame. The first integration of the acceleration yields the updated velocity values. The second integration yields the position values. A 3-axis accelerometer provides values in all three coordinate axes, making it possible to update the navigation equations in 3D space.

2.3.4 Inertial Measurement Unit Initial Alignment

When power is first applied to the IMU the orientation of the system, the current velocity, and the current position are all unknown. A navigation system may use any number of external references to determine the precise values of these parameters. This project uses a very simple approach.

For this thesis, only the relative sensor position is important. The system is therefore free to initialize the position to $\vec{P} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$. The design requires that the sensors be held still during the initialization period, so the velocity vector can also be initialized to $\vec{V} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$.

On the surface of the earth, the acceleration due to gravity is equal to one gravitational constant, or $1G$. Gravity is also a directional vector. In the navigation frames used for this system, the gravity vector points downward toward the center of the earth. The local navigation frame is always defined with the z-axis parallel to the gravity vector, hence for the local navigation frame we have $\vec{A_L} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T G$. Here, the subscript L indicates that this is the acceleration vector relative to the local navigation frame. When the IMU is held still, the gravity vector can be used to perform a course alignment to determine the starting orientation of the system. Note that this does not require that the system be level, just that it be held still.

When the system is held still, the measured acceleration in the body frame will be $\vec{A_B} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T$. The magnitude of $\vec{A_B}$ will be 1G. We seek to determine the DCM matrix that will transform $\vec{A_B}$ into $\vec{A_L}$.

2.3.4.1 DCM Course Alignment

The course alignment procedure used for this project was taken from the book *Strapdown Analytics*, by Paul G. Savage. The following is a brief summary of the technique documented in that book.

Let u^B_{xL}, u^B_{yL}, and u^B_{zL} be unit vectors along the L frame x-, y-, and z-axes projected on the B frame axes [29]. In matrix form, we have

$$R_B^L = \begin{bmatrix} \left(u_{xL}^B\right)^T \\ \left(u_{yL}^B\right)^T \\ \left(u_{zL}^B\right)^T \end{bmatrix}. \tag{2.12}$$

We know that the local frame is level, and the z-axis is parallel to the gravity vector, but opposite in sign, therefore we know that $\vec{A}_{Lz} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$. The third row of R_B^L must conform to the relationship $\begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T = \left(u_{zL}^B\right)^T \left(\vec{A}_B\right)$. Hence, $\left(u_{zL}^B\right) = -\left(\vec{A}_B\right)^T$. In words, the third row of the DCM is always initialized to the negative transpose of the measured body frame vector.

The DCM matrix must always have the properties that the rows and columns are orthogonal to each other, and the matrix must always have unity magnitude. As long as these criteria are maintained, the first and second row values are arbitrary [29]. As long as the x-axis of the body frame is not vertical, the following procedure works. Let the first column of the second row = 0, or $C_{21} = 0$. Since we know that rows two and three are perpendicular, their dot product must equal zero, hence:

$$C_{21}C_{31} + C_{22}C_{32} + C_{23}C_{33} = C_{22}C_{32} + C_{23}C_{33} = 0. \tag{2.13}$$

Since $C_{21} = 0$, Equation 2.13 can be solved by setting $C_{22} = KC_{33}$ and $C_{23} = -KC_{32}$. K is then chosen to normalize row two to unity, and we have $K = \frac{1}{\sqrt{{C_{32}}^2 + {C_{33}}^2}}$. Row two therefore becomes represented by

$$C_{21} = 0, \quad C_{22} = \frac{C_{33}}{\sqrt{{C_{32}}^2 + {C_{33}}^2}}, \quad C_{23} = \frac{-C_{32}}{\sqrt{{C_{32}}^2 + {C_{33}}^2}}. \tag{2.14}$$

Row three can be initialized easily from the values in rows two and three. Since row one must be orthogonal to both rows two and three, row one equals the cross product of rows two and three, and we have

$$\begin{aligned} C_{11} &= C_{22}C_{33} - C_{23}C_{32} \\ C_{12} &= C_{23}C_{31} - C_{21}C_{33} \ , \\ C_{13} &= C_{21}C_{32} - C_{22}C_{31} \end{aligned} \tag{2.15}$$

completing the course alignment of the DCM.

2.3.5 Error Sources for Inertial Measurement

The previous discussions have assumed perfect measurements. Unfortunately, the data values collected from the accelerometers and gyros contain appreciable amounts of noise. Furthermore, the devices also contain fixed offset values. Navigation systems that require positional accuracy must pay close attention to the noise and offset values, and use outside sources of information to compensate for drift values. A GPS system is a common outside reference for such purposes. This project used very simplified versions of the full set of navigation equations. The simplified equations overpower the noise issues, allowing them to be ignored. Future iterations of this project would require that these issues be addressed.

2.4 Summary

This chapter provided a short history of inertial navigation and some background on the two primary types of INS: platform and strapdown. Coordinate systems were defined, as well as the common navigation reference frames. This chapter outlined only a small subset of the math behind inertial navigation systems. The math presented covered the techniques used during this project only. Many references exist with exhaustive details about the INS equations, their derivation and use. The primary references used for this chapter were *Aided Navigation: GPS with High Rate Sensors*[17] and *Strapdown Analytics, Part 1* [29].

CHAPTER 3

HARDWARE AND SOFTWARE TOOLS

A design project such as the GyroGlove requires a large infrastructure. Schematic capture and board layout software is required for printed circuit board (PCB) design. Component libraries are required to describe schematic symbols, board layout footprints and links to component vendors. Board assembly requires soldering tools and skills. A variety of software tools are needed for testing, data capture, processing, and visualization. This chapter documents the tools and technologies used during this project. Implementation details of the project are mentioned only briefly. Details about the implementation can be found in Chapter 4.

3.1 Board Design Tools

The PCBs for this project were designed using Altium Designer™ summer 2009, student edition. Altium Designer™ is a very powerful board design package, but with power comes complexity. This project required an appreciable amount of setup for the Altium Designer™ tools.

Before the design can be entered into the schematic tool, each of the design components must be added to the library. A component consists of a schematic symbol, a layout footprint, and other important part data. Important part data includes manufacturer information, part ordering information, and links to design data sheets.

The schematic symbol identifies the logical connections to the device. The symbol is typically drawn with standard engineering graphics. The connections in the schematic symbol are logical, and do not necessarily represent the physical connection to the part. The layout pattern provides the physical dimensions of the part as they will appear on the printed circuit board. A particular type of device may be available in multiple packages, and each package will have a different physical pattern. The device component, which includes the schematic symbol and the physical footprint, completes the association between logical schematic pins and the physical pins on the printed circuit board.

Components are generally available in a range of packages. The choice of the best package for a part is driven by price, availability, size, and ease of assembly. Availability is a key factor — it is very difficult to build a board with parts that you cannot buy. Small packages use less board space, but may make assembly difficult. Larger parts are easier to assembly, but may increase the size of the circuit board beyond the design limit.

Digikey was the primary supplier of components for the GyroGlove project. The Digikey website was used to ascertain the availability and cost of parts and the ideal package. The components were added to the library based on the chosen parts and then ordered to ensure that they would not be out of stock when needed.

Some components, such as resistors and capacitors, have many different values but the same schematic symbol and layout footprint. Resistors come in standard sizes, so it is not necessary to make a unique footprint for each component. Since there are hundreds of resistors available to choose from, a database library is the preferred solution for passive components.

A local MySql™ database was used to store component information for the passive parts libraries. The part information was gathered from the Digikey™ website. Website data for each part was captured into an Excel spreadsheet. The spreadsheet data was then uploaded into the database using a Python™ script. The database setup and linking to the Digikey™ site made the parts order easy and accurate. A bill of materials (BOM) was generated from Altium Designer™ and used to order all required parts.

The fully designed PCBs were manufactured at Advanced Circuits™. There were two runs of boards, but the second run was combined with boards from other projects. The IMU board cost was $50 for all six boards. The second run cost a total of $522, with $150 of that cost for the GyroGlove project.

The accelerometers cost $10 each, while the gyroscopes cost $15 each. The boards and components brought the total cost for the six IMU boards to $200, not counting wire and small components such as resistors and capacitors. The component cost for the controller board was about $40, bringing the total controller board cost to about $190.

Additional items such as wire, connectors, and the gloves for the project added additional cost. Wire for the project was purchased in spools, where only a small amount of the wire purchased was needed. The total estimated cost of the project was approximately $450.

3.2 Board Assembly Tools

The board assembly process required an entire day. A work area was prepared with all of the parts inventoried and organized. A solder paste mask was used to apply the solder paste to the boards. The part assembly listings provided a reference for the part numbers required for each component, and the location on the board. A microscope was used to help place small components on the boards, and to align the pads of larger components. The rest of the process was a slow and tedious hand placement of each component on the board.

A custom modified toaster oven was used to melt the solder paste during the reflow process. Hand soldering was used to repair components that were not soldered well, and to add the components to the back side of the boards.

The major equipment items used during the assembly process included:

- Pace™ MBT-350 soldering station with hot air pencil, solder extractor, hot tweezers, and fine tip soldering pencil.
- Amscope™ stereo zoom microscope with 60 LED light ring.
- Black and Decker toaster oven, with modified controller driven by MATLAB™ software.
- Molex wire crimpers.

3.3 System Testing Equipment

The assembled IMU and controller boards were tested using bench top test equipment, which included:

- Two Agilent™ U8002A power supplies
- Fluke™ 8645A precision voltmeter
- Tektronix™ TDS 3032B digital oscilloscope
- Zeroplus™ LAP-16128U USB logic analyzer

The power supplies were used to carefully apply power while limiting the maximum current to the boards. This technique avoids damaging components when there are power supply shorts on the board. The boards were carefully powered up in this manner until all of the shorts were isolated and repaired. The fluke voltmeter was also used to verify the power supply voltages, and check for shorts.

The oscilloscope was used to view analog signals. The boards were checked to ensure that noise levels were within an acceptable range. The processor on the board uses an internal oscillator circuit. The clock output was routed to a pin and checked with the scope to ensure that the frequency was correct.

3.4 Firmware Development Tools

The processor used on the controller board is an Atmel™ ATXmega128A1. The processor was programmed using the C++ language. The Atmel™ development environment for these processors is AVR Studio™. The development environment runs only on Windows™ PC's, but the primary development environment for this project was a Mac™ computer. The CrossPack AVR™ development environment was used on the Mac™.

The logic analyzer was used extensively during firmware development. The firmware design includes multiple parallel processes, which present special challenges for debugging. External pins on the processor were set within the firmware to mark specific occurrences within the logic. The internal logic checkpoints were compared with external signal values to track down firmware bugs and resolve timing issues.

The logic analyzer automatically recognizes and decodes inter-integrated communications (I^2C) and RS-232 transactions. The decoder greatly simplifies the task of verifying the data to and from the controller from the computer. The I^2C protocol decoder is a huge help when debugging communication issues with the IMU boards.

3.4.1 Processor Configuration and Debug

AVR Studio™ version 4.0, was used for firmware development on this project, and AVR Studio™ is freely available from Atmel™. The software requires a Windows™ PC for operation and connects to the joint test action group (JTAG) in-circuit emulator

(ICE) MkII using a universal serial bus (USB) cable. AVR Studio™ is used to compile updated source code and to download the code to the controller board.

The ATXMega128A1™ processor is configured using an Atmel™ JTAG ICE MkII programmer and debugger. The MkII uses the JTAG protocol for configuring and for debugging.

3.5 Software Development Tools

The GyroGlove uses software to transfer data to the PC, visualize the results on 2D graphs, perform the INS calculations, and visualize the motion of the glove in a virtual 3D environment. Following are the software tools used:

- The Python™ programming language
- MATLAB™
- Panda3D™

The glove software system is written in three distinct pieces. The first piece is the data capture server. The server is written in Python™ and located in the file GloveServer.py. The server captures data from the hardware and makes the data available to client code. The second piece is written in MATLAB™, and is called GloveGui.m. The MATLAB™code provides a GUI with graphs for the raw accelerometer, gyro, velocity, and position data from one IMU unit. The MATLAB™ code also includes all of the IMU calculations. The final piece is the 3D visualization part, written in Panda3D™ and called Glove3D.py.

3.5.1 Python™ Language

The Python™ programming language is a popular scripting language among software engineers. It is used for a wide variety of applications, such as rapid prototyping, GUI development, system administration, and many others. The language includes a rich set of data structures that make processing and manipulating captured data very easy. Python™ has a large number of community supplied library modules that are freely available for download. These modules provide capabilities for serial communication, GUI design, numerical analysis, 2D and 3D charting, and much more.

Python™ is available on most computer platforms, including Mac™, PC, Linux, and Unix. Scripts that perform hardware access, such as with the serial port, sometimes require platform specific code. Python™ is free to download and several outstanding free development environments, such as the Eric IDE™, exist. Komodo™, from Activestate™, was used for this project.

Python™ version 2.6 was used for this project. The latest version of the Python™ 2.x series is 2.7. Python™ also has a 3.x series but the 3 series made some language changes that are not backward compatible. The code for this project would require some modification to run under the 3.x series.

3.5.1.1 Python™ Libraries

Much of the power in Python™ lies in the availability of free, open-source library modules. The standard install of Python™ includes many pre-packaged modules, but some additional libraries must be installed to support this project. The following additional Python™ libraries are required:

- socket
- PyQt4
- numpy
- scipy
- pyserial

The socket library is required in order to communicate between the different pieces of the glove system. GloveServer, GloveGui, and Glove3D all transfer data via the sockets interface.

The pyserial library is required for the GloveServer. The software drivers for the USB device on the glove controller create a virtual COM port on the host. Python™ uses the pyserial library to connect to this port and transfer data. A Mac™ computer running OSX 10.6 Snow Leopard was used while developing this project. The GloveServer should work on a Windows™ PC, but the python files would require a few modifications to work with a Windows™ COM port.

The GUI library used for this project is PyQt 4.0. This library is based on Qt™, which is a cross platform C++ library. PyQt is a Python™ wrapper around the Qt™ C++ libraries that enables full support of the Qt™ GUI development platform from Python™.

The default installation of Python™ is able to perform numerical calculations. The numpy and scipy libraries are enhanced numerical libraries that improve calculation speed. They also provide matrix manipulation capabilities, much like MATLAB™. Numpy is used during the python data captures to transform the captured data from instrument frames into body frames. Scipy is used only to output MATLAB™ files in .mat format. The code to output .mat files is optional, making the scipy library optional also.

3.5.2 MATLAB™ Environment

MATLAB™ is a well-known matrix library environment and a standard tool at most universities and many businesses. MATLAB™ is perhaps the easiest and most powerful software tool available for performing mathematical manipulations, especially where matrix operations are involved. It is the perfect tool for developing INS algorithms. All of the INS algorithms were implemented in MATLAB™ class files.

The MATLAB™ GUI receives results from the Python™ GloveServer and displays selected data on a set of 2D graphs. The calculated positions and orientations of the hand and fingers are sent to the Panda3D™ Glove3D server for 3D visualization.

3.5.2.1 Mex Files

MATLAB™ mex functions are compiled functions written in C or C++. These functions can be called from MATLAB™ just like a .m file function. Mex function development requires a working C++ compiler on the MATLAB™ system. If the MATLAB™ environment is properly configured, then it is a simple matter to build a MATLAB™ mex function with the command:

```
mex <filename>
```

which will compile the source files into an executable MATLAB™ command.

3.5.3 Panda3D™ Library

Panda3D™ is a freely available 3D gaming engine. The engine was originally built for a Disney™ movie project. Panda3D™ is currently maintained by Carnegie Mellon University and provided as free, open-source software. The key motivators behind using the Panda3D™ environment were the performance of the engine and the use of Python™ as the software interface.

Panda3D™ was a great discovery. The author owes credit to his brother, David, for making him aware of the Panda3D™ project. Panda3D™ provides the ability to render a 3D virtual hand, and the ability to update the position of the hand and fingers using X,Y,Z coordinates and 3-axis rotations. Panda3D™ also includes the ability to specify rotations in quaternions. Panda3D™ was the perfect tool for this project.

Not only is Panda3D™ a great graphics engine, but it is a cross platform tool as well, with Mac™ and Windows™ versions available. The best part is that Panda3D™

provides a Python™ application programming interface (API). The Panda3D™ Python™ API eliminated the need to learn yet another language. Panda3D™ provided more than enough capability to easily implement the visualization portion of this project.

CHAPTER 4

GYROGLOVE SYSTEM DEVELOPMENT

The glove developed for this project includes both accelerometers and gyroscopes combined into a complete IMU. None of the gloves found during the literature search included gyroscopes, hence the term GyroGlove seems appropriate. The GyroGlove consists of a controller board, six IMU boards, and a set of software programs on the host computer. The host computer software is comprised of three main components. The first component is a Python™ script to connect to the hardware through the USB interface, retrieve the IMU data, and act as a socket server for the other components. The second component is written in MATLAB™. This component retrieves data from the socket interface and performs the IMU calculations for each IMU in the glove. The final component is the Panda3D™ server component that is used for visualization of the calculated glove positions and orientations. This component uses a socket interface as well.

4.1 GyroGlove Design

The GyroGlove incorporates 6DOF IMUs, with one mounted on each finger, one on the thumb, and another on the back of the hand. The IMUs are wired to a controller unit mounted on the back of the hand. The controller unit is attached to the host computer using a USB cable. The controller uses a microprocessor to collect data

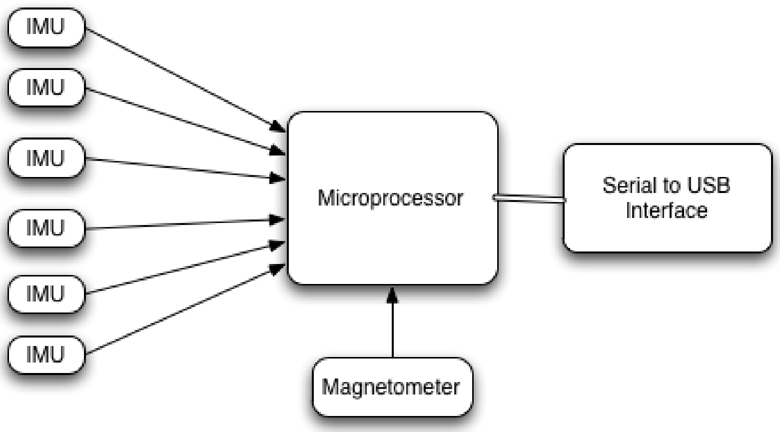


Figure 4.1: Block Diagram of GyroGlove

from the six IMUs. The controller assembles the collected data into data packets and sends the packets to the host.

Each of the six IMUs are wired to the main controller with four wires. Two wires provide power and ground connections, with the remaining two implementing a two-wire serial I^2C protocol. A block diagram of the GyroGlove is shown in Figure 4.1. The communication from the glove to the host computer uses a serial-to-USB interface device. This device is capable of transmitting data in both directions. The USB serial converter has sufficient bandwidth to transfer sensor data from all six IMUs at a rate of 200 Hz.

4.1.1 Data Handling

The controller board assembles the captured IMU data into binary packets. The data packets are automatically sent to the host PC at the sample rate. The system was tested with sample rates as high as 200 Hz. The controller was most reliable with sample rates of 150 Hz or less.

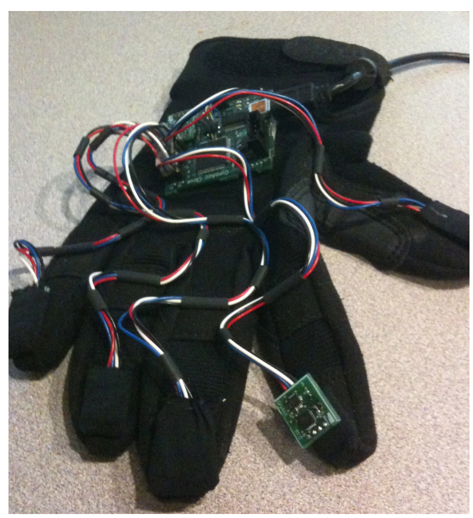
On the host computer a Python™ GloveServer program captures the data from the glove. This program also applies the rotation matrices shown earlier. The GloveServer program listens for connections on a network socket interface. A program written in

Python™, MATLAB™ , or any other language supporting network sockets can connect to this interface and retrieve the glove data. For this thesis, a MATLAB™ program captured the results, displayed some of the results on 2D graphs, and sent commands to a 3D visualization tool for real-time display of the calculated hand and finger orientations.

4.1.2 Glove Versions

The first glove version used thin 30 gauge solid core wire to connect the sensors to the controller board. Also, to save time, the boards were glued directly to the glove. After more consideration, gluing the boards to the glove seemed like a bad idea. The solid core wires used on the first glove version are prone to breakage. The wires break inside the insulation, causing intermittent failures that can be quite difficult to track down. The Version 1.0 glove is shown in Figure 4.2a.

(a) GyroGlove Version 1.0



(b) GyroGlove Version 2.0

Figure 4.2: The GyroGlove

The Version 2.0 glove, shown in Figure 4.2b, uses 30 gauge 7-strand wire. This wire has a thicker insulation, but is much more flexible. A new glove was selected for Version 2.0. The new glove is somewhat bulkier, but also includes pads on the fingertips that make assembly of the glove easier. The author's wife sewed small black pouches to house the IMUs on each finger. The pouches were then secured to the glove using glue, and the IMUs were inserted into the pouches. The material for the pouches has some stretch to it, and in most cases the IMUs fit snugly. Some of the pouches required a few stitches to secure the unit inside.

4.2 Hardware Design

The GyroGlove hardware consists of two custom designed PCBs. The first board is an IMU board used to capture the accelerometer and gyroscope motion of the fingers, thumb, and hand. The second board is the controller board. The IMU boards were designed to be roughly the size of a fingernail so that they could be attached to the distal phalanges of each finger. The controller board was designed for mounting on the back of the hand.

The controller board for this project needed to interface to five remote IMU boards, and include an onboard IMU as well. These requirements eliminated off-the-shelf solutions. The controller board required the following capabilities:

- Interface to five remote IMUs,
- Onboard IMU,
- USB interface,
- Power regulators,
- Processor capable of controlling the IMUs and USB interface.

4.2.1 Inertial Measurement Units

The IMUs for the GyroGlove are custom designed and built PCBs. Figure 4.3 shows a 3D Altium Designer™ rendering of the IMU board physical layout.

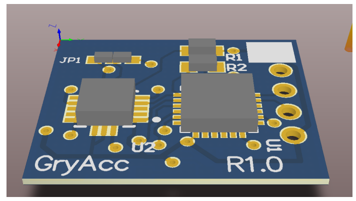
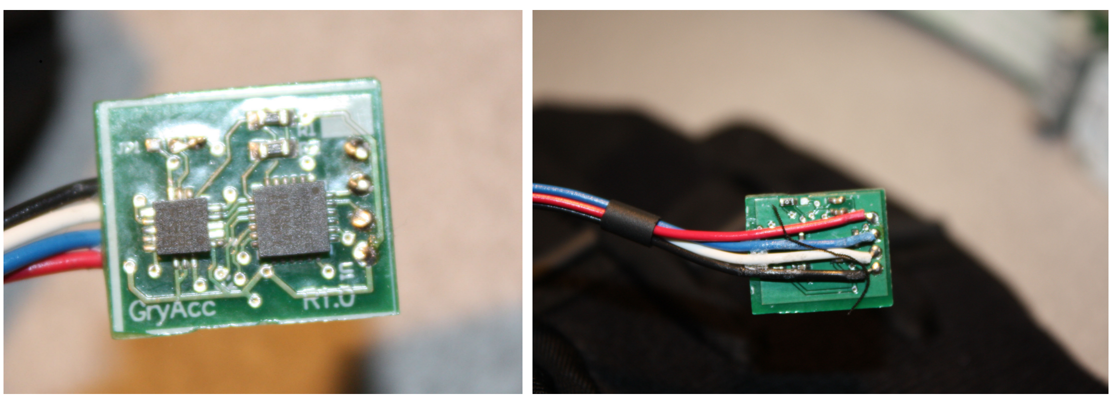


Figure 4.3: 3D PCB Render of the IMU Board

The IMU board, shown in Figure 4.4a, measures 12.7 mm (0.5”) by 15.2 mm (0.6”). The IMU boards were manufactured as 2-layer boards with a standard thickness of 1.57 mm (0.062”). The board size was designed to be small yet practical for hand assembly. Cost constraints also drove the size and thickness of the board — custom thin boards are more expensive to manufacture. The board provides soldering holes for the 4-wire interface. These holes are on a 2 mm pitch, suitable for 2 mm connectors, if desired. The boards used on the GyroGlove do not use connectors, but instead have the wires directly soldered to them.



(a) IMU PC Board **(b) IMU PC Board Back**

Figure 4.4: GyroGlove IMU

The IMU wiring, seen in Figure 4.4b, stretches across the back of the board and is glued to the board using a cyan acrylic adhesive. This low-tech solution provides a strain relief for the wires to avoid breaking the solder connections. Small pieces of shrink wrap are used to bundle the wires together and secure them to the glove. The shrink wrap guides are attached to the glove using cyan acrylic. The wires from the controller to the IMU are 30 gauge multi-strand insulated wire. The multi-stranded wire is flexible to allow free movement of the glove and fingers. The wires are color coded, with the 3.3V power on the red wire, ground on the black, serial clock on blue and serial data on white.

The glove hardware uses the I^2C bus to connect the IMU boards to the controller board. The I^2C bus must always have a single master device, which is the controller board for the GyroGlove. The I^2C bus is a multi-drop bus, which means that multiple slave devices can be placed onto the same bus. Each slave device on the bus must have a different address however. It is common for two or more devices from the same manufacturer to be used on the same bus. Devices, such as the gyroscope or accelerometer included on the IMU board, will generally have an option to control one or more bits of the I^2C address with an external resistor. The IMU boards are designed such that two boards can share a common I^2C bus. Programming resistors on each board provide the option to set one of two unique addresses for each device on the board.

The gyroscopes and accelerometers on the IMU boards are housed in quad flat-pack no lead (QFN) surface mount packages. The accelerometer dimensions are 3 mm x 3 mm, while the gyros are 4 mm x 4 mm. The IMU board interfaces to the controller using four wires — two wires for power and ground, and two for the I^2C interface. Several capacitors on the board provide power supply bypassing with two additional capacitors required for proper operation of the gyroscope. Resistors are used to set the address zero bit for both the gyroscope and the accelerometer devices. Two pull-up resistors are placed on the I^2C bus.

The gyroscope device on the IMU board has a dual I^2C interface. The primary interface connects to the board input. The secondary I^2C bus is connected to the accelerometer. The gyroscope device has the ability to operate in pass-through mode or auxiliary interface mode. In auxiliary interface mode, the gyroscope controls the accelerometer device and the controller can read data for all six axes from the gyroscope.

4.2.1.1 InvenSense IMU-3000 Gyroscope

The IMU-3000 is a digital, 3-axis gyro with onboard 16-bit ADCs. The digital gyro has programmable full-scale ranges of ±250, ±500, ±1000, and ±2000 degrees per second (DPS). The gyro rate noise specification is 0.01 $\frac{dps}{\sqrt{Hz}}$. The gyro has a VDD operating range of 2.1V to 3.6V, and an interface voltage range of 1.7V to 3.6V. The IMU boards use 3.3V for the VDD and interface voltages.

The IMU-3000 requires a minimum number of external components. Bypass capacitors are required on regout, pin 10, and vlogic, pin 8. Bypass capacitors are also used on each of the VDD lines.

The data sheet for the IMU-3000 lists the internal registers for the device. There are a total of 63 registers in the device, each with an 8-bit address. The registers are initialized by firmware at system startup in order to configure the IMU-3000 for data collection.

4.2.1.2 ST Microelectronics LIS331 DLH Accelerometer

The LIS331 is a 3-axis accelerometer packaged in a 3 mm square QFN package. The LIS331 has a programmable full scale reading of $\pm 2g$, $\pm 4g$ or $\pm 8g$. The accelerometer outputs for the x-, y-, and z-axis have 16-bit resolution.

4.2.1.3 Instrument Frames

The instrument frame for the gyroscope does not match the body frame of the fingers or thumb. The mismatch between instrument and body frames requires a fixed rotation matrix between the instrument frame and body frame. A similar matrix is required for the accelerometers. The instrument frame for the hand is rotated 180 degrees relative to the fingers, so a different matrix is required for the hand instrument to body rotation matrix. The rotation matrices for the hand gyro and accelerometers are

$$R_i^B{}_{HG} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.1}$$

and

$$R_i^B{}_{HA} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{4.2}$$

While the rotation matrices for the finger and thumb are

$$R_i^B{}_{FG} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.3}$$

and

$$R_i^B{}_{FA} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{4.4}$$

4.2.2 Controller Board

Figure 4.5 shows the controller board. The controller board was manufactured in a 4-layer PCB process. The board dimensions are 45.7 mm (1.8") by 50.8 mm (2"). The top and bottom layers are signal routing layers, while the inner layers are power and ground. The controller board is designed such that the connections for the finger mounted sensors are forward, while the USB and serial interfaces point back toward the wrist.

The sectioning of the plane layers is shown in Figure 4.6. The internal power

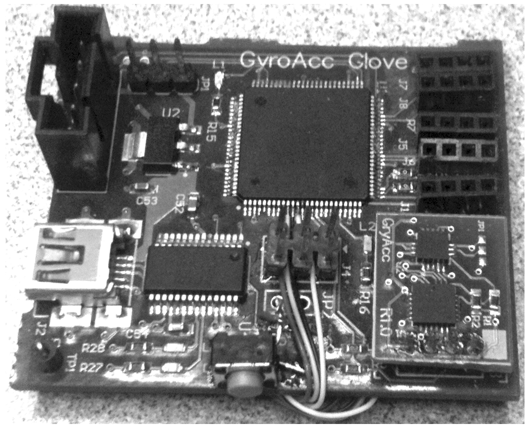


Figure 4.5: Controller Board

plane was split with a 5.0V and a 3.3V section. The smaller 5.0V section is for the incoming USB power, while the rest of the board uses 3.3V power. The split plane makes connection to the different power supplies much easier, when compared to individual routing of the power supplies on a two layer board.

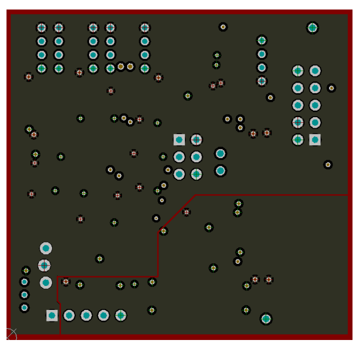


Figure 4.6: Controller Board Plane Layers Detail

Data sheets from the manufacturer of each device were the primary reference for component connection. The data sheets were consulted during the schematic design phase to ensure correct device connections. The resistor and capacitor values were selected according to the manufacturer's specifications.

The glove controller board block diagram is shown in Figure 4.7. The controller board includes the following components:

- Microprocessor,

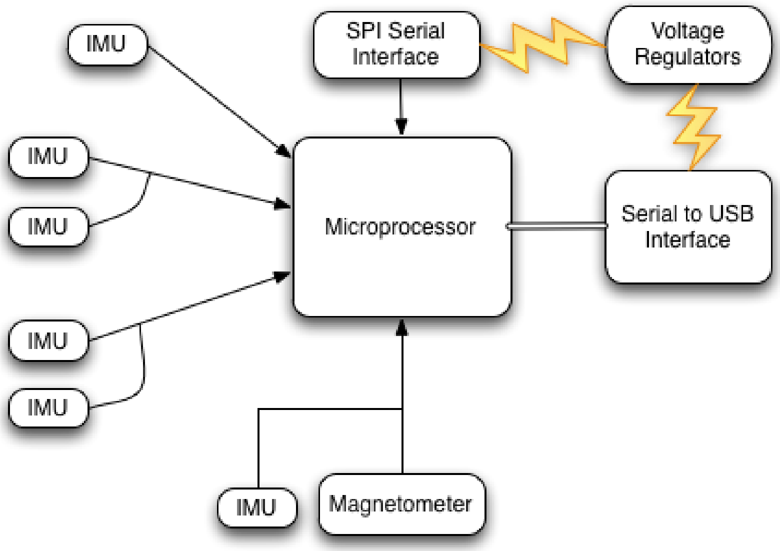


Figure 4.7: Block Diagram of GyroGlove Controller and IMU Boards

- USB to Serial interface,
- Voltage regulators,
- Magnetometer,
- Connectors for six IMUs,
- Programming header,
- Serial expansion port.

Additional details about the key components are provided in the following sections.

4.2.2.1 Microprocessor

The controller board uses an Atmel™ ATXMega128A1™ microprocessor. This device contains four hardware I^{2}C channels. The processor is packaged in a 100-pin thin quad flat pack (TQFP) package. This device was chosen for the internal resources, even though there are many unused pins. The ATXMega128A1™ provides a hardware engine for each of the four I^{2}C channels and the serial interfaces. The hardware engines allow the processor to control multiple I^2C and serial interfaces simultaneously, an important goal of the controller board.

4.2.2.2 USB Interface

The USB interface is provided by an FTDI™ FT232RL USB to serial device. FTDI™ provides software drivers that allow the USB interface to be treated as a virtual serial port. On a Windows™ PC, this means that the devices will show up as a COM port. The Windows™ device manager will display the COM ports in use once the USB is connected. On a Mac™ computer, the virtual COM port shows up as a device in the /dev device driver directory. The interface from the FTDI™ to the processor is a standard serial interface. The FTDI™ devices are able to operate at serial speeds up to three Mbps, however, the processor and FTDI™ devices must be closely aligned in speed, which is difficult in practice. The highest speed achieved in the GyroGlove was 400 kilobits per second (KBPS).

4.2.2.3 Voltage Regulators

Power for the board is provided from the USB interface. USB provides 5.0V at a maximum of 500ma of current. The actual current draw of this board is around 10ma, so the power consumption is not an issue. The voltage regulator used on the board is an LT1963 linear regulator with a fixed 3.3V output.

4.2.2.4 Magnetometer

The magnetometer is a Honeywell™ HMC5843 3-axis magnetic compass. This device is useful to determine the direction that the glove is facing, relative to magnetic north. Without the magnetometer, it is not possible to determine the direction of the glove about the z-axis.

4.2.2.5 Communication Ports

The GyroGlove controller board has support for four separate I^2C channels. There are six IMU devices on the system, so two of the I^2C channels must support dual IMUs. The IMU boards that share the same I^2C line must have the resistor address configurations set to opposite values to avoid conflicts. There are two possible address values for the gyros and two for the accelerometers. The gyro base address is binary b110100x, or 0xD0. The x is controlled by the external resistor on the IMU board and can be a 0 or a 1. The possible gyro addresses for the system are 0xD0 and 0xD2. The accelerometer addresses are 0x30 and 0x32.

The controller has a secondary serial port. This port can provide 3.3V power to the controller board as an alternative to the USB power. The port has a 3-wire serial peripheral interface (SPI). The SPI could be used by a higher level controller. The secondary interface was provided as an expansion option so that a wireless controller board with a battery supply could be used with the existing hardware.

4.3 Firmware Design

The GyroGlove firmware is written in C and C++. Atmel™ products use GNU compilers for development, so it is possible to develop on a Windows™, Mac™, or Linux™ platform. On Windows™, the Atmel™ AVR Studio software is available. On the Mac™, there is a free development kit available named CrossPack™. CrossPack™ was used for most of the development on this project. Command line GNU tools are used on the Linux™ platform.

The GNU compiler supports a limited subset of C++ for the AVR processors. The main advantage of using C++ is the ability to group the methods and data associated with a particular part of the system. Methods that manipulate the I^2C

interface are one example. An IMU_Manager class is used to capture all of the logic needed to work with the I^2C system within the firmware architecture.

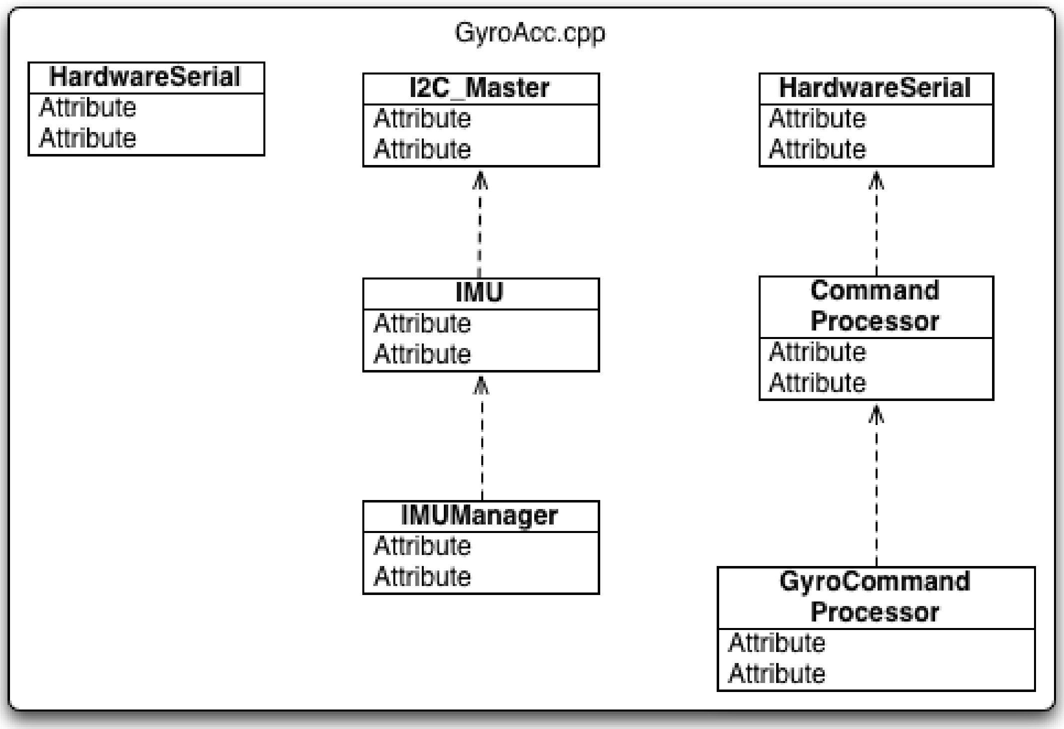


Figure 4.8: GyroGlove Firmware Architecture Block Diagram

A block diagram of the firmware architecture is shown in Figure 4.8. The C++ main function is located in GyroAcc.cpp. This function creates all of the C++ objects used in the firmware and then associates the objects together; refer to Code Listing 4.1. There are four I^2C ports in the processor, so four I2C_Master classes are created, with one connected to each port.

Code Listing 4.1: I^2C Class Creation

```
I2C_Master hand(&TWIC);
I2C_Master single(&TWID);
I2C_Master pair1(&TWIE);
I2C_Master pair2(&TWIF);
```

The classes created are given the variable names hand, single, pair1, and pair2. In Code Listing 4.2, four IMU classes are created and associated with their respective

I2C_Master classes. Once this association is made, the IMU class is able to access the assigned I^2C port. The IMU class is written in such as way that it does not matter which I^2C port is used.

Code Listing 4.2: Initialization of the IMUManager Class and the IMU Classes

```
IMU     hand_imu(&hand);
IMU     single_imu(&single);
IMU     pair1_imu(&pair1);
IMU     pair2_imu(&pair2);

IMUManager imumgr(&cmdSerial);
imumgr.LedOff();
imumgr.SetTimer(&tcA);
imumgr.AddIMU(&hand_imu);
imumgr.AddIMU(&single_imu);
imumgr.AddIMU(&pair1_imu);
imumgr.AddIMU(&pair2_imu);
```

The last part of Code Listing 4.2 shows the initialization of the IMUManager. Here, the IMU classes are associated with the IMUManager using the AddIMU method. This association completes the IMUManager hierarchy shown in Figure 4.8.

Code Listing 4.3: Initialization of the HardwareSerial and GyroCommandProcessor Classes

```
HardwareSerial dbgserial(&USARTF1, &PORTF, PIN6_bm, PIN7_bm);
dbgserial.begin(115200);
pdbgserial = &dbgserial;
pdbgserial->enable(false);

HardwareSerial cmdSerial(&USARTD0, &PORTD, PIN2_bm, PIN3_bm);
cmdSerial.begin(115200);

GyroCmdProcessor cmdproc(&cmdSerial,&pMaster[0],&imumgr);
```

The HardwareSerial and the GyroCommandProcessor classes are associated in a similar manner, as shown in Code Listing 4.3. Two HardwareSerial classes are created in the code. One is used as a debug port to allow functions within the code to print out messages. This is a useful debug technique, but also has problems. In hardware time, the serial communication is very slow, so calling a function to print out a debug message changes the timing of the firmware and can modify the very code that is being debugged.

The second HardwareSerial object is connected to the GyroCmdProcessor. The GyroCmdProcessor uses the CmdProcessor as a base class, but overrides the Loop method. Loop is the method where incoming commands are checked against the command table. The command table is actually just a long “If, else if” block that checks each command in turn.

Code Listing 4.4: GyroAcc Main Loop

```
while(1) {
    cmdproc.Loop();
    imumgr.Loop();
}
```

The last part of the main block of code, shown in Code Listing 4.4, is an endless while loop. This loop repeatedly calls the command processor loop() method and the IMUManager loop() methods. The command processor loop() method checks for any new commands. If a new command is available, the command loop processes the command and returns a result. The result is always “Ok” or “Fail:*message*”. The “Ok” response can optionally return some data values, such as “Ok:10,20,30”.

The IMUManager loop() method services the IMUManager state machine. The IMU objects operate primarily using hardware interrupts, but when a new packet is ready, the IMUManager loop() method is where the packet is assembled and transmitted down the serial link to the host processor.

4.3.1 Gyro Command Processor

The GyroCmdProcessor has associations with the IMUManager and HardwareSerial classes. The processor checks for new commands from the HardwareSerial and then traverses an “if, else” block. If the command is found, the command parameters are used to call the appropriate functions. The processing loop is quite long, but an excerpt from the loop is shown in Code Listing 4.5.

Code Listing 4.5: GyroCmdProcessor Loop Example

```
} else if(strcmp(pCmd,"streamstart") == 0) {
    uint16_t bUseGyro = 0;
    if (paramCnt() > 0) {
        getParam(0,bUseGyro);
    }
    int retc = _pMgr->StreamStart(bUseGyro == 1);
    if (retc < 0) {
        sprintf(buffer,"Fail:%d\n",retc);
        _pHW->print(buffer);
    } else {
        _pHW->print("Ok\n");
    }
} else if(strcmp(pCmd,"streamstop") == 0) {
    _pMgr->Stop();
    _pHW->print("Ok\n");
```

The “StreamStart” command takes one optional parameter. If the paramCnt is greater than 0, the parameter is extracted into bUseGyro and used during the command. The “StreamStop” command takes no parameters, but calls the Stop() method of the IMUManager class.

4.3.2 I^{2}C Transaction Primer

The ATXMega128A1™ processor includes a powerful hardware I^{2}C engine. The engine allows the hardware to perform all of the low-level (tedious) aspects of the I^{2}C transaction, thus freeing up the software to manage other tasks. A little background information about I^{2}C transactions is required before going into more detail about the I2C_Master class.

All I^{2}C transactions begin with a **START** marker and conclude with a **STOP** marker. These markers are shown in Figure 4.9.

A key item to note about I^{2}C transactions is that the SDA line never transitions while SCL is high, *unless* the master is generating a **START** or **STOP**. Therefore, a

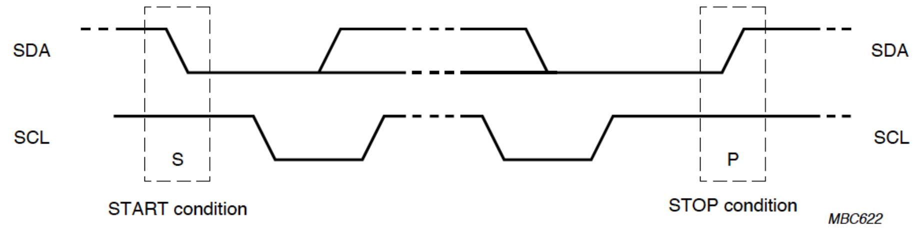


Figure 4.9: I^{2}C Start and Stop Transactions

START is always defined as a falling edge of SDA while SCL is high, and a **STOP** is defined as a rising edge of SDA while SCL is high.

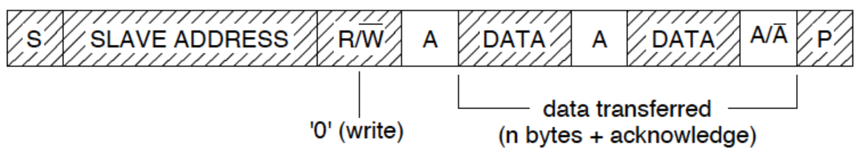


Figure 4.10: I^{2}C Write Transaction

Data transactions for I^{2}C are always 8-bit transactions, with an acknowledge cycle. The acknowledge is the ninth bit of every transaction. All transactions are either write transactions or read transactions. In a write transaction, the master controls the SDA line during the first 8-bit section, while the slave controls the data line for a read. At the end of a write transaction, the master releases the SDA line for the ninth bit. The slave must hold the SDA line LOW in order to acknowledge receipt of the 8-bit value. If the slave does not hold the SDA line low, then the master registers a Not Acknowledged (NACK) condition.

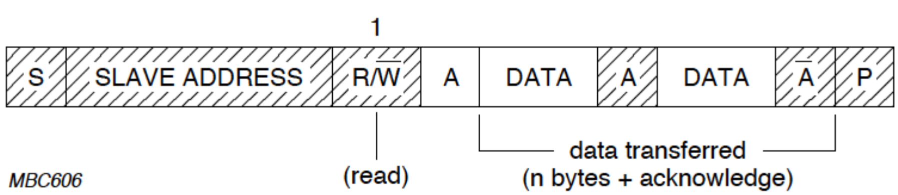


Figure 4.11: I^{2}C Read Transaction

For a read, the slave controls the SDA line for the first 8-bits, then releases the line. The master must then hold the SDA line low to acknowledge (ACK). For reads, the ACK generally signals to the slave that it is okay to send the next byte in the sequence. Some slave devices use an auto-increment feature that allows the master to request data from the slave starting at a particular address and continuing until the master NACKs.

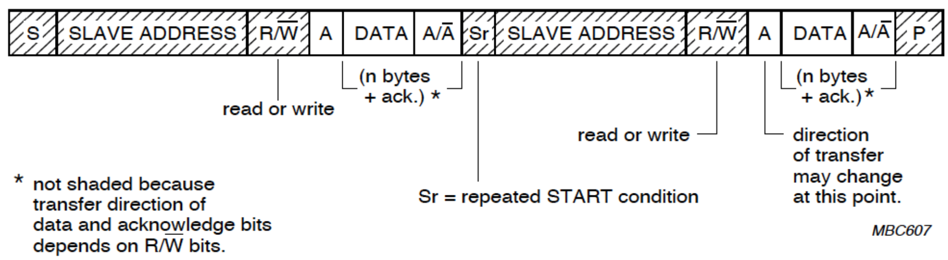


Figure 4.12: I^{2}C Combined Transaction

The first 9-bit transaction after every **START** is always the **ADDRESS** transaction. In this transaction, the first seven bits of the 8-bit data contain the **ADDRESS** of the intended slave device. The last bit is a Read-Not-Write bit. A '1' indicates this is a read request, while a '0' is used for a write.

There are three basic types of I^{2}C sequences. A write sequence, a read sequence, and a combined sequence. These sequences are illustrated in Figure 4.10, Figure 4.11, and Figure 4.12, respectively. A write transaction sends a **START**, **ADDRESS**, one or more data bytes, and a **STOP**. A read transaction sends a **START**, **address**, then receives one or more read bytes, then issues a **STOP**. The combined transaction begins with a **START**, **ADDRESS**, and a **WRITE**. The first **WRITE** is generally the slave device register address. Next, the master issues a **REPEATED START**, which is defined as a second **START** without a **STOP** signal. After the **REPEATED START**, the slave **ADDRESS** is sent again, this time with the read bit set.

4.3.3 I2C_Master

The I2C_Master class manages the complexities of the I^{2}C transactions. The class uses the ATXMega128A1™ hardware interrupts to minimize the software load on the processor. Because the master is completely interrupt driven, all I^{2}C communications must be treated as asynchronous operations.

The I2C_Master class uses a state machine to manage the transactions and to determine the next action upon receipt of an interrupt. Clients of the I2C_Master class call the Write and Read functions in order to initiate I^{2}C transactions. The I2C_Master calls back to the client code when a significant event occurs, such as when a transaction completes or a failure is detected. The notification mechanism allows the client code to initiate a transaction and then wait for the transaction to complete. In practice, the client logic also uses a state machine so that it can respond properly to the I^{2}C notifications.

This document will not delve further into the internals of the I2C_Master class. It should be sufficient to understand that transactions are initiated by the client, the master manages the transaction until it either completes or an error occurs, then the master notifies the client of the result. The client is then free to respond appropriately, which may include reading the data results from the I2C_Master.

4.3.4 IMU and IMUManager

The IMU and IMUManager classes are quite complex. A complete description of their functions would take a significant amount of space. The complete source code, with comments, is included in Appendix E should more details be required. This section will provide just a high level overview of the operation of these classes.

The IMU class is responsible for performing the initialization of the connected gyro and accelerometer devices. When the IMU class is created, it performs a query of the associated I^{2}C interface. The query checks all of the possible I^{2}C addresses

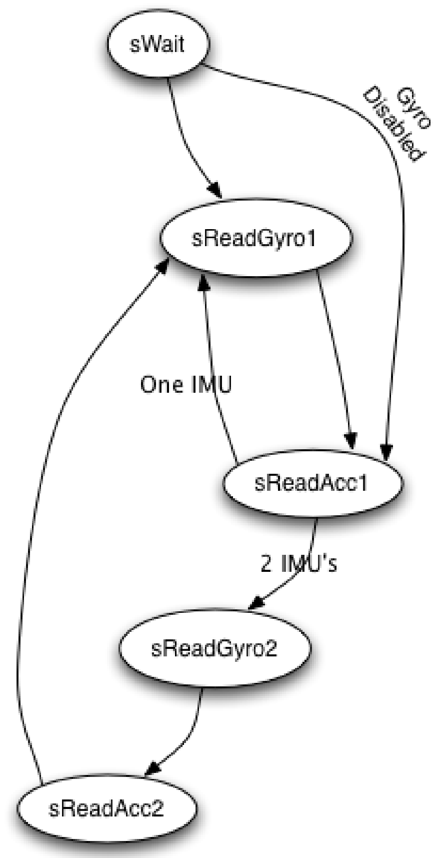


Figure 4.13: Partial State Machine Diagram for the IMU Class

to determine what devices are connected. If two IMUs are connected, then the IMU class will be configured in dual IMU mode.

The IMUManager will call the Start() method to begin data streaming. The sequence of operations is partially shown in the state machine diagram of Figure 4.13.

The IMU class state is updated by one of two possible events. A hardware timer is configured to call the Run() method periodically. The Run() method is the method that initiates the state machine data sequence, but it also contains logic to check for a read timeout. In the event of a timeout, the IMU is reset and started again. After a Start(), the class variables are reset and the state is set to sWait. The first time that Run() occurs while state = sWait, the IMU state is transitioned to one of the read states. The state chosen is based on IMU class boolean flags and is shown in Table 4.1.

Table 4.1: IMU Class Run Method Initial State Table

Gyros Enabled	State
True	sReadGyro1
False	sReadAcc1

Each time a state is changed to one of the read states, StartTransaction() is called. This method initiates an asynchronous read on one of the I^2C devices, with the choice based on the current state. The asynchronous transaction will call back to the IMU class if the transaction fails, or completes successfully. The IMU class contains logic to handle all of the possible outcomes of the read. If the read is successful, then I2CReadDone() is called. This method resets the busy timeout and fail counters (since it just passed), and then calls ProcessTransaction().

Code Listing 4.6: IMU Class ProcessTransaction Method

```
void IMU::ProcessTransaction()
{
    switch(_State) {
        case sReadGyro1:
            StoreGyroData(1);
            SetState(sReadAcc1);
            break;
        case sReadAcc1:
            StoreAccData(1);
            PushData(1);
            if (_bDualChan) {
                if (_bUseGyro) {
                    SetState(sReadGyro2);
                } else {
                    SetState(sReadAcc2);
                }
            } else {
                SetState(sWait);
                if (_pNextIMU) {
                    _pNextIMU->BeginRead();
                }
            }
            break;
        case sReadGyro2:
            StoreGyroData(2);
            SetState(sReadAcc2);
            break;
        case sReadAcc2:
            StoreAccData(2);
```

```
            PushData(2);
            SetState(sWait);
            if (_pNextIMU) {
                _pNextIMU->BeginRead();
            }
            break;
        default:
            break;
    }

    //! Start the next transaction.
    StartTransaction();
}
```

ProcessTransaction(), shown in Code Listing 4.6, extracts data from the I^2C object and stores it in the class. It then changes the state based on the state transition diagram shown in Figure 4.13 and ends with another call to StartTransaction(). This completes the loop that continues as long as IMU data is streaming.

When a read completes in the IMU class, the new data is stored with the StoreGyroData(n) or StoreAccData(n). The methods simply transfer the data from the I^2C class to the IMU class. The PushData(n) function is called next. PushData(n) sets a boolean flag. This flag indicates that all of the data for a transaction has been read. The IMU Master calls the DataReady() method to determine if the IMU has a complete packet of data available.

The IMU_Manager stores a list of IMU class pointers. Much of the work that the Manager does is to iterate over the IMUs to perform some operation. The most important function in the Manager is the Loop() function, which is shown in Code Listing 4.7. The Loop() function is called repeatedly, as was shown in Code Listing 4.4.

Code Listing 4.7: IMU_Manager Loop Method

```
int IMUManager::Loop()
{
    switch(_State) {
    case sIdle:
        break;
    case sDataWait:
```

```
        if (DataReady()) {
            ResetDataReadyTO();
            _State = sDataReady;
        } else if (DataReadyTimeout()) {
            ResetDataReadyTO();
            _State = sDataTimeout;
        }
        break;
    case sDataReady:
        PacketLedIndicator();
        if (_nStreamWDCounter == 0) {
            DiscardData();
            _State = sDataWait;
        } else {
            --_nStreamWDCounter;
            SendPacket(false);
            _State = sDataWait;
        }
        break;
    case sDataTimeout:
        PacketLedIndicator();
        if (_nStreamWDCounter == 0) {
            DiscardData();
            _State = sDataWait;
        } else {
            --_nStreamWDCounter;
            SendPacket(true);
            _State = sDataWait;
        }
        break;
    }

    return 0;
}
```

The Loop() function waits in the sDataWait state until all of the IMU objects return true, or a timeout occurs. If a timeout occurs, then a dummy packet is sent to avoid data starvation of the client. In normal cases, the sDataReady state is set, and the next call to Loop() will assemble the packet data and send a packet of measured data to the client.

Code Listing 4.8: IMU_Manger SendPacket Method

```
void IMUManager::SendPacket(bool bTimeout)
{
    uint8_t*    pPacket = &_dataPacket[0];
    if (true || !bTimeout) {
        for (int x = 0;x<4;x++) {
```

```
6              if (_pIMU[x]) {
7                  // This puts the data at the pointer,
8                  // then returns the end of the data.
9                  // This might be 2*14 or 1*14
10                 pPacket = _pIMU[x]->GetPacketData(pPacket);
11             }
12         }
13     }
14     // Packet format:
15     // SNP header
16     // byte: length of packet
17     // byte: packet type (0xB7)
18     // byte(s): length bytes
19     // bytes(2): 2 byte CRC
20     // string: END
21     // newline
22     uint8_t size = pPacket - &_dataPacket[0];
23     buffer[0] = 'S';
24     buffer[1] = 'N';
25     buffer[2] = 'P';
26     buffer[3] = 0xB7;
27     buffer[4] = _packetId++;
28     buffer[5] = size;
29     memcpy(&buffer[6],&_dataPacket[0],size);
30     // Compute CRC -- someday
31     uint16_t crc = 0xaf5a;
32     uint8_t crchi = (crc >> 8) & 0xff;
33     uint8_t crclo = crc & 0xff;
34     buffer[6+size]   = _nStreamWDCounter;
35     buffer[6+size+1]   = crchi;
36     buffer[6+size+2] = crclo;
37     sprintf((char*)&buffer[6+size+3],"END\n");
38     _pSerial->write(&buffer[0],6+size+3+4);
39 }
```

The last firmware method described here is the SendPacket() method, shown in Code Listing 4.8. In line 10, the hry() method of each IMU class is called. The IMU classes assemble the read data into a well-defined format. The total length of this data is calculated in line 22. The packet is assembled starting in line 23 where the packet header of “SNP” and the packet type of 0xB7 are added. Finally, the packet size, packet data, and a dummy CRC code are added, and the packet is terminated with the “END” string. The assembled packet is transmitted to the serial port using the write command.

The final write to the serial port is a synchronous call. When viewed on the logic analyzer, the serial write overlaps with the I^2C reads, so the minimum system period is not the total of the serial and I^2C transactions, but the longest of the two. The total time to read all IMUs is about 4 ms. This sets the maximum read rate of the system to about 200 Hz, which is a 5 ms period.

4.3.5 Packet Data Rate Calculations

Each IMU reads seven, 16-bit values for a total of 14 bytes. Two additional bytes are sent as packet headers. The size of a full packet of data is 12 bytes + 6 * 16 for a total of 108 bytes. A serial byte includes a start and stop bit, for a total of 10 bits per byte of data. This results in a total of 1080 bits per full packet. At a serial rate of 115,200 Kbps, the total time to send a packet is 1080/115,200 or 9.3 ms. A baud rate of 115,200 is barely fast enough to operate at 100 Hz. Fortunately, the serial interface is able to go much faster, but higher rates do make the communication setup more difficult. The baud rates between the host computer and the hardware must be closely aligned in frequency. Baud rates as high as 400 Kbps were achieved during this project, but the more reliable 115,200 was used for much of the project, along with an IMU update rate of 100 Hz or lower.

4.4 Software Design

The host computer software is designed using the Python™ scripting language, MATLAB™ , and Panda3D™. Python™ captures data from the hardware. MATLAB™ reads data from the Python™ server and performs all IMU, kinematic and position calculations. Panda3D™ is used by MATLAB™ to display the calculated results in a real-time 3D “view” of the hand and fingers.

4.4.1 Python™

The GloveServer application has a simple Qt™ GUI, shown in Figure 4.14. The GUI displays the number of IMUs identified by the system, which should normally be six. The *Rate* setting is used to configure the IMU capture rate. *Packets Captured* updates as the data is streamed and shows that the streaming interface is working.

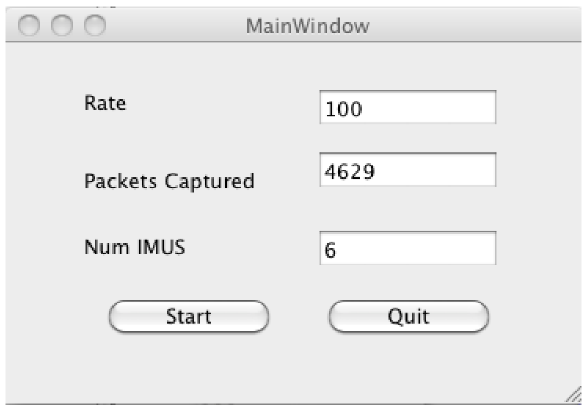


Figure 4.14: GloveServer GUI

The code for the GloveServer application is contained in several Python™ files. The first file contains the GUI code and two python thread objects. The DataWorker thread object calls the GloveAPI getIMUPacket() method. This method is a synchronous call and does not return until a packet is read. Since the DataWorker is in a separate thread, the GUI continues to respond while it is waiting. The source code files for the GloveServer are included in Appendix A.

The SocketWorker thread creates a network socket on the localhost IP address `127.0.0.1`, port 5120. The socket connection is accessible by any computer on the local network. SocketWorker listens for connections on this port. After a connection is established, the SocketWorker continuously calls the recv(1024) method to get commands from the client. A command processor determines the command, any options passed, and then executes the requested function. The command most used is the "data" command, which requests a single packet of data from the server.

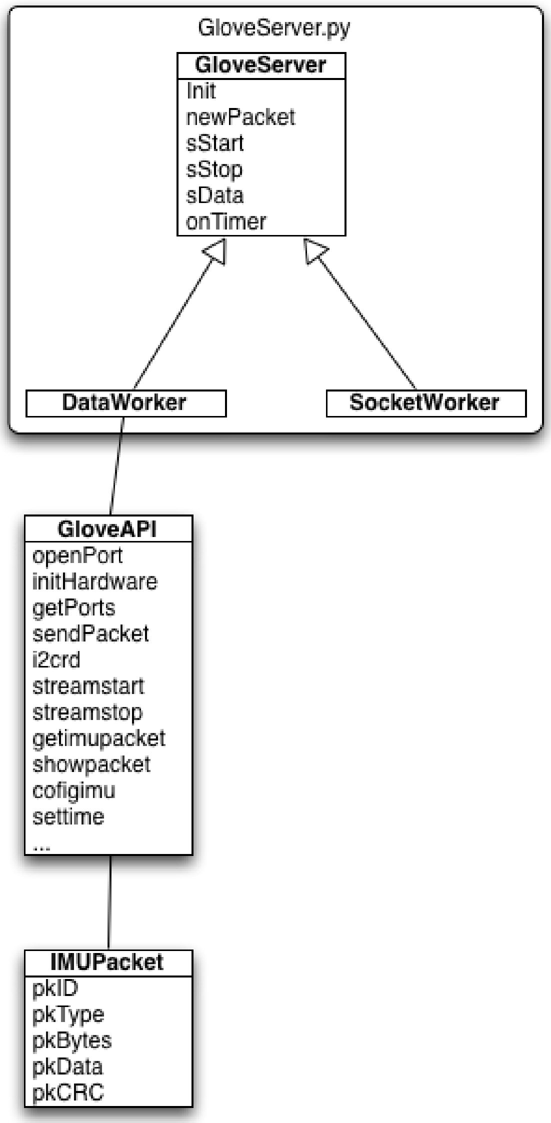


Figure 4.15: UML Diagram of the Python™ GloveServer

GloveAPI is a reusable Python™ class. GloveAPI is used within the main GUI, and also works as a standalone application for streaming IMU data to a file. The class implements functions useful for working with the GyroGlove over the serial port. GloveAPI was used repeatedly during development as a debug tool. GloveAPI contains too many methods to document here. The key capabilities provided by GloveAPI are as follows:

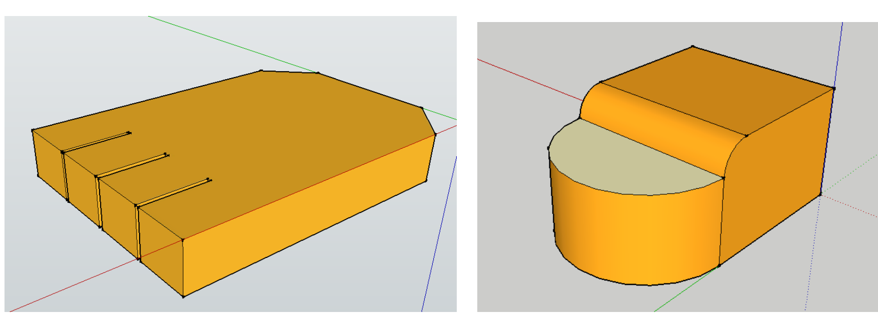
- Open and close the serial port, with options to set the baud rate.

- Start and stop the IMU data stream.
- Retrieve a new IMU Packet.
- Send debug commands to perform I^2C reads, writes and initializations.
- Configure the IMUs.

The IMUPacket.py file contains two classes used to capture and manipulate the IMU packet data. This class encapsulates the packet data formats and includes methods to extract particular parts of the packet. The IMUPacket class also includes the numpy code to perform the instrument frame to body frame rotations. All data returned from the IMUPacket class is therefore in body frame coordinates.

4.4.2 Panda3D™

The Panda3D™ server is contained in a single Python™ file Glove3D.py. This file loads physical models, which are 3D graphic objects. The graphic objects for this project were drawn in Google Sketchup™. The sketchup files were saved into .dae COLLADO format, which requires Google Sketchup Pro™. The .dae files were converted to panda eggs using the dae2egg command in the Panda3D™ distribution. Details about this process are available in the Panda3D™ documentation.



(a) Hand Base **(b) Fingertip**

Figure 4.16: Google Sketchup™ Drawings Used in the Panda3D™ 3D Visualizer

Panda3D™ provides mechanisms to load models into the 3D world and position them. The position of a model is set using 6DOF coordinates using the setPosHpr() command. This command takes the X,Y, and Z coordinate, as well as a Roll, Pitch, and Yaw angle. A very nice feature of Panda3D™ is that the position of an object can be set relative to its parent. When the parent moves, all child objects move with it. The position and orientation of the child object is always in relation to the parent. The fingers in the model are configured to be child objects of the hand. This means that the center of the coordinate system for the fingers is the hand, while the center of the coordinate system for the hand is the world. The child objects can also be given offset values, such that their zero position places them at a more natural location in the model.

The relative positioning of the objects in Panda3D™ lends itself particularly well to the IMU calculations. It is convenient to perform translations from the body frame of the fingers to the body frame of the hand. The Panda3D™ models defined for this project make it easy to represent these relationships visually.

The Glove3D.py file creates a network socket, similar to the GloveServer. The IP address for the socket is again`127.0.0.1`, but the port is 5432. The data interface to the Panda3D™ server is quite simple. Data packets consist of a set of seven comma separated values. The first value is an integer that determines the target of the data. An index value of 0 sets the hand coordinates, index values of 1-4 set the fingers, and an index value of 5 sets the thumb. The remaining six values set the X, Y, and Z position and the Roll, Pitch, and Yaw of the selected element.

4.4.3 MATLAB™

The MATLAB™ code for this project consists of a set of six classes. The diagram of the classes, in unified modeling language (UML) format, is shown in Figure 4.17. The base classes are at the top, while the derived classes are below. The GuiBase class provides a number of helper functions for building a MATLAB™ GUI. The functions available perform basic operations, such as creating a ComboBox and adding it to the GUI at a specified location. All of the GUI elements created are stored in a uidata structure.

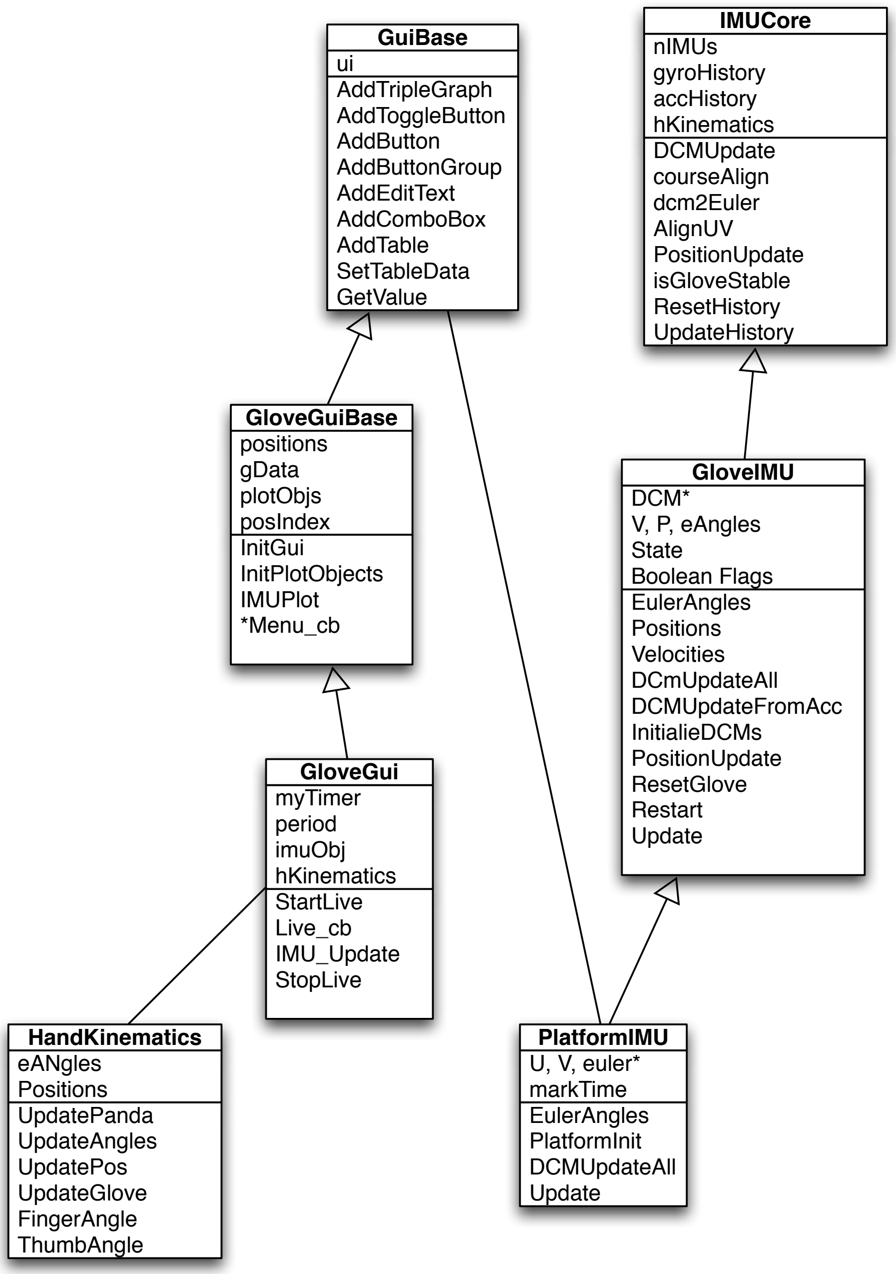


Figure 4.17: GloveGui UML Diagram

GloveGuiBase creates the GUI in the InitGui method. This method calls the

GuiBase methods to create all of the elements needed in the GloveGui. GloveGuiBase also contains the callback functions for all of the GUI elements. When a button is pressed or a combo box value is changed, the callback performs the requested action. The GloveGui is shown in Figure 4.18.

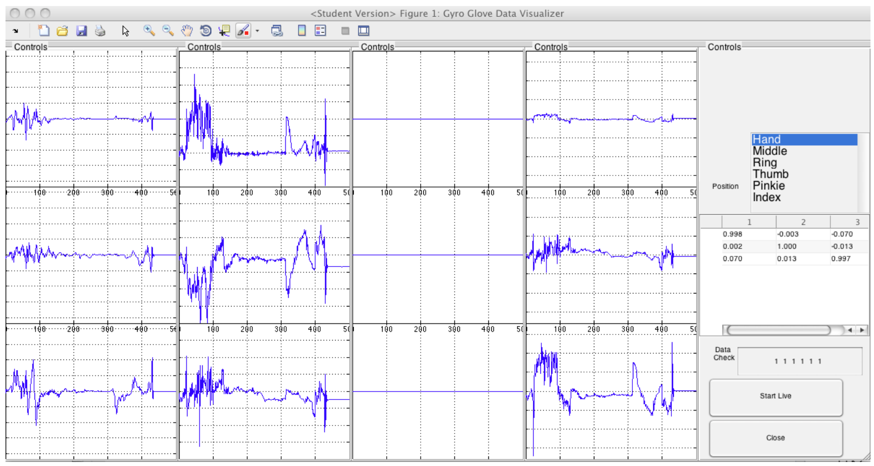


Figure 4.18: MATLAB™ Glove GUI

GloveGui implements the streaming capabilities for the MATLAB™ code. The class starts a timer that periodically retrieves new data, performs processing, and updates the Panda3D™ visualization. None of the IMU processing is performed in the GUI classes. The GloveGUI constructor function takes an optional argument that is used to specify an IMU object. If no object is provided, the GloveIMU class is used as a default.

The IMUCore class contains functions that implement the basic IMU equations developed during this project. This class implements the algorithms described in Chapter 2. The class provides functions to perform the DCM update, courseAlign, convert a DCM to Euler angles, and determine if the glove has been stable for a specified period of time.

The GloveIMU class uses the functions provided by IMUCore. This class tracks the IMU data for all six of the IMUs in the system, so it usually calls the IMUCore class in a loop. GloveIMU also contains a state machine. The state machine begins in the Idle state, and transitions to the Run state after performing all of the required initialization steps. The state machine ensures that the glove is stable for a period of 1.5 seconds. Then, the machine uses the course align method to set the initial orientation of all six IMUs. Finally, the class enters the Run state where the IMU data is updated at the IMU update rate.

A PlatformIMU class overrides some of the GloveIMU functions to use a different algorithm for updating the glove data. The Glove IMU algorithms track each of the six IMUs relative to the local inertial frame of reference. In order to determine the finger position relative to the hand, the finger body frame is rotated to the inertial frame. Then, the hand frame is rotated to the inertial frame, and the two frames are then compared. A different approach is to calculate and maintain the orientation of the finger frame relative to the hand frame directly.

The HandKinematics class provides some (very rudimentary) functions for calculating kinematic relationships. The calculated values are used to update the Panda3D™ model and to maintain reasonable relationships in the model. For example, it does not make sense for the fingers to be one meter away from the hand. The HandKinematics class takes the calculated position data from the IMU classes, but limits the position to reasonable values.

4.4.4 MATLAB™ Mex Functions

MATLAB™ support for socket programming requires the distributed computing toolbox. This toolkit was not available on the development system, therefore the socket programming was done in a pair of C++ mex files.

The disadvantage of the mex files is that they are not necessarily cross platform. To use the GyroGlove on a Windows™ PC, the mex files will need to be ported to that platform. Most of the code in the files is portable between platforms. Porting the code to a different operating system should not be difficult, but the mex files will need to be recompiled on the new platform.

Two C++ mex files were written for this project. The first one, named GyroGloveCapture.cpp creates a persistent socket object and connects to the GloverServer. MATLAB™ calls into this mex file to retrieve new packets of data. The second mex file is called GyroGloveClient.cpp. MATLAB™ calls this file to send data to the Panda3D™ server, thus updating the Panda3D™ graphic view.

Static variables within a mex file retain their value after the function is called. In this way, a mex function can maintain state between calls. The GyroGloveCapture mex function uses static variables to hold the socket connection. This allows the mex function to maintain a persistent socket connection, and improves performance. A simple set of commands are needed to control the internal state of the socket. The commands supported are *connect*, *close*, *start*, *stop*, *quit*, and *recv*.

The *connect* and *close* commands are used to connect to the GloveServer socket and close the socket, respectively. The *start* and *stop* commands are used to control the streaming of data from the GloveServer. The *recv* command takes one parameter that specifies the number of packets to receive. The function will retrieve the specified number of packets into a MATLAB™ array.

The GyroGloveClient mex function is used to send position and rotation values to the Panda3D™ server. The first argument to the mex function is the index of the model element to update, as described in Section 4.4.2. The second argument to the command is an array of six values. The values are the x, y, z, ρ, θ, ψ values for the specified element.

CHAPTER 5

RESULTS

The GyroGlove system developed for this thesis project is a successful hardware and software platform that can be used for further development and exploration of IMU systems, inertial measurement algorithms, gesture capture, and gesture pattern recognition. The project resulted in development of the following hardware and software artifacts:

- A set of IMU boards and a microprocessor-based controller.
- A USB interface from the controller to the host PC.
- Microprocessor firmware for IMU board initialization and data processing.
- A Python™ software component to connect to the controller and capture streaming data.
- A MATLAB™ software system to collect data from the hardware, perform calculations, and update a real-time 3D visualization of the hand.
- A flexible and extensible MATLAB™ software system for GUI development and IMU calculations.
- A Panda3D™ software component to display a 3D real-time view of the hand with updates from MATLAB™.

5.1 Hardware Results

The IMU boards developed for the project are small, simple and easy to assemble (with the right tools). The boards provide 6-axis IMU sensing with a 4-wire I^2C interface to the host controller. The boards are small enough to mount on to the fingertips and fast enough to be used for data capture of natural biokinematic movements. The IMU boards use state-of-the art, low cost, digital output accelerometers and gyroscopes.

The controller board uses a fast, efficient, low power processor. This processor is programmable using free development tools in C/C++. The processor includes sufficient horsepower to capture data from all six IMU boards simultaneously. The controller can transfer the raw data to the host PC at full speed using a simple, platform agnostic USB to serial protocol. The controller includes a complete, working firmware solution that captures data from all IMU devices at 100 Hz.

5.2 Software Results

The software components developed for this project form a flexible and powerful architecture. The overall system is constructed of three main components that work independently. Independent components have the advantage that they each perform a single, cohesive task, resulting in a simpler overall solution. The independent blocks can also take advantage of modern multi-core computers more effectively.

The Python™ data capture component provides a simple yet powerful hardware interface. Since the data capture server uses network sockets, the server is able to provide data to an client program with socket capabilities. While the target for the project was a MATLAB™ component, further development could easily substitute other languages or platforms. The flexible nature of this interface will be a great benefit to future work.

Applications written in MATLAB™ can be very complex. Standard function based .m files are poorly suited for large software applications. While MATLAB™ has supported object-oriented programming for a number of years, a majority of MATLAB™ applications are still written in the more traditional functional format. A recent review of the applications submitted to the MATLAB™ file exchanged showed that there were 818 functions, 380 scripts, and 57 classes. The initial learning curve of MATLAB™ classes may be steep, but it always helps to have working code to build upon.

The MATLAB™ framework developed herein provides an object-oriented suite of classes for the GUI and the IMU calculations. Low level GUI tasks are provided, as well as more specific objects and functions tailored for this application. The IMU objects form a 3-level structure with each level building upon the lower level. The tiered structure simplifies development of new algorithms, since they can be easily added on top of the main IMU objects without affecting existing code.

Verifying software algorithms can be a difficult task. Often the easiest way to see if a task works is to have a visual representation of the output results. The Panda3D™ server provides real-time visualization of the calculated MATLAB™ results. The visual output provides immediate and clear feedback during algorithm improvements. The Panda3D™ environment is also simple enough to expand and is available on a range of platforms.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The small size of MEMS devices makes them suitable for use in applications like the GyroGlove. Unfortunately, their noise levels are high, and so they cannot be used effectively for position tracking. Some possible solutions to this problem were considered early in the project, but never explored.

The kinematic relationships between the hands and fingers imply limits to how much a finger can move relative to the hand. This relationship could be used as additional information to the INS equations. As an example, point your index finger out in front of you. If the INS drift seems to indicate that the finger is moving forward, but the hand (and other sensors) is not, then this information should be disregarded. It is easy to state these relationships, but much more challenging to code them into suitable algorithms. A potential future project would be to merge the kinematic and INS equations.

Position tracking of an IMU was shown to be difficult or impossible. The GyroGlove, however, includes six IMUs. The hand and fingers are also tightly coupled, as mentioned previously, so there should be significant redundancy between the sensors. If the hand moves, so do the fingers, so if one finger drifts off in a different direction than the other, the algorithm should take this inconsistency into account. None of the books reviewed during this project discuss the use of redundant sensors. A second future project could attempt to capitalize on the redundant information to compensate for the noise issues prevalent in small MEMS sensors.

A final future project idea would be to focus on relative, short-time duration gestures. As previously stated, it is difficult to track the absolute position of the IMUs. It may be possible, however, to ignore the absolute position and focus only on the relative position. Hand gestures intended to control computer operations are short-term events. The IMUs are good at tracking over short time periods. It may be possible to focus only on short time periods and relative positions of the hand and fingers to recognize gestures.

The results of this project form a solid foundation for future work. The hardware and software platform developed here will support a wide variety of projects covering algorithms, pattern recognition, firmware development, or even hardware development. Hopefully, this thesis can serve as a great starting point for such work.

REFERENCES

[1] acceleglove. http://www.acceleglove.com/, Oct 2011.

[2] Businessweek.com images. http://images.businessweek.com/ss/06/05/phaidon/source/10.htm, Oct 2011.

[3] Computer history museum. http://www.computerhistory.org/timeline/?year=1951, Oct 2011.

[4] Doug engelbart institute. http://www.dougengelbart.org/firsts/mouse.html, Oct 2011.

[5] Timeline of computer history. http://www.computerhistory.org/timeline/?category=cmptr, Feb 2011.

[6] Wikipedia - kensington expertmouse. http://en.wikipedia.org/wiki/File:Trackball-Kensington-ExpertMouse5.jpg, Oct 2011.

[7] Wikipedia - space navigator. http://en.wikipedia.org/wiki/File:Space-Navigator.jpg, Oct 2011.

[8] Xsens mtx sensor. http://www.xsens.com/en/general/mtx, Oct 2011.

[9] Xsens mvn biomech. http://www.xsens.com/en/general/mvn-biomch, Oct 2011.

[10] P Asare. A sign of the times: A composite input device for human-computer interactions. *Potentials, IEEE*, 29(2):pages 9 – 14, MAR/APR 2010.

[11] TD Bui and L.T Nguyen. Recognizing postures in Vietnamese sign language with MEMS accelerometers. *Sensors Journal, IEEE*, 7(5):pages 707–712, May 2007.

[12] Community. Wikipedia - douglas engelbart. `http://en.wikipedia.org/wiki/Douglas_Engelbart`, Oct 2011.

[13] Community. Wikipedia - eniac. `http://en.wikipedia.org/wiki/File:ENIAC_Penn1.jpg`, Oct 2011.

[14] CyberGlove. Cyberglove ii. `http://www.cyberglovesystems.com/products/cyberglove-ii/overview`, Feb 2011.

[15] Charles Stark Draper. Origins of inertial navigation. *Journal of Guidance and Control*, 4(5):449–463, Sep 1981.

[16] W English, D Engelbart, and M Berman. Display-selection techniques for text manipulation. *Human Factors in Electronics, IEEE Transactions on*, HFE-8(1):pages 5 – 15, Mar 1967.

[17] Jay A. Farrell. *Aided Navigation: GPS with High Rate Sensors*. McGraw-Hill, 2008.

[18] Herman Goldstine. A brief history of the computer. *Proceedings of the American Philosophical Society*, 121(5):pages 339 – 345, Oct 1977.

[19] H Graf, Sang Min Yoon, and C Malerczyk. Real-time 3d reconstruction and pose estimation for human motion analysis. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3981 – 3984, Hong Kong, Sep 2010.

[20] J Hanse. Honeywell mems inertial technology & product status. *Position Location and Navigation Symposium, 2004. PLANS 2004*, pages 43 – 48, 2004.

[21] K Hays, R Schmidt, W Wilson, J Campbell, D Heckman, and M Gokhale. A submarine navigator for the 21st century. In *Position Location and Navigation Symposium, 2002 IEEE*, pages 179 – 188, Palms Springs, CA, Apr 2002.

[22] Jose L. Hernandez-Rebollar, Nicholas Kyriakopoulos, and Robert W. Lindeman. The acceleglove: a whole-hand input device for virtual reality. 2002.

[23] Ji-Hwan Kim, Nguyen Duc Thang, and Tae-Seong Kim. 3-d hand motion tracking and gesture recognition using a data glove. pages 1013 – 1018, 2009.

[24] J Perng, B Fisher, S Hollar, and K Pister. Acceleration sensing glove (asg). In *Wearable Computers, 1999. The Third International Symposium on*, pages 178 – 180, San Francisco, CA, Oct 1999.

[25] PowerGlove. Nintento power glove. `http://www.ageinc.com/tech/index.html`, Feb 2011.

[26] J Raskin. Holes in history: a personal perspective on how and why the early history of today's major interface paradigm has been so often misreported. *interactions*, 1(3):pages 11 – 16, Jul 1994.

[27] J Rekimoto. Gesturewrist and gesturepad: unobtrusive wearable interaction devices. In *Wearable Computers, 2001. Proceedings. Fifth International Symposium on*, pages 21 – 27, Zurich , Switzerland, 2001.

[28] M Sadiku and C Obiozor. Evolution of computer systems. In *Frontiers in Education Conference, 1996. FIE '96. 26th Annual Conference., Proceedings of*, volume 3, pages 1472 – 1474, Salt Lake City, Utah, Nov 1996.

[29] Paul G. Savage. *Strapdown Analytics, Part 1.* Strapdown Associates, Inc., 2007.

[30] Guoqing Xu, Yangsheng Wang, and Xiufeng Zhang. Human computer interaction for the disabled with upper limbs amputation. In *Advanced Computer Control*

(ICACC), 2010 2nd International Conference on, volume 3, pages 120–123, Shenyang, China, Mar 2010.

APPENDIX A

Python™ GLOVESERVER SOURCE CODE

Python™ GLOVESERVER SOURCE CODE

Code Listing A.1: Top Level

```python
#!/usr/bin/env python

import sys
import serial
import platform
import glob
import time
import re
import os.path
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from IMUPacket import *
from IMUManager import *
from GloveAPI import GloveAPI
from Ui_GloveServer import *
from threading import Thread,Lock
from socket import *
from struct import *

#HOST = '192.168.1.147'
HOST = '127.0.0.1'
PORT = 5120

class DataWorker(Thread):
    '''
    Worker thread for retrieving results from the IMU.
    This thread runs a loop that captures data from the IMU and
    processes it into packets. it then takes the important IMU
    results and passes it to a function on the data object. This
    function uses a lock object to protect access from multiple
    threads, and serves as the data transfer point between this thread
    and the display thread.

    An enhancement would be to have this tread JUST capture the data
    and then have another thread or more handle the processing of the
    data, even the basic conversion and extraction of the data. Normally
    this might not be a good idea, but since I have an 8 Core machine
    at home, this makes a ton of sense...
    '''

    def __init__(self,api,dataObj,rate):
        super(DataWorker,self).__init__()

        self.api = api
        self.dataObj = dataObj
        self.stopme = False
```

```
        self.rate = rate

    def run(self):
        '''
        Do the hard work..
        '''
        print("worker started..")
        api = GloveAPI()

        while not api.openPort():
            print("Open port failed.. retry")

        print("Clear Packet Engine")
        api.clearIMUPacketEngine()
        print("Set Rate")
        api.rate(self.rate)
        print("Start Stream")
        api.streamstart(True)
        print("Set WD")
        api.StreamWD()

        while self.stopme == False:

            packet = api.getIMUPacket()
            if packet:
                data = packet.MeasuredData()
                self.dataObj.newPacket(packet.numIMUs,packet.pkID,data)
                api.StreamWD()
            else:
                '''
                A Timeout occured, see if I can recover
                '''
                print("Empty data returned")
                api.streamstart(True)
                api.StreamWD()

        print("Worker asked to quit")

class SocketWorker(Thread):
    '''
    Worker thread for accepting socket connections.
    Will listen, then accept connections. While a connection
    is established, it will read data and then send it back.
    '''

    def __init__(self,parent):
        super(SocketWorker,self).__init__()

        self.parent  = parent

    def run(self):
        '''
        Do the hard work..
```

```
        '''


        print ("Setting up the socket")
        s = socket(AF_INET, SOCK_STREAM)
        s.bind((HOST,PORT))

        while True:
            print ("Waiting for a connection")
            s.listen(1)
            conn,addr = s.accept()
            print "Connected by ",addr

            packetCount = 0

            while 1:
                data = conn.recv(1024)
                if data:
                    if re.match("start",data):
                        self.parent.sStart()
                    elif re.match("stop",data):
                        self.parent.sStop()
                    elif re.match("data",data):
                        [id,d] = self.parent.sData()
                        if d:
                            packetCount = packetCount + 1
                            if packetCount % 100 == 0:
                                print("Total of %d packets sent." % packetCount)
                            num = conn.send(pack("BHH",0xb7,len(d),id)+d)
                            #print("Len d:%d number sent:%d" % (len(d), num))
                        else:
                            # Send a null packet, nothing left to send.
                            conn.send(pack("BHB", 0xb7,0,0))
                            #print("Sent null packet")
                    elif re.match("quit",data):
                        print("Exiting the socket connection")
                        conn.close()
                        return
                    else:
                        conn.send("Unknown command")
                else:
                    break

            conn.close()

class GloveServer(QMainWindow,
                       Ui_GloveServer):
    def __init__(self,parent=None):
        super(GloveServer, self).__init__(parent)

```

```
        self._running = False
        self.ser = None

        self.step = 0.01
        self.data = []
        self.pData = None
        self.numPackets = 0
        self.numIMUs = 0
        self.setupUi(self)
        #self.api = GloveAPI()
        #self.api.initHardware()
        self.editPackets.setText("0")
        self.editRate.setText("40")
        self.editNumIMUs.setText("0")

        self.btnStartStop.setText("Start")

        self.connect(self.btnStartStop, SIGNAL("clicked()"),
                     self.StartStop)

        self.connect(self.btnQuit, SIGNAL("clicked()"),
                     self.OnExit)

        self.lock = Lock()

        self.timer = QTimer(self)
        self.connect(self.timer,
                     SIGNAL("timeout()"),
                     self.onTimer)
        self.timer.start(1)

        self.sockworker = SocketWorker(self)
        self.sockworker.start()
        self.worker = None

    def OnExit(self):
        if self.worker:
            self.worker.stopme = True
            print("Stopped the worker")
        else:
            print("No worker to stop")
        s = socket(AF_INET, SOCK_STREAM)
        s.connect((HOST, PORT))
        s.send('quit')

    def newPacket(self,numIMUs,id,data):
        self.lock.acquire()
        self.numPackets = self.numPackets + 1
        self.numIMUs = numIMUs
        if len(self.data) == 10:
            """Shift the elements down, append new element to end """
            self.data = self.data[1:]
        self.data.append([id,data])
        self.lock.release()
```

```

    def sStart(self):
        self._Start()

    def sStop(self):
        self._Stop()

    def sData(self):
        self.lock.acquire()
        if self.data:
            [id,d] = self.data.pop(0)
        else:
            [id,d] = [0,None]
        self.lock.release()
        return [id,d]

    def _Start(self):
        self.lock.acquire()
        self.numPackets = 0
        self.data = []
        self.lock.release()

        if self._running == False:
            self._running = True

            self.btnStartStop.setText("Stop")
            '''
            The speed here is not critical, since we can process multiple
            values for each timer even. The timer locks the data, grabs all
            of the values present, then unlocks it.. running this loop faster
            would probably only serve to increase the overhead. It might make
            the viewer smoother, but I doubt it, since a 25ms update rate is
            faster than we can really discern anyway, assume we can discern a
            30Hz update rate...
            '''
            srate = self.editRate.text()
            try:
                rate = int(srate)
            except:
                rate = 100
                self.editRate.setText("%d" % rate)

            self.worker = DataWorker(None,self,rate)
            self.worker.start()

    def _Stop(self):
        if self._running:
            self.btnStartStop.setText("Start")
```

```
            self._running = False
            ''' Signal the worker thread to stop, then wait for it '''

            print("Telling worker to stop..")
            self.worker.stopme = True
            self.worker.join()
            print("Worker done")
            self.worker = None

    def StartStop(self):
        if self._running:
            self._Stop()
        else:
            self._Start()

    def onTimer(self):
        self.editPackets.setText("%d" % self.numPackets)
        self.editNumIMUs.setText("%d" % self.numIMUs)


if __name__ == "__main__":
    import sys

    app = QApplication(sys.argv)
    mw = GloveServer()
    mw.show()
    mw.raise_()
    app.exec_()
```

Code Listing A.2: IMU Manager

```
#!/usr/bin/env python

import serial
import platform
import glob
import time
import re
import numpy
from GloveAPI import GloveAPI

class IMUManager(object):
    def __init__(self,api):
        super(IMUManager,self).__init__()

        self.api = api

    def Configure(self):
        pass

    def StreamOn(self,bOn = True):
        self.api.stream(bOn)
```

```

    def StreamCont(self,bOn = True):
        if bOn:
            print("Started streaming..")
            self.api.sendPacket(b'streamcont 1\n')
        else:
            print("Ended streaming..")
            self.api.sendPacket(b'streamcont 0\n')

    def ReadIMUData(self):
        '''
        Need to read the current state of the streams..
        Need to turn on streaming first of course.

        '''
        data = self.api.getIMUPacket()
        return data

if __name__ == "__main__":
    api = GloveAPI()
    api.initHardware()

    print ("Done")
```

Code Listing A.3: Application Programming Interface

```
#!/usr/bin/env python

import sys
import serial
import platform
import glob
import time
import re
import os.path
from IMUPacket import *
from optparse import OptionParser
import numpy as np
#import scipy as sp
#import scipy.io

class GloveAPI(object):

    def __init__(self):
        super(GloveAPI,self).__init__()
        self.ser = None

        self.pktrd = IMUPacketRead()

    def openPort(self,baud=115200,port=None,retries=20):
        if not port:
            port = self.getPorts()

```

```
        nTries = 0
        while nTries < retries:
            try:
                self.ser = serial.Serial(port,baud, timeout=1)
                if self.ser.isOpen():
                    print("Opened port %s at %d baud" % (port,baud))
                else:
                    print("Serial Port did not open..!")
                    raise("Bad thing happend.")
                #self.ctrl = ControllerAPI.ControllerComm(port,9600, timeout=1)
            except:
                raise "Could not open serial port!"
                self.ser = None
                return False

            retc = self.initHardware()
            if not retc:
                print("Init Failed. Reset Serial port")
                self.ser = None
                nTries = nTries + 1
            else:
                return True

    def initHardware(self):
        '''
        The first time we write to the hardware it appears that the FTDI chip resets
        it.. this has been problematic. It requires about 3 seconds for the hardware
        to come up, so I do a write, then a read with a long timeout, and see what happens.
        This should insure we are ready to run....
        '''
        self.ser.timeout = 3
        print("Timeout set to 3 seconds. Getting Time")
        retc = self.sendPacket(b'gettime\n')
        if not retc:
            retc = self.sendPacket(b'gettime\n')
        print("Timeout done. Restore to 1 second")
        self.ser.timeout = 1

        return retc

    def getPorts(self):
        if platform.system() == 'Darwin':
            """scan for available ports. return a list of device names."""
            ports = glob.glob('/dev/tty.usbserial-A400*')
            return ports[0]
        elif platform.system() == 'Windows':
            return "COM11" # No super good way to determine this..

    def sendPacket(self,packet):
```

```
        self.ser.write(packet)

        while (1):
            t = time.time()
            retval = self.ser.readline()
            try:
                tdiff = time.time() - t
                if tdiff > self.ser.timeout:
                    print("Timeout probably occured:%f" % tdiff)
                    return None

                retval = retval.decode()
                m = re.search("Ok|Fail",retval,re.IGNORECASE)
                if m:
                    print("{0} Result:{1}".format(packet.decode(),
                        retval))
                    break
                else:
                    print("{0}".format(retval))
                    pass
            except:
                # Invalid format, cannot decode it..
                print("Exception")
                return None

        return retval

    def i2crd(self,port,ID, addr, nbytes = 1):
        packet = "i2crd {0} {1} {2} {3}\n".format(port, ID, addr,
            nbytes)
        retval = self.sendPacket(packet.encode('utf-8'))

        m = re.search("Ok:(.*)\r\n",retval)
        if m:
            dvals = m.group(1)
            dvals = dvals.strip()
            dlist = []
            dlist = dvals.split()
            vallist = []
            for d in dlist:
                try:
                    vallist.append(int(d,16))
                except:
                    pass
            return vallist

        return None

    def i2cwr(self,port,ID,addr,data):
        if isinstance(data,(list,tuple)):
            data = " ".join(data)

        packet = "i2cwr {0} {1} {2} {3}\n".format(port, ID, addr,
            data)
```

```
        retval = self.sendPacket(packet.encode('utf-8'))

        if retval:
            m = re.search("Ok:(.*)\r\n",retval)
            if m:
                return True

        return False

    def streamstart(self,bUseGyro = False):
        if bUseGyro:
            retval = self.sendPacket(b'streamstart 1\n')
        else:
            retval = self.sendPacket(b'streamstart 0\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False

    def StreamWD(self):
        self.ser.write(b'wd\n')
        #print("Reset watchdog")
        return True

    def streamstop(self):
        retval = self.sendPacket(b'streamstop\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False

    def rate(self,nHz):
        print("Setting rate to %d" % nHz)
        retval = self.sendPacket(b'rate %d\n' % nHz)

        if retval:
            m = re.search("Ok",retval)
            print("Retval from rate setting:%s" % retval)
            if m:
                return True

    def fiforeset(self):
        retval = self.sendPacket(b'fiforeset\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False
```

```python
    def fifoenable(self):
        retval = self.sendPacket(b'fifoenable\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False

    def debug(self,bOn):
        if bOn:
            retval = self.sendPacket(b'debug 1\n')
        else:
            retval = self.sendPacket(b'debug 0\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False

    def reset(self):
        while (1):
            print ("Attempting Reset..")
            retval = self.sendPacket(b'reset\n')
            self.ser.write(b'checkids\n')

            bFail = False
            for x in range(0,12):
                retval = self.ser.readline()
                print ("Checkid returned:%s for %d" % (retval,x))
                try:
                    m = re.search("NAck",retval,re.IGNORECASE)
                    if m:
                        # One of these fails..
                        bFail = True
                except:
                    # Invalid format, cannot decode it..
                    bFail = True

            if not bFail:
                self.configimu()
                return True

    def startTimeToClockTime(self,startTime):
        minutes = startTime * 5
        hr = int(minutes / 60)
        min = minutes % 60
        if hr > 11:
            ampm = 'pm'
            hr = hr = 12
        else:
            ampm = 'am'
```

```
        clkTime = "{0:02d}:{1:02d}{2}".format(hr,min,ampm)
        return clkTime

    def getTime(self):

        retval = self.sendPacket(b'gettime\n')
        m = re.search("Ok:(\d+)",retval)
        if m:
            dt = m.group(1)
            try:
                unixTime = int(dt)
            except:
                print("Failed to convert datatime value:%s to integer
                    " % dt)
                return None

            tm = time.localtime(unixTime)
            timeString = time.strftime("%b %d,%Y %H:%M",tm)

            return timeString
        else:
            return None

    def getBaud(self):

        retval = self.sendPacket(b'baudquery\n')
        if retval:
            m = re.search("Ok:(.*)",retval)
            if m:
                dt = m.group(1)
                print("Baud Values:%s" % dt)

        return None

    def setTime(self):
        timeval = time.time() - time.timezone
        packet = "settime   %d\n" % int(timeval)
        retval = self.sendPacket(packet.encode('utf-8'))
        if retval:
            if re.search("Fail",retval):
                return None
            return "Ok"
        else:
            return False

    def configimu(self):
        retval = self.sendPacket(b'configimu\n')

        if retval:
            m = re.search("Ok",retval)
            if m:
                return True
        return False

```

```

    def clearIMUPacketEngine(self):
        self.pktrd.Clear()

    def getIMUPacket(self):
        self.pktrd.Clear()
        t = time.time()
        while (self.pktrd.isValidPacket == False):
            self.pktrd.getChars(self.ser)
            tdiff = time.time() - t
            if tdiff > self.ser.timeout:
                print("Timeout occurred:%f" % tdiff)
                return None

        return self.pktrd.packet

def GetApi():
    return GloveAPI()

def showPacket(p,keys):
    #print("Packet: ID:%d Type:0x%x CRC:0x%x DataLen:%d" % (p.pkID, p
        .pkType, p.pkCRC, len(p.pkData)))
    imudata = p.Results()
    wd = int(p.WatchDog)
    strings = []
    for i in imudata:
        strings.append("\t".join(["%x" % i[k] for k in keys]))
    print("Wd:%d" % wd + " " + "\t".join(strings))

def printPacket(f,fkeys,p):
    imudata = p.Results()
    strings = []
    for i in imudata:
        strings.append("\t".join(["%d" % i[k] for k in fkeys]))
    f.write("\t".join(strings) + "\n")

def testIMU(api):
    #api.fiforeset()
    #api.rate(10)
    #api.configimu()
    #api.streamWD()
    #api.fifoenable()
    nToRead = 2
    num2Read = 2
    fiforeadcounter = 20
    for z in range (100):
        vals = api.i2crd(0,210,58,2)
        if vals:
            s = " ".join(["%8d" % x for x in vals])
            fifoSize = vals[0] << 8 or vals[1]
            print ("Fifo Size %d: " % (fifoSize) + s)

            if fifoSize > 4:
                fifoSize = fifoSize - 2
```

```
                nToRead = fifoSize - fifoSize % 2
                if nToRead > 20:
                    nToRead = 16

                if nToRead > 0:
                    print ("Reading %d from fifo\n" % nToRead)
                    fvals = api.i2crd(0,210,60,nToRead)
                    if fvals:
                        s = ",".join(["%8d" % x for x in fvals])
                        print("Fifo:" + s + "\n")
                    fiforeadcounter = fiforeadcounter - 1
                    if fiforeadcounter == 0:
                        print ("Done")
                        break

            if vals[0] == 2:
                pass
                #break
    api.fiforeset()

def AutoIncrementFile(infile):
    if os.path.isfile(infile):
        ''' We have some work to do.. to autoincrement the file '''
        dir = os.path.dirname(options.outfile)
        fullname = os.path.basename(options.outfile)
        (fname,ext) = os.path.splitext(fullname)

        '''
        List the directory, then search only for those that include the base part of the
        filename, so File File1 File2 File3, etc would all be returned
        '''

        maxIndex = 0
        for f in os.listdir(dir):
            m = re.search(fname,f)
            if m:
                (fbase,e) = os.path.splitext(f)
                m = re.search("(\d+)$",fbase) # Find trailing number
                if m:
                    idx = int(m.group(1))
                    if idx > maxIndex:
                        maxIndex = idx

        index =maxIndex + 1

        outfile = os.path.join(dir,fname + "%d" % index + ext)

        print("%s incremented to %s" %(infile,outfile))

        return outfile
    else:
        return infile
```

```python


if __name__ == "__main__":

    usage = "usage: %prog -n -m -s]"
    parser = OptionParser(usage)
    parser.add_option("-r", dest="rate",
                      action="store",
                      type="int",
                      default=100,
                      help="Sample Rate")
    parser.add_option("-c", dest="count",
                      action="store",
                      type="int",
                      default=200,
                      help="Number to read.")
    parser.add_option("-p",dest="path",
                       action="store",
                       type="string",
                       default="DataCollection",
                       help="Directory to output debug data to")
    parser.add_option("-f",dest="outfile",
                       action="store",
                       type="string",
                       default="",
                       help="File to output debug data to")
    parser.add_option("-n",dest="dataname",
                      action="store",
                      type="string",
                      default="arr",
                      help="Name to use for the matlab data"
    )
    parser.add_option("-v",action="store_false",dest="verbose")
    parser.add_option("-s",action="store_true",dest="stop")

    (options, args) = parser.parse_args()

    if options.outfile == "":
        options.outfile = os.path.join(options.path,options.dataname
            + ".mat")

    #outfile = AutoIncrementFile(options.outfile)
    outfile = options.outfile

    print("Get API")
    api = GloveAPI()
    print("Init HW")
    api.initHardware()
    print("Get Baud")
    api.getBaud()

    print ("Got this far, established communication")

    if options.stop:
```

```
        api.streamstop()
        sys.exit(0)

    numToRead = options.count

    if options.outfile:
        writeFile = True
    else:
        writeFile = False

    api.clearIMUPacketEngine()
    keys = ['sentinal','temp','gx','gy','gz','ax','ay','az','footer']
    #keys = ['gx','gy','gz','ax','ay','az']
    #packets = []
    t = time.time()

    fkeys = ['temp','gx','gy','gz','ax','ay','az']

    api.rate(options.rate)
    api.streamstart(True)
    api.StreamWD()

    data = np.zeros((numToRead,1+6*7))

    nLeft2Stream = numToRead
    api.StreamWD()
    for x in range(0,numToRead):

        api.StreamWD()
        p = api.getIMUPacket()
        nLeft2Stream = nLeft2Stream - 1

        '''
        Reset the watchdog each time. No reason not to since the serial link to the
        unit is not used for anything except this.
        '''
        api.StreamWD()

        if p:
            #if writeFile:
                #printPacket(f,fkeys,p)
            #packets.append(p)
            showPacket(p,keys)

            '''
            Append the data to my potential Matlab array
            '''
            imudata = p.Results()
            imuidx = 0
            data[x][0] = p.pkID

            for i in imudata:
                #print("Adding IMU %d of %d\n" % (imuidx,x))
```

```
                colidx = 0
                for k in fkeys:
                    data[x][7*imuidx+colidx+1] = i[k]
                    colidx = colidx + 1
                imuidx = imuidx + 1
        else:
            print "Bad Packet"

    tdiff = time.time() - t

    if writeFile:

        if os.path.isfile(outfile):
            os.unlink(outfile)

        sp.io.savemat(outfile, mdict={
            'Data': data,
            'Name':options.dataname,
            'T':1.0/float(options.rate)})
        #f.close()

    api.streamstop()

    print("Total time:%6.6f Time per Packet:%6.6f" % (tdiff,tdiff/x))

    print ("Done")
```

Code Listing A.4: Packet Handling

```
#!/usr/bin/env python

import serial
import platform
from struct import *
import numpy as np
import sys

class IMUPacket:
    def __init__(self):
        self.pkID = 0
        self.pkType = None
        self.pkBytes = 0
        self.pkData = []
        self.WatchDog = 0
        self.pkCRC = 0
        self.lenPerIMU = 9
        self.numIMUs = 6 # Default value.

        ''' Numpy transform matrices '''
        self.handgyro = np.array([[ 0,-1,0],[-1,0,0],[0,0,-1]])
        self.handacc =  np.array([[-1, 0,0],[ 0,1,0],[0,0,-1]])

        self.finggyro = np.array([[0,1,0],[1,0,0],[0,0,-1]])
```

```
        self.fingacc =  np.array([[1,0,0],[0,-1,0],[0,0,-1]])

    def RawData(self):
        if self.pkType == 0xB7:
            numIMUs = len(self.pkData)/(self.lenPerIMU*2)
            numVals = (len(self.pkData))/2
            return self.pkData

    def IMUIndexes(self,nIMUs,lenPerIMU):
        i = []
        for x in range(0,nIMUs):
            offset=x*lenPerIMU
            [i.append(y) for y in range(offset+2,offset+8)]
        return i

    def IMUData(self):
        """
        This data is for the Python capture and store routine.
        I return the data back as a python array of 6 values
        per IMU. This routine is the same as "MeasuredData", except
        that measured data returns back a packed array.
        """
        values = self.PacketToArray();
        if values:
            '''
            I want to re-pack the data. Currently we have
            a sentinal, temp, 3* gyro, 3* acc, bogus
            Lets get rid of the two sentinals and temp to just get
            3*gyro and 3* acc

            I need to reorganize the data to get the axis in their propper
            order.
            '''


            indexes = self.IMUIndexes(self.numIMUs,self.lenPerIMU)
            newVals = [values[i] for i in indexes]

            imuarray = np.array(newVals ).reshape(self.numIMUs,6)

            hand = imuarray[0]
            t = hand.reshape(2,3)
            t[0] = self.handgyro.dot(t[0])
            t[1] = self.handacc.dot(t[1])
            imuarray[0] = t.ravel()

            for x in range(1,self.numIMUs):
                imu = imuarray[x]
                t = imu.reshape(2,3)
                t[0] = self.finggyro.dot(t[0])
                t[1] = self.fingacc.dot(t[1])
                imuarray[x] = t.ravel()

```

```
            return imuarray
        else:
            return None

    def MeasuredData(self):
        imuarray = self.IMUData()
        if not imuarray == None:
            sPack = ">" + ''.join(["6h" for x in range(0,imuarray.size/6)])
            imuarray = imuarray.astype('int16')
            data = pack(sPack,*imuarray.ravel())
            return data
        else:
            return None

    def PacketToArray(self):
        if self.pkType == 0xB7:
            '''
            The packet data consists of a single byte "Mask", and then N 2-byte unsigned
            values.
            The Unpack will unpack as many values as there are, since the values are all 2 bytes
            and the length of the data packet is 2*#values + 1
            '''
            Values = []
            #print ("Length of packet:%d" % len(self.pkData))
            self.numIMUs = len(self.pkData)/(self.lenPerIMU*2)
            #print ("Number of IMU's:%d" % numIMUs)
            numVals = (len(self.pkData))/2
            sUnpack = ">" + ''.join(["H8h" for x in range(0,self.numIMUs)])
            (Values) = unpack(sUnpack,self.pkData)
            return Values
        else:
            print("Invalid packet type:%d" % self.pkType)
            return None

    def Results(self):
        values = self.PacketToArray()
        if values:

            '''
            At this point, the Mask will tell us which values are valid
            '''

            imuData = []

            keys = ['sentinal','temp','gx','gy','gz','ax','ay','az','footer']
            numIMUs = len(Values)/self.lenPerIMU
            for x in range(0,numIMUs):
                imu = {}
```

```
                y = 0
                for k in keys:
                    imu[k] = Values[x*self.lenPerIMU+y]
                    y = y +1
                imuData.append(imu)

            return imuData

        '''
        Only handling one type for now..
        '''
        return None

class IMUPacketRead:

    sStart, sFndS, sFndN, sFndP, sPkType, sPkID, sPkSize, sPkData, sPkCRC, sPkDone = range(0,10)

    def __init__(self):
        self.pkLoc = 0
        self.pkState = IMUPacketRead.sStart
        self.packet = IMUPacket()
        self.isValidPacket = False
        self.verbose = False

    def isValid(self):
        return self.isValidPacket

    def getChars(self,ser):

        while not self.isValidPacket and ser.inWaiting() > 0:
        #while not self.isValidPacket:
            if self.pkState == IMUPacketRead.sStart:
                byte = ser.read(1)
                if byte == 'S':
                    self.pkState = IMUPacketRead.sFndN
                    if self.verbose:
                        print ("Found S")
            elif self.pkState == IMUPacketRead.sFndN:
                byte = ser.read(1)
                if byte == 'N':
                    self.pkState = IMUPacketRead.sFndP
                    if self.verbose:
                        print ("Found N")
                else:
                    self.pkState = IMUPacketRead.sStart
                    if self.verbose:
                        print ("Returning to Start")
            elif self.pkState ==  IMUPacketRead.sFndP:
                byte = ser.read(1)
                if byte == 'P':
                    self.pkState = IMUPacketRead.sPkType
                    if self.verbose:
                        print ("Found P")
```

```
            else:
                self.pkState = IMUPacketRead.sStart
                if self.verbose:
                    print ("Returning to Start")
        elif self.pkState == IMUPacketRead.sPkType:
            pkType = ser.read(1)
            self.packet.pkType = unpack('>B',pkType)[0]
            self.pkState = IMUPacketRead.sPkID
            if self.verbose:
                print ("Found PkType:0x%x" % self.packet.pkType)
        elif self.pkState == IMUPacketRead.sPkID:
            ID = ser.read(1)
            self.packet.pkID = unpack('>B',ID)[0]
            self.pkState = IMUPacketRead.sPkSize
            if self.verbose:
                print ("Found pkID:0x%x" % self.packet.pkID)
        elif self.pkState == IMUPacketRead.sPkSize:
            pkBytes = ser.read(1)
            self.packet.pkBytes = unpack('>B',pkBytes)[0]
            if self.verbose:
                print("PkBytes:0x%x" % self.packet.pkBytes)
            if self.packet.pkBytes > 0:
                '''
                Doing a direct read here, rather than defering the read and
                reading a seperate bytes... should be mucho faster.
                '''
                if self.verbose:
                    print ("Reading %d Bytes" % self.packet.pkBytes)
                try:
                    #self.packet.pkData = []
                    #for x in range(0,self.packet.pkBytes):
                    #    b = ser.read(1)
                    #    self.packet.pkData.append(unpack('>B',b)[0])
                    d = ser.read(self.packet.pkBytes)
                    self.packet.pkData = d
                    self.pkState = IMUPacketRead.sPkCRC
                    #self.pkState = IMUPacketRead.sPkDone
                    if self.verbose:
                        print ("Going to PkDone")
                except:
                    '''
                    What happens if I time out... want to handle this someday
                    '''
                    self.pkState = IMUPacketRead.sStart
                    print("Exception while reading packet data")
            else:
                self.pkState = IMUPacketRead.sPkDone
                if self.verbose:
                    print ("Packet Size == 0, Returning to Start"
```

```
                           )
            elif self.pkState == IMUPacketRead.sPkCRC:
                pkWD = ser.read(1)
                self.packet.WatchDog = unpack('>B',pkWD)[0]
                pkCRC = ser.read(2)
                self.packet.pkCRC = unpack('>H',pkCRC)[0]
                if self.verbose:
                    print("CRC Code:0x%x" % self.packet.pkCRC)
                self.pkState = IMUPacketRead.sPkDone
                if self.verbose:
                    print ("Going to PkDone")

            if self.pkState == IMUPacketRead.sPkDone:
                self.isValidPacket = True

        return self.isValidPacket

    def Clear(self):
        self.isValidPacket = False
        self.packet = IMUPacket()
        self.pkState = IMUPacketRead.sStart
```

Code Listing A.5: User Interface

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'GloveServer.ui'
#
# Created: Tue May 17 17:08:16 2011
#      by: PyQt4 UI code generator 4.7.7
#
# WARNING! All changes made in this file will be lost!

from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_GloveServer(object):
    def setupUi(self, GloveServer):
        GloveServer.setObjectName(_fromUtf8("GloveServer"))
        GloveServer.resize(368, 230)
        self.centralwidget = QtGui.QWidget(GloveServer)
        self.centralwidget.setObjectName(_fromUtf8("centralwidget"))
        self.label = QtGui.QLabel(self.centralwidget)
        self.label.setGeometry(QtCore.QRect(50, 30, 62, 16))
        self.label.setObjectName(_fromUtf8("label"))
        self.editRate = QtGui.QLineEdit(self.centralwidget)
        self.editRate.setGeometry(QtCore.QRect(200, 30, 113, 22))
        self.editRate.setObjectName(_fromUtf8("editRate"))
        self.label_2 = QtGui.QLabel(self.centralwidget)
```

```
        self.label_2.setGeometry(QtCore.QRect(50, 80, 111, 16))
        self.label_2.setObjectName(_fromUtf8("label_2"))
        self.editPackets = QtGui.QLineEdit(self.centralwidget)
        self.editPackets.setGeometry(QtCore.QRect(200, 70, 113, 22))
        self.editPackets.setObjectName(_fromUtf8("editPackets"))
        self.btnStartStop = QtGui.QPushButton(self.centralwidget)
        self.btnStartStop.setGeometry(QtCore.QRect(60, 160, 114, 32))
        self.btnStartStop.setObjectName(_fromUtf8("btnStartStop"))
        self.btnQuit = QtGui.QPushButton(self.centralwidget)
        self.btnQuit.setGeometry(QtCore.QRect(200, 160, 114, 32))
        self.btnQuit.setObjectName(_fromUtf8("btnQuit"))
        self.label_3 = QtGui.QLabel(self.centralwidget)
        self.label_3.setGeometry(QtCore.QRect(50, 120, 111, 21))
        self.label_3.setObjectName(_fromUtf8("label_3"))
        self.editNumIMUs = QtGui.QLineEdit(self.centralwidget)
        self.editNumIMUs.setGeometry(QtCore.QRect(200, 120, 113, 22))
        self.editNumIMUs.setObjectName(_fromUtf8("editNumIMUs"))
        GloveServer.setCentralWidget(self.centralwidget)
        self.menubar = QtGui.QMenuBar(GloveServer)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 368, 22))
        self.menubar.setObjectName(_fromUtf8("menubar"))
        GloveServer.setMenuBar(self.menubar)
        self.statusbar = QtGui.QStatusBar(GloveServer)
        self.statusbar.setObjectName(_fromUtf8("statusbar"))
        GloveServer.setStatusBar(self.statusbar)

        self.retranslateUi(GloveServer)
        QtCore.QObject.connect(self.btnQuit, QtCore.SIGNAL(_fromUtf8("clicked()")), GloveServer.close)
        QtCore.QMetaObject.connectSlotsByName(GloveServer)

    def retranslateUi(self, GloveServer):
        GloveServer.setWindowTitle(QtGui.QApplication.translate("GloveServer", "MainWindow", None, QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("GloveServer", "Rate", None, QtGui.QApplication.UnicodeUTF8))
        self.label_2.setText(QtGui.QApplication.translate("GloveServer", "Packets Captured", None, QtGui.QApplication.UnicodeUTF8))
        self.btnStartStop.setText(QtGui.QApplication.translate("GloveServer", "Start", None, QtGui.QApplication.UnicodeUTF8))
        self.btnQuit.setText(QtGui.QApplication.translate("GloveServer", "Quit", None, QtGui.QApplication.UnicodeUTF8))
        self.label_3.setText(QtGui.QApplication.translate("GloveServer", "Num IMUS", None, QtGui.QApplication.UnicodeUTF8))
```

APPENDIX B

Panda3D™ Python™ SOURCE CODE

Panda3D™ Python™ SOURCE CODE

Code Listing B.1: 3D Server

```
#!/usr/bin/env python


#Importing math constants and functions
#from direct.stdpy import threading
#from direct.stdpy import socket
import socket

# Thread class that executes processing
class SocketListener(threading.Thread):
    """Worker Thread Class."""
    def __init__(self, destClass):
        """Init Worker Thread Class."""
        print("Initing the thread")
        threading.Thread.__init__(self)
        self.destClass = destClass
        print("Done..")
        # This starts the thread running on creation, but you could
        # also make the GUI thread responsible for calling this
        self.start()

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread. Simulation of
        # a long process (well, 10s here) as a simple loop - you will
        # need to structure your processing so that you periodically
        # peek at the abort variable
        print("Opening the socket")
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind(("", 5010))
        server_socket.listen(5)

        print ("Client Socket Setup")

        while 1:
            client_socket, address = server_socket.accept()
            print "I got a connection from ", address
            data = client_socket.recv(512)
            while data:
                cmdFound = False
                m = re.search("pos (\d+)",data,re.IGNORECASE)
                if m:
                    pos = int(m.group(1))
                    #wx.PostEvent(self._notify_window,PositionEvent(pos) )
                    #print ("Updated Position to %d" % pos)
                    destClass.notifyPos(pos)
                    cmdFound = True
```

```
m = re.search("load \"(.*)\"", data, re.IGNORECASE)
if m:
    # Re-load a new file
    #wx.PostEvent(self._notify_window,LoadEvent(m.group(1)))
    cmdFound = True

if re.search("play",data,re.IGNORECASE):
    #wx.PostEvent(self._notify_window,ControlEvent(2))
    print("Play Video")
    cmdFound = True
elif re.search("pause",data,re.IGNORECASE):
    #wx.PostEvent(self._notify_window,ControlEvent(1))
    print("Pause Video")
    cmdFound = True
elif re.search("stop",data,re.IGNORECASE):
    #wx.PostEvent(self._notify_window,ControlEvent(0))
    print("Stop Video")
    cmdFound = True

if not cmdFound:
    print("Unknown Command:%s" % data)

data = client_socket.recv(512)
```

APPENDIX C

MATLAB™ CODE

MATLAB™ CODE

Code Listing C.1: GloveGui Top

```
classdef GloveGui < GloveGuiBase
    % GloveGui Extend teh basic GloveGui to include live capture controls.
    % The GloveGuiBase provides the GUI setup code. This class adds more of
    % the logic to the class. A higher level class allowed me to isolate
    % the code better, and keep the files I work on smaller.

    % This class also allowed me to isolate the kinematics portions of the
    % logic, here I can override the IMU object with more specific
    % versions, or experimental versions.

    properties

        myTimer
        period

        imuObj

        hKinematics

    end

    methods
        function obj = GloveGui(imuObj)
            % Constructor. Takes as an argument the
            % video processing object which contains the images and such.
            obj = obj@GloveGuiBase();

            obj.hKinematics = HandKinematics();

            % Allow an alternate - or customized - imuObj to be passed in
            if nargin == 1
                obj.imuObj = imuObj;
            else
                obj.imuObj = GloveIMU();
            end
            obj.imuObj.SetHK(obj.hKinematics);
        end

        function delete(obj)
            % Clean up anything that needs to be fixed up. Close timers,
            % close the data capture, etc.

```

```
            display('Deleting GloveGui.');

            try
                clear(obj.imuObj);
                clear(obj.gData);
            catch e
            end
        end

        function StartLive(obj,periodms)
            % Play through the sequence with a timer. This one is cool. I
            % setup a callback timer, see the Play_cb function, that is
            % called once each period.. This gives a "frame rate". A normal
            % period is probably 1/30, or technically, 1/29.7 or so.

            period = periodms / 1000;
            obj.period = period;
            obj.gData.StartCapture(500);

            % Create some IMU Objects to manage the IMU calculations. This
            % is a seperate class that makes it easier to keep the logic
            % all organized, and will make it much easier to document
            % later. It won't be the fastest performing, but that isnt the
            % primary concern at this point.

            obj.imuObj.ResetGlove();

            obj.myTimer = timer(...
                'TimerFcn',@(src,event)Live_cb(obj,src,event),...
                'Period',period,...
                'ExecutionMode','fixedRate');

            start(obj.myTimer);
        end

        function Live_cb(obj,src,event)
            % Callback for timer.. called after each period,.

            try
                [cnt,acc,gyro] = obj.gData.GetData();
            catch e
                display(sprintf('Exception in GetData:%s',e.identifier));
            end

            if cnt
                for x = 1:cnt
                    obj.IMU_Update(acc(x,:),gyro(x,:),obj.period);
```

```
                end
            end
        end

        function IMU_Update(obj,acc,gyro,T)
            % This function will perform the incremental IMU calulcations,
            % update kalman gain matrices, etc. It will then update the
            % glove position values and then update the glove visual
            % position

            %obj.imuObj.UpdateAccData(imudata(1,4:6),T);
            try
                obj.imuObj.Update(gyro,acc,T);
            catch e
                display(sprintf('Exception in imuObj.Update:%s',e.identifier));
                return;
            end

            try
                obj.IMUPlot(acc,gyro);
            catch e
                display(sprintf('Exception in obj.IMUPlot:%s',e.identifier));
                return;
            end

            try
                obj.hKinematics.UpdatePanda();
            catch e
                display(sprintf('Exception in obj.UpdatePanda:%s',e.identifier));
                return;
            end

            % Update display of IMU Data, as a data check
            chk = abs(sum(reshape(acc,3,6)',2)') > 0.2;
            set(obj.ui.text('Data Check').edit,'String',sprintf('%d',chk));
            dcm = obj.imuObj.DCMBody2Inertial(obj.posIndex);
            myFormat = @(x) sprintf('%5.3f',x);
            tdcm = arrayfun(myFormat,dcm,'UniformOutput',0);
            obj.SetTableData('DCM',reshape(tdcm,3,3));
        end

        function StopLive(obj)
            % Sto the timer, and delete it. Stop the capture process.
            try
                stop(obj.myTimer);
                delete(obj.myTimer);
            catch e
                % No timer to delete I guess..
```

```
                delete(timerfind);
            end

            obj.gData.StopCapture();
        end

        function Restart(obj)
            display('Restarting IMU');
            obj.imuObj.Restart();
        end

        function Zero(ob)
        end

    end

end
```

Code Listing C.2: GloveGui Base

```
classdef GloveGuiBase < GuiBase
    % GloveGuiBase Extend the GuiBase class with Glove specicif code.
    % This class exends the GuiBase and builds a glove GUI.

    properties

        positions

        % GloveData object - contains the capture and manipulation
           code
        gData

        plotObjs
        posIndex

    end

    methods
        function obj = GloveGuiBase()
            % Constructor. Takes as an argument the
            % video processing object which contains the images and
               such.

            obj.InitGui();
            obj.plotObjs = {};
            obj.InitPlotObjects();

            obj.gData = GloveData();
            obj.posIndex = 1;
        end

        function delete(obj)
```

```
        % Clean up anything that needs to be fixed up. Close timers,
        % close the data capture, etc.

        display('Deleting GloveGuiBase.');

        try
            for x = 1:size(obj.plotObjs,2)
                delete(obj.plotObjs{x})
            end
        catch e
        end

        try
            clear(obj.imuObj);
            clear(obj.gData);
            %obj.gData.StopCapture();
        catch e
        end
    end

    function InitGui(obj)
        % Build the GUI. Note that the callbacks are for this object.
        % The size is fixed.. should query the screen size or
        % something..
        % I store all of the UI handles into a .ui parameter of this
        % object.

        figWidth = 1200;
        figHeight = 580;

        panelWidth = 0.2;
        panelLeft = 1 - panelWidth;
        obj.ui.Figure = figure(...
            'Visible','Off',...
            'Colormap',gray(256),...
            'Position',[100 100 figWidth figHeight]);

        set(obj.ui.Figure,...
            'KeyPressFcn',@(src,event)keypress_cb(obj,src,event));

        obj.ui.hPanelCtrl = uipanel('Title','Controls','FontSize',12,...
            'Parent',obj.ui.Figure,...
            'Units','normalized',...
            'Position',[panelLeft 0 panelWidth 1]);

        % Add one axes for the image display,
        % then add a tile of 6 axes for the Gyro and Acc data display,
        % these are 2 columns by 3 rows... finally, we have the
```

```
    panel

            obj.ui.plotWidth = panelLeft;
            obj.ui.plotLeft = 0;
            obj.ui.plotMargin = 0.02;
            obj.ui.plotVertMargin = 0.08;
            obj.ui.dataWidth = obj.ui.plotWidth/2;

            panelIdx = 0;
            obj.ui.hPanelGyro  = obj.AddTripleGraph('GyroAxes',[0 0 panelWidth 1],'Gyro','Gyro Axes');
            obj.ui.hPanelAcc   = obj.AddTripleGraph('AccAxes',[panelWidth 0 panelWidth 1],'Accelerometer','Accelerometer Axes');
            obb.ui.hPanelV     = obj.AddTripleGraph('VAxes',[2*panelWidth 0 panelWidth 1],'Velocity','Velocity Axes');
            obu.ui.hPanelPos   = obj.AddTripleGraph('PAxes',[3*panelWidth 0 panelWidth 1],'Position','Position Axes');

            editPos = 0;

            obj.ui.btnClose= obj.AddButton(...
                'Close',...
                0,...
                @(src,event)btnClose_cb(obj,src,event)...
            );
            obj.ui.startLive= obj.AddToggleButton(...
                'Start Live',...
                1,...
                @(src,event)startLive_cb(obj,src,event)...
            );
            obj.AddEditText('Data Check','1',2);
            obj.AddTable('DCM',eye(3),3,3);

            % Build a menu of the data sets and populate it with
            % a list of all of the data set names.

            mnu2 = uimenu('Label','Positions');
            obj.positions = {'Hand','Middle','Ring','Thumb','Pinkie','Index'};
            acc = {'1','2','3','4','5','6'};
            for k = 1:length(obj.positions)
                uimenu(mnu2,'Label',obj.positions{k},...
                    'Callback',@(src,event)positionMenu_cb(obj,k),...
                    'Accelerator',acc{k});
            end

            obj.AddComboBox('Position',obj.positions,6,2);
            h = obj.ui.list('Position').list;
            set(h,'Callback',@(src,event)positionsCombo_cb(obj,src,event));

```

```
            set(obj.ui.Figure,'Name','Gyro Glove Data Visualizer');

            % Make the GUI visible.
            set(obj.ui.Figure,'Visible','on');
            set(obj.ui.hPanelCtrl,'Visible','on');
        end

        function InitPlotObjects(obj)
            obj.plotObjs{1} = PlotData(obj.ui.GyroAxes, 500, 18,...
                [-500*2*pi/360 500*2*pi/360]);
            obj.plotObjs{2} = PlotData(obj.ui.AccAxes, 500, 18,[-2 2]);
            obj.plotObjs{3} = PlotData(obj.ui.VAxes, 500, 18,[-5 5]);
            obj.plotObjs{4} = PlotData(obj.ui.PAxes, 500, 18,[-180 180]);
        end

        function IMUPlot(obj,acc,gyro)
            obj.plotObjs{1}.UpdateData(gyro);
            obj.plotObjs{2}.UpdateData(acc);
            obj.plotObjs{3}.UpdateData(obj.imuObj.Velocities());
            %obj.plotObjs{4}.UpdateData(obj.imuObj.Positions());
            obj.plotObjs{4}.UpdateData(obj.imuObj.EulerAngles());
        end

        function StartLive(obj,periodms)
        end

        function StopLive(obj)
        end

        function positionMenu_cb(obj,posIndex)
            % Called by the menu item to update the index that selects
            % which IMU dataset to display on the graphs.
            obj.posIndex = posIndex;
            for x = 1:size(obj.plotObjs,2)
                obj.plotObjs{x}.UpdateIndex(posIndex)
            end
            display(sprintf('Updated position to %d',posIndex));
            h = obj.ui.list('Position').list;
            set(h,'Value',obj.posIndex);
        end

        function positionsCombo_cb(obj,src,event)
            % Called by the menu item to update the index that selects
            % which IMU dataset to display on the graphs.

            posIndex = get(src,'Value');
            obj.posIndex = posIndex;
            for x = 1:size(obj.plotObjs,2)
                obj.plotObjs{x}.UpdateIndex(posIndex)
            end
```

```
            display(sprintf('Updated position to %d',posIndex));
        end

        function startLive_cb(obj,src,event)
            if get(src,'Value') == get(src,'Max')
                obj.StartLive(40);
            else
                obj.StopLive();
            end
        end

        function btnClose_cb(obj,src,event)
            close(gcbf);
            obj.delete;
        end

        function keypress_cb(obj,src,event)
            if event.Character == 'r'
                % restart
                obj.Restart();
            elseif event.Character == 'z'
                obj.Zero();
            end
        end

        % Functions to override.
        function Restart(obj)
        end
        function Zero(obj)
        end
    end

end
```

Code Listing C.3: GUI Base

```
classdef GuiBase < handle
    %GuiBase Core GUI Class with helpers for creating the gui.
    % The GuiBase class provides the core features needed for a Gui.
    % is also provides a number of helper functions to make building a gui
    % easier, such as functions for adding buttons, text boxes, graphs,
    % etc. Any functions that apply to a generic GUI are put into this
    % class so that they are available to any GUI's I build in Matlab.

    properties
        ui
    end

    methods
```

```
        function obj = GuiBase()
            obj.ui = struct();
            obj.ui.text = containers.Map();
            obj.ui.tables = containers.Map();
        end

        function hPanel = AddTripleGraph(obj,uiName, pos, dName, title)
            % This function is pretty specific, but it is still genertic i
            % some respects. It adds a triple graph that I used to display
            % x,y,z or rho,theta,psi coordinates, etc.


            hPanel = uipanel('Title','Controls','FontSize',12,...
                'Parent',obj.ui.Figure,...
                'Units','normalized',...
                'Position',pos);

            % Define the Gyro Axes
            xyz = {'Gyro X', 'Gyro Y', 'Gyro Z'};
            obj.ui.(uiName) = [];
            for r = 1:3
                % Left is always just right of the Video Axes.
                % Bottom is 2/3, 1/3 and 0/3
                % Width and height are fixed.
                aop = [0 (r-1)/3 1 1/3];
                ap = aop;
                obj.ui.(uiName)(r) = axes(...
                        'DataAspectRatio', [1 1 1],...
                        'DrawMode','fast',...
                        'Visible','on',...
                        'Units','normalized',...
                        'Position',ap,...
                        'Parent', hPanel);
                %title(obj.ui.(uiName)(r),title);
            end
        end

        function h = AddToggleButton(obj,string,row,cb)
            h = uicontrol(obj.ui.hPanelCtrl,...
                'Style','togglebutton',...
                'String',string,...
                'Units','normalized',...
                'Position',[0.05 row*(0.1) 0.8 0.1],...
                'Visible','on',...
                'Callback',cb...
            );
        end

        function h = AddButton(obj,string,row,cb)
            h = uicontrol(obj.ui.hPanelCtrl,...
                'Style','pushbutton',...
```

```
                'String',string,...
                'Units','normalized',...
                'Position',[0.05 row*(0.1) 0.8 0.1],...
                'Visible','on',...
                'Callback',cb...
            );
        end

        function [h,u] = AddButtonGroup(obj,type,group,names,row)
            % A routine to make adding a set of butons easier.
            h = uibuttongroup('visible','on',...
                'Units','normalized',...
                'Position',[0.05 row*(0.1) 0.8 0.1],...
                'Parent',obj.ui.hPanelCtrl);

            nitems = length(names);
            for x = 1:nitems
                n = names{x};

                u(x) = uicontrol('Style',type,...
                    'String',n,...
                    'Units','normalized',...
                    'pos',[(x-1)*(1/nitems) 0 1/nitems 1],...
                    'Parent',h,...
                    'HandleVisibility','off');
            end
            set(h,'SelectedObject',[]);
            set(h,'Visible','on');
        end

        function AddEditText(obj,label,value,row)
            % Add a text box and label.
            s.text = uicontrol('Style','text',...
                'Parent',obj.ui.hPanelCtrl,...
                'String',label,...
                'Units','normalized',...
                'Position',[0.05 row*(0.1) 0.185 0.08]);

            s.edit = uicontrol('Style','edit',...
                'Parent',obj.ui.hPanelCtrl,...
                'String',value,...
                'Units','normalized',...
                'Position',[0.21  row*(0.1) 0.76 0.08]);

            obj.ui.text(label) = s;
        end

        function  AddComboBox(obj,label,values,row,height)
            % Add a text box and label.
            s.text = uicontrol('Style','text',...
                'Parent',obj.ui.hPanelCtrl,...
                'String',label,...
                'Units','normalized',...
                'Position',[0 row*(0.1) 0.3 0.08]);
```

```
            s.list = uicontrol('Style','listbox',...
                'Parent',obj.ui.hPanelCtrl,...
                'String',values,...
                'Value',1,...
                'Max',1,'Min',1,...
                'Fontsize',16,...
                'Units','normalized',...
                'Position',[0.3 row*(0.1) 0.7 height*0.1]);

            obj.ui.list(label) = s;
        end

        function h = AddTable(obj,label,data,row,height)
            h = uitable(obj.ui.hPanelCtrl,...
                'Data',data,...
                'Units','normalized',...
                'Position',[0 row*(0.1) 1 height*(0.1)]);
            obj.ui.tables(label) = h;
        end

        function SetTableData(obj,label,data)
            set(obj.ui.tables(label),'Data',data);
        end

        function s = GetValue(obj,label)
            s = get(obj.ui.text(label).edit,'String');
        end
    end

end
```

Code Listing C.4: Platform IMU

```
classdef PlatformIMU < GloveIMU
    % PlatformIMU Add even more specifics, such as the Kinematics.
    %

    properties
        % Values for initialization
        U
        V

        % Euler agles for first and second alignment
        euler1
        euler2

        markTime
    end

    methods
        function obj = PlatformIMU()
            display('Constructing PlatformIMU');
```

```
            obj = obj@GloveIMU();
        end

        function delete(obj)
            display('Deleting PlatformIMU');
        end

        function Restart(obj)
            obj.currTime = 0;
            obj.State = GloveState.InitialWait;
        end

        function eAngles = EulerAngles(obj,idx)
            % Caclculate and return a matrix of Euler angles, one row
                per
            % IMU and the columns are roll, pitch, yaw

            % I pass in the previous calculated values so that if we
              are
            % pitched up or down at about 90 degrees, we can use the
            % previous roll and yaw values.

            rad2deg = @(x) 360*(x/(2*pi));

            if nargin == 1
                eAngles = zeros(6,3);
                % Update the hand first
                DCM_I_H = obj.DCMBody2Inertial(1);
                obj.eAngles{1} = obj.dcm2Euler(DCM_I_H,obj.eAngles
                    {1});
                eAngles(1,:) = rad2deg(obj.eAngles{1});

                % Then update the fingers.
                for x = 2:6
                    % Convert DCM H to Inertial to DCM
                    DCM_H_D = obj.DCMDig2Hand(x,DCM_I_H);
                    obj.eAngles{x} = obj.dcm2Euler(DCM_H_D,obj.
                        eAngles{x});
                    eAngles(x,:) = rad2deg(obj.eAngles{x});
                end
            else
                if idx == 1
                    DCM_I_H = obj.DCMBody2Inertial(1);
                    obj.eAngles{1} = obj.dcm2Euler(DCM_I_H,obj.
                        eAngles{1});
                    eAngles = rad2deg(obj.eAngles{1});
                else
                    DCM_H_D = obj.DCMDig2Hand(x);
                    obj.eAngles{idx} = obj.dcm2Euler(DCM_H_D,obj.
                        eAngles{idx});
                    eAngles = rad2deg(obj.eAngles{idx});
                end
            end
        end
```

```

        function PlatformInit(obj)
            % The glove has been placed flat on the table with the fingers
            % and hand down on the table as much as practical. The
            % body-inertial DCM's shold be considered vertical, so any
            % offsets are in the platform to body. Take these measurements,
            % and then initialize the Body-Inertial to Z up,

            % Fingers
            acc = reshape(mean(obj.accHistory(1:40,:)),3,6)';
            for x = 1:6
                tacc = acc(x,:);
                dcm_i_p = obj.courseAlign(tacc);
                dcm_i_b = obj.courseAlign([0 0 -1]);
                obj.DCM_I_P{x} = dcm_i_p;
                obj.DCM_B_P{x} = dcm_i_p;
            end
        end

        function DCMUpdateAll(obj,gyro,acc,T)
            % Take in new gyro data and accelerometer data and update the
            % DCM's with it. Doing this just for the fingers and the thumb.
            % To make this better, I need to incorporate the Kinematics of
            % the hand so that I can insure that the fingers do not get out
            % of whack... and the DCM is constrained by the kinematics of
            % the hand.

            % function object to calculate the skew symetric matrix
            g = reshape(gyro,3,6)';
            a = reshape(acc,3,6)';
            for x = 1:6
                %obj.DCM_I_P{x} = obj.DCMUpdate(obj.DCM_I_P{x},g(x,:),T);
                obj.DCM_I_P{x} = obj.courseAlign(a(x,:));
            end
            obj.hKinematics.UpdateAngles(obj.EulerAngles());
        end

        function Update(obj,gyro,acc,T)
            % Update the set of IMU's with new gyro and accelerometer data.
            % The new data will be used to update the set of DCMs, as well
            % as the velocity and position values for each of the IMUs in
            % the system.
```

```

        % Note: The input format for the gyro and acc is one row per
        % IMU, and the 3 columns are the x,y, and z values.

        % Scale the accelerometer and gyro data. Subtract out any bias
        % values we have determined. For now the ACC Bias will be zero
        % for all settings.
        acc = acc-obj.accBias;

        gyro = gyro-obj.gyroBias;

        % I want to update the history in all state except the Idle
        % state. It makes it easier to make a seperate switch for this
        % operation rather than adding the UpdateHistory to all other
        % states of the main switch
        if ~(obj.State == GloveState.Idle)
            obj.UpdateHistory(gyro,acc);
        end

        switch obj.State
            case GloveState.Idle
                obj.ResetGlove();
                display('Place glove flat on the table.');
                obj.State = GloveState.InitialWait;

            case GloveState.InitialWait

                if obj.currTime > 3.0
                    if obj.isGloveStable(40)
                        display(sprintf('Glove Stable for init at %f',obj.currTime));
                        obj.PlatformInit();
                        obj.State = GloveState.IMURun;
                    end
                end
            case GloveState.InitialZero
            case GloveState.SecondZeroWait
            case GloveState.SecondZero
            case GloveState.IMURun

                % Update the DCM from the gyro data. In some cases,
                % this is all we need or all that we want to use.
                obj.DCMUpdateAll(gyro,acc,T);

                %obj.PositionUpdate(acc,T);
            otherwise
        end
```

```
            obj.currTime = obj.currTime + T;
            %display(sprintf('Current time %f',obj.currTime));
        end
    end

end
```

Code Listing C.5: Glove IMU

```
classdef GloveIMU < IMUCore
    %GloveIMU IMU Functions more specific to the GyroGlove.
    % These functions override or extend the lower level functions to
    % provide more Glove specific capabilities.

    properties

        % Save values of bias that are calculated during the init
        gyroBias
        accBias

        DCM_I_P = {}
        DCM_B_P = {}
        Velocity = {}
        Position = {}
        eAngles = {}

        currTime

        State  = GloveState.Idle;

        bGyroOnly = false
        bDCM_AccUpdate = true

    end

    methods

        function obj = GloveIMU()
            % Constructor for GloveIMU
            % The construture initialize all of the required data
            % structures

            % Initialize the IMUCore class.
            display('Constructing GloveIMU');
            obj = obj@IMUCore(6);

            obj.ResetGlove();

        end

        function delete(obj)
            display('Deleting GloveIMU');
        end
```

```

        function dcm = DCMBody2Inertial(obj,i)
            dcm = obj.DCM_I_P{i}*(obj.DCM_B_P{i})';
        end

        function dcm = DCMDig2Hand(obj,i,dcm_i_h)
            % DCM From a digit to the hand
            if nargin == 2
                dcm_i_h = obj.DCMBody2Inertial(1);
            end
            dcm_i_b = obj.DCMBody2Inertial(i);
            dcm = dcm_i_h' * dcm_i_b;
        end

        function eAngles = EulerAngles(obj,idx)
            % Caclculate and return a matrix of Euler angles, one row per
            % IMU and the columns are roll, pitch, yaw

            % I pass in the previous calculated values so that if we are
            % pitched up or down at about 90 degrees, we can use the
            % previous roll and yaw values.

            rad2deg = @(x) 360*(x/(2*pi));

            if nargin == 1
                eAngles = zeros(6,3);
                % Update the hand first
                DCM = obj.DCM_B_I{1};
                obj.eAngles{1} = obj.dcm2Euler(DCM,obj.eAngles{1});
                eAngles(1,:) = rad2deg(obj.eAngles{1});

                % Then update the fingers.
                for x = 2:6
                    % Convert DCM H to Inertial to DCM
                    DCM = (obj.DCM_B_I{1})'*obj.DCM_B_I{x};
                    %obj.eAngles{x} = obj.dcm2Euler(obj.DCM_B_I{x},obj.eAngles{x});
                    obj.eAngles{x} = obj.dcm2Euler(DCM,obj.eAngles{x});
                    eAngles(x,:) = rad2deg(obj.eAngles{x});
                end
            else
                obj.eAngles{idx} = obj.dcm2Euler(obj.DCM_B_I{idx},obj.eAngles{idx});
                eAngles = rad2deg(obj.eAngles{idx});
            end
        end

        function Pos = Positions(obj)
            % Convert the cell array into a matrix and then reshape it into
            % one row per IMU with columns x,y,z
```

```

            Pos = reshape(cell2mat(obj.Position),3,6)';
        end

        function V = Velocities(obj)
            % Convert the cell array into a matrix and then reshape it into
            % one row per IMU with columns x,y,z

            V = reshape(cell2mat(obj.Velocity),3,6)';
        end

        function DCMZeroP_B(obj)

        end

        function DCMUpdateAll(obj,gyro,T)
            % Take in new gyro data and accelerometer data and update the
            % DCM's with it.

            % function object to calculate the skew symetric matrix
            g = reshape(gyro,3,6)';
            for x = 1:6
                obj.DCM_I_P{x} = obj.DCMUpdate(obj.DCM_I_P{x},g(x,:),T);
            end
            obj.hKinematics.UpdateAngles(obj.EulerAngles());
        end

        function DCMUpdateFromAcc(obj,acc,idx)
            % Use the accelerometer to perform a course align of the DCM.
            % This technique works well if the IMU is stationarry and the
            % gravity vector is the only real acceleration in the system.

            acc = reshape(acc,3,6)';
            if nargin < 3
                for x = 1:6
                    obj.DCM_I_P{x} = obj.courseAlign(acc(x,:));
                end
            else
                obj.DCM_I_P{idx} = obj.courseAlign(acc(idx,:));
            end
            obj.hKinematics.UpdateAngles(obj.EulerAngles());
        end

        function InitializeDCMs(obj)
            % Use the current accelerometer and gyro history values in
            % order to initialize the DCM's using the course align
            % procedure. Use the current mean gyro value as the gyro
```

```
                bias
            % value. This neglects all other effects, such as Earth rate.

            % Zero out the gyro bias
            gyromean = mean(obj.gyroHistory,1);
            obj.gyroBias = gyromean;

            % Goal: take the accmean data and update each of the DCM's
            % based on the gravity vector.
            accmean = mean(obj.accHistory,1);
            at = reshape(accmean,3,6)';
            for x = 1:6
                C = obj.courseAlign(at(x,:));
            end
        end

        function PositionUpdate(obj,acc,T)
            % This is a very rudimentary implementation of the velocity and
            % position update based on current velocity/position and new
            % accelerometer values. The accelerometer values are oriented
            % to inertial frame and then used to update the components of V
            % and P in the I frame.
            a = reshape(acc,3,6)';
            for x = 1:6
                dcm = obj.DCM_B_I{x};
                aI = dcm*a(x,:)'; %rotate to I coordinates
                aI = aI+[0 0 1]'; % remove gravity vector.
                v = obj.Velocity{x}+aI*T; % add acceleration * T to velocity
                p = obj.Position{x} + v*T^2; % update position

                % Update current values.
                obj.Velocity{x} = v;
                obj.Position{x} = p;
            end
            obj.hKinematics.UpdatePos(obj.Positions);
        end

        function ResetGlove(obj,histSize)
            % Reset all of the internal parameters used for tracking the
            % glove position.

            % The default history size is 40, generally 1 second in my
            % examples, but this is programmable.
            if nargin == 1
                histSize = 40;
```

```matlab
            end

            obj.ResetHistory(histSize);

            obj.State = GloveState.InitialWait;
            obj.accBias = zeros(1,18);
            obj.gyroBias = zeros(1,18);
            obj.currTime = 0;
            for x=1:6
                obj.DCM_I_P{x} = eye(3);
                obj.DCM_B_P{x} = eye(3);
                obj.Velocity{x} = [0 0 0]';
                obj.Position{x} = [0 0 0]';
                obj.eAngles{x} = [0 0 0]';
            end
        end

        function Restart(obj)
            obj.currTime = 0;
            obj.State = GloveState.InitialWait;
        end

        function Update(obj,gyro,acc,T)
            % Update the set of IMU's with new gyro and accelerometer data.
            % The new data will be used to update the set of DCMs, as well
            % as the velocity and position values for each of the IMUs in
            % the system.

            % Note: The input format for the gyro and acc is one row per
            % IMU, and the 3 columns are the x,y, and z values.

            % Scale the accelerometer and gyro data. Subtract out any bias
            % values we have determined. For now the ACC Bias will be zero
            % for all settings.
            acc = acc-obj.accBias;

            gyro = gyro-obj.gyroBias;

            % I want to update the history in all state except the Idle
            % state. It makes it easier to make a seperate switch for this
            % operation rather than adding the UpdateHistory to all other
            % states of the main switch
            if ~(obj.State == GloveState.Idle)
                obj.UpdateHistory(gyro,acc);
            end
```

```

            switch obj.State
                case GloveState.Idle
                    obj.ResetGlove();
                    obj.State = GloveState.InitialWait;

                case GloveState.InitialWait

                    if obj.currTime > 1.5
                        if obj.isGloveStable(40)
                            display(sprintf('Glove Stable for init at %f',obj.currTime));
                            obj.InitializeDCMs();
                            obj.State = GloveState.InitialZero;
                        end
                    end
                case GloveState.InitialZero
                    % I am waiting for the glove to NOT be stable. This
                    % keeps everything in reset state until the glove moves
                    % the fist time. While the glove remains stable, I will
                    % continue to Init the DCM's so that they are in a
                    % current state when the glove starts to move.
                    if ~obj.isGloveStable(40)
                        obj.State = GloveState.IMURun;
                        display(sprintf('Going to Run state at %f',obj.currTime));
                    else
                        obj.InitializeDCMs();
                    end
                case GloveState.SecondZeroWait
                case GloveState.SecondZero
                case GloveState.IMURun

                    % Update the DCM from the gyro data. In some cases,
                    % this is all we need or all that we want to use.
                    %obj.DCMUpdateAll(gyro,T);


                    if obj.bDCM_AccUpdate
                        obj.DCMUpdateFromAcc(acc);
                    end

                    %obj.DCMUpdateFromAcc(acc);
                    %obj.PositionUpdate(acc,T);
                otherwise
            end
            obj.currTime = obj.currTime + T;
            %display(sprintf('Current time %f',obj.currTime));
        end
```

```

    end

end
```

Code Listing C.6: IMU Core

```
classdef IMUCore < handle
    % IMUCore Core functions for performing the IMU calcuations.
    %   These are the most generic functions. Classes that derive from this
    %   one can override these or add more.

    properties

        nIMUs

        % History for averaging purposes
        histSize
        gyroHistory
        accHistory

        rad2deg

        hKinematics
    end

    methods
        function obj = IMUCore(nIMUs)
            display('Constructing IMUCore');
            obj.nIMUs = nIMUs;
            obj.rad2deg = @(x) (x/(2*pi))*360;
            obj.hKinematics = [];
        end

        function delete(obj)
            display('Deleting IMUCore');
        end

        function dcm = DCMUpdate(obj,dcmin,gyro,T)
            ssomega = @(omega) [0 -omega(3) omega(2);omega(3) 0 -omega(1);-omega(2) omega(1) 0];
            gomega = ssomega(gyro);
            dDCM = dcmin*gomega; % This is rate of change of DCM
            dcm = dcmin + dDCM*T;
        end

        function dcm = DCMUpdateAB(obj,dcmin,gyroA,gyroB,T)
            ssomega = @(omega) [0 -omega(3) omega(2);omega(3) 0 -omega(1);-omega(2) omega(1) 0];
            gA = ssomega(gyroA);
            gB = ssomega(gyroB);
            dDCM = dcmin*gomega; % This is rate of change of DCM
```

```
            dcm = dcmin + dDCM*T;
        end

        function C = courseAlign(obj,accData)
            % This alrogithm taken directly from strapdown analytics by
            % Paul G. Savage. Many thanks to Mr. Savage for his kind
            % assistance.

            % ****** THIS MUST BE NORMALIZED ******
            % I was not normalizing this, so I added the /norm(accData),
            % which should take care of it.
            % Reference: Strapdown Analytics page 6-3, eq. 6.1.1-2
            c3 = -accData/norm(accData);
            c2 = zeros(1,3);

            % If the X axis is near vertical, we need to use a slightly
            % different initialization technique, otherwise we will have a
            % null in the denominator of the equations.
            if abs(c3(1)) < 0.85
                c2(2) = c3(3)/sqrt(c3(2)^2+c3(3)^2);
                c2(3) = -c3(2)/sqrt(c3(2)^2+c3(3)^2);
            else
                c2(1) = c3(2)/sqrt(c3(1)^2+c3(2)^2);
                c2(2) = -c3(1)/sqrt(c3(1)^2+c3(2)^2);
            end

            c1 = cross(c2,c3);
            %c1 = zeros(1,3);
            %c1(1) = c2(2)*c3(3)-c2(3)*c3(2);
            %c1(2) = c2(3)*c3(1)-c2(1)*c3(3);
            %c1(3) = c2(1)*c3(2)-c2(2)*c3(1);

            C = [c1;c2;c3];
        end

        function euler = dcm2Euler(obj,dcm,oldAngles)
            % Implementation of the dcm2Euler algorithm in the Strapdown
            % analytics book

            % This uses the values from the AeroBlockset. I'll disable that
            % for now.... I think I've got the other figured out.
            %[y,p,r] = dcm2angle(dcm);
            %euler = [r,-p,y];
            %return

            pitch = atan(-dcm(3,1)/sqrt(dcm(3,2)^2+dcm(3,3)^2));
            %pitch = -asin(dcm(3,1));
            %pitch = atan2(-dcm(3,1),sqrt(dcm(3,2)^2+dcm(3,3)^2));
```

```

            if abs(dcm(3,1)) < 0.98
                roll = atan2(dcm(3,2),dcm(3,3));
                yaw = atan2(dcm(2,1),dcm(1,1));
            else
                roll = oldAngles(1);
                yaw = oldAngles(3);
            end
            euler = [roll,pitch,yaw];
        end

        function dcm = AlignUV(obj,U,V)
            % implement the algorighm to calculate the DCM that maps thumb
            % and finger coordinates to hand coordinates. There will be 5
            % such matrices, all referenced to the hand matrix.
            % Input parameters are U a V, where U(1,:) and V(1,:) are two
            % vectors taken from both frames of reference, and U(2,:) and
            % V(2,:) are two distinct vectors.

            % Initialize W to have two columns
            W = zeros(3,2);

            % Generate W with the cross product
            for y = 1:2
                W(:,y) = cross(U(:,y),V(:,y));
            end

            % build the DA1 and DA2 matrices
            DA = zeros(3,3,2);
            for d = 1:2
                DA(:,:,d) = [U(:,d) V(:,d) W(:,d)];
            end

            % And, the computed DCM is the DA1 * DA2^-1
            dcm = DA(:,:,1)*DA(:,:,2)^-1;
        end

        function [v,p] = PositionUpdate(obj,vin,pin,dcmin,acc,T)
            % position update based on current velocity/position and new
            % accelerometer values. The accelerometer values are oriented
            % to inertial frame and then used to update the components of V
            % and P in the I frame.
            [r,c] = size(a);
            if r == 1
                a = a';
            end
            aI = dcmin*a; %rotate to I coordinates
```

```
            aI = aI+[0 0 1]'; % remove gravity vector.
            v = vin+aI*T; % add acceleration * T to velocity
            p = pin + v*T^2; % update position
        end

        function bool = isGloveStable(obj,n)
            % Calculate if the glove has been stable over the last N
            % samples

            % Get the maximum gyro deviation of any gyro over the history.
            gmax = max(max(obj.gyroHistory(1:n,:)));

            % Don't bother with all of the calclations, just see if the
            % acceleration has been steady. If the glove is moving with
            % constant motion, we would consider that to be okay. One thing
            % this does not account for is noise, so the stability level
            % must be higher than the noise level.
            maxDev = max(max(obj.accHistory(1:n,:))-min(obj.accHistory(1:n,:)));

            if maxDev > 0.05 || gmax > 0.1
                bool = false;
            else
                bool = true;
            end
            %display(sprintf('T:%f Gmax:%f maxDev:%f', obj.currTime, gmax, maxDev));
            return
        end

        function ResetHistory(obj,nHistSize)
            obj.histSize = nHistSize;
            obj.gyroHistory = zeros(nHistSize,3*obj.nIMUs);
            obj.accHistory = zeros(nHistSize,3*obj.nIMUs);
        end

        function UpdateHistory(obj,gyro,acc)
            % Update the history values used for averaging, stability
            % detection and initialization.
            % These history values have bias removed, which makes them
            % different from the history values in the GloveData class. I
            % was thinking of using the GloveData class, but this makes
            % that more difficult, and makes it more appropriate to keep
            % the history data in this class.

```

```
            obj.gyroHistory = circshift(obj.gyroHistory,[1 0]);
            obj.accHistory = circshift(obj.accHistory,[1 0]);

            obj.gyroHistory(1,:) = gyro;
            obj.accHistory(1,:) = acc;
        end

        function SetHK(obj,hk)
            obj.hKinematics = hk;
        end

    end

end
```

Code Listing C.7: Hand Kinematics

```
classdef HandKinematics < handle
    %HandKinematics Manage kinematics of hand, fingers and thumb
    %   This class managers the position kinimatics of the hand, including
    %   the fingers and the thumb. The IMU object can call this with
    %   updated values to determine if those values are valid. How exactly
    %   this gets incorporated into the IMU is unclear at the timer of this
    %   writing, but some type of Kalman or particle filter, etc. would
    %   probably be the ideal solution.

    properties
        eAngles
        Positions
    end

    methods
        function obj = HandKinematics()
        end

        function UpdatePanda(obj)
            obj.UpdateGlove();
        end

        function UpdateAngles(obj,eAngles)
            obj.eAngles = eAngles;
        end

        function UpdatePos(obj,newPos)
            obj.Positions = newPos;
        end

        function UpdateGlove(obj)
```

```
            % Update the Panda3D visualization with current hand, finger
            % and thumb positions and orientations.

            %eAngles = obj.imuObj.EulerAngles();
            %Pos = obj.imuObj.Positions();
            if size(obj.eAngles,1) > 0
                eAngles = obj.eAngles;
                Pos = obj.Positions;
                Pos = [0 0 1;zeros(5,3)];
                Row2Glove = [0 3 2 5 1 4];

                % Upate the hand using all 3 gyro values.
                rpq = eAngles(1,:);
                GyroGloveClient(0,[0 0 0 rpq],0);

                Finger2Idx = fliplr([5 3 2 6]);
                for x = 1:4
                    idx = Finger2Idx(x);
                    rpq = eAngles(idx,:);
                    pos = [2.5 0 0];
                    FingerAngle(x,pos',rpq(2));
                    %GyroGloveClient(Row2Glove(x),[pos 0 rpq(2) 0],0);
                end

                % Do the Thumb
                rpq = eAngles(4,:);
                pos = [3 1 -0.5];
                obj.ThumbAngle(pos',rpq(2));
            end

            % Ignore the thumb for now..

        end

        function singular = FingerAngle(obj,fidx,pos, angle)
        %FingerAngle Translate position in X and Z axis based on angle.
        %   Calculate the rotation of the finger and translation of the position
        %   value. I am limiting the range of angle from +20 to -90. Fingers can
        %   move much more than that, but I am making this simple approximation for
        %   now.

            if angle > 20 || angle < -90
                singular = true;
                return;
            end
            singular = false;

            rangle = (angle/360)*2*pi;
```

```
            C = [cos(rangle) 0 sin(rangle)
                0 1 0
                sin(rangle) 0 cos(rangle)];

            try
                newpos = C*pos;

                GyroGloveClient(fidx,[newpos' 0 angle 0],0);
            catch e
                display('Error when doing translation.');
            end
        end

        function singular = ThumbAngle(obj,pos,angle)
        %FingerAngle Translate position in X and Z axis based on angle.
        %   Calculate the rotation of the finger and translation of the position
        %   value. I am limiting the range of angle from +20 to -90. Fingers can
        %   move much more than that, but I am making this simple approximation for
        %   now.

            if angle > 30 || angle < -90
                singular = true;
                return;
            end
            singular = false;

            rangle = (angle/360)*2*pi;
            C = [cos(rangle) 0 sin(rangle)
                0 1 0
                sin(rangle) 0 cos(rangle)];

            try
                newpos = C*pos;

                GyroGloveClient(5,[newpos' 0 angle 0],0);
            catch e
                display('Error when doing translation.');
            end
        end

    end

end
```

APPENDIX D

MATLAB™ MEX CODE

MATLAB™ MEX CODE

Code Listing D.1: GyroGlove Main

```
#include <fcntl.h>
#include <sys/ioctl.h>
#include <paths.h>
#include <sysexits.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>

#include <CoreFoundation/CoreFoundation.h>

#include <IOKit/IOKitLib.h>
#include <IOKit/serial/IOSerialKeys.h>
#include <IOKit/IOBSD.h>

#include <mex.h>
#include <math.h>


void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{

    int   SSEnable = 0;

    if (nrhs < 1) {
        mexErrMsgTxt("One input required.");
    } else if (nlhs != 1) {
        mexErrMsgTxt("One output argument required.");
    }

    // I am expecting an nxm array, where n is the # of clusters
    // and m is the number of features.
    mwSize nrows = mxGetM(prhs[0]);
    mwSize ncols = mxGetN(prhs[0]);
    mwSize elements = mxGetNumberOfElements(prhs[0]);
    mwSize number_of_dims=mxGetNumberOfDimensions(prhs[0]);

    if (!mxIsDouble(prhs[0])) {
        mexErrMsgTxt("Input must be a double array.");
    }

    if (nrhs > 1) {
        SSEnable = 1;
    }

    plhs[0] = mxCreateDoubleMatrix(1,3,mxREAL);

    double *pOut = mxGetPr(plhs[0]);
```

```
    pOut[0] = 12.2;
    pOut[1] = 1.12;
    pOut[2] = 12;

    double *pFa = mxGetPr(prhs[0]);

    double *pCol[10];
    for (int z=0;z<ncols;z++) {
        pCol[z] = pFa+z*nrows;
    }

    double dMin = 1e12;

    // Data is in column major order... so,
    // x and y can point to each column, then
    // iterate on features by adding x/y + mcols
    double fsum;
    for (int x=0;x<nrows;x++) {
        for (int y=0;y<nrows;y++) {
            if (y != x) {
                fsum = 0;
                for (int z=0;z<ncols;z++) {
                    // Simple squared function.
                    double t = (pCol[z][x]-pCol[z][y]);
                    fsum += t*t;
                    //fsum += (pFa[x+z*nrows]-pFa[y+z*nrows])*(pFa[x+
                        z*nrows]-pFa[y+z*nrows]);
                }
                //fsum = sqrt(fsum); // Don't bother with the sqrt..
                    we just want t relative value..
                if (fsum < dMin) {
                    dMin = fsum;
                    pOut[0] = double(x+1);
                    pOut[1] = double(y+1);
                    pOut[2] = dMin;

                    // This is an optimization. If a lot of pixels
                        are close to zero in distance,
                    // then it is not that important to find the "
                        closest" of those.
                    if (dMin < 0.02) {
                        return;
                    }
                }
            }
        }
    }
}
```

Code Listing D.2: GyroGlove Client

```
#include <fcntl.h>
```

```
#include <sys/ioctl.h>
#include <paths.h>
#include <sysexits.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>

#include <CoreFoundation/CoreFoundation.h>

#include <IOKit/IOKitLib.h>
#include <IOKit/serial/IOSerialKeys.h>
#include <IOKit/IOBSD.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>

#include <mex.h>
#include <math.h>

static int sock_client = 0;
static sockaddr_in sa;
static char buffer[100];

void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{

    int   SSEnable = 0;

    if (nrhs == 0) {
        if (sock_client) {
            close(sock_client);
            sock_client = 0;
            mexPrintf("Closed Socket connection.\n");
        }

        return;
    }

    if (nrhs < 2) {
        mexErrMsgTxt("Two arguments required. The index,"
            " and an array of position and rotation.");
    }

    double idx = mxGetScalar(prhs[0]);

    // I am expecting an nxm array, where n is the # of clusters
    // and m is the number of features.
    mwSize nrows = mxGetM(prhs[1]);
    mwSize ncols = mxGetN(prhs[1]);
```

```
    mwSize elements = mxGetNumberOfElements(prhs[1]);
    mwSize number_of_dims=mxGetNumberOfDimensions(prhs[1]);

    if (ncols != 6) {
        mexErrMsgTxt("Input array in 2nd argument must have 6 columns
            ");
    }

    unsigned long sleeptime = 10000;
    if (nrhs > 2) {
        sleeptime = int(mxGetScalar(prhs[2]));
    }

    double* pin = mxGetPr(prhs[1]);

    // Initialize the socket if this is the first time.
    if (sock_client == 0 ) {
        sock_client = socket(AF_INET, SOCK_DGRAM,0);
        if (sock_client == 0) {
            mexErrMsgTxt("Failed to open socket!");
        }
        mexPrintf("Opened Socket connection.\n");

        sa.sin_family = AF_INET;
        sa.sin_port = htons(5432);

        inet_pton(AF_INET, "127.0.0.1",(void*)&sa.sin_addr.s_addr);
    }

    double* pStart = pin;

    for (int r = 0;r< nrows;r++) {
        sprintf(buffer,"%d,%f,%f,%f,%f,%f,%f",
            int(idx),
            pStart[r],
            pStart[r+1*nrows],
            pStart[r+2*nrows],
            pStart[r+3*nrows],
            pStart[r+4*nrows],
            pStart[r+5*nrows]
            );

        sendto(sock_client, &buffer[0], strlen(buffer), 0,
            (const sockaddr*)&sa, sizeof(struct sockaddr_in)
            );

        usleep(sleeptime);
    }
}
```

APPENDIX E

FIRMWARE CODE DOXYGEN OUTPUT

GyroAccGlove

1.0

Generated by Doxygen 1.7.3

Sat Sep 10 2011 17:54:13

Contents

1 Main Page **2**

2 Class Index **3**
2.1 Class Hierarchy 3

3 Class Index **3**
3.1 Class List 3

4 File Index **4**
4.1 File List 4

5 Class Documentation **5**
5.1 CmdProcessor Class Reference 5
5.1.1 Detailed Description 7
5.1.2 Constructor & Destructor Documentation 7
5.1.3 Member Function Documentation 8
5.1.4 Member Data Documentation 13
5.2 Fifo Class Reference 15
5.2.1 Detailed Description 15
5.2.2 Member Typedef Documentation 16
5.2.3 Constructor & Destructor Documentation 16
5.2.4 Member Function Documentation 16
5.2.5 Member Data Documentation 19
5.3 HardwareSerial Class Reference 19
5.3.1 Detailed Description 20
5.3.2 Constructor & Destructor Documentation 21
5.3.3 Member Function Documentation 21
5.3.4 Member Data Documentation 25
5.4 I2C_Master Class Reference 27
5.4.1 Detailed Description 29
5.4.2 Member Typedef Documentation 29
5.4.3 Member Enumeration Documentation 29
5.4.4 Constructor & Destructor Documentation 31
5.4.5 Member Function Documentation 31
5.4.6 Member Data Documentation 35
5.5 I2CNotify Class Reference 37
5.5.1 Detailed Description 38
5.5.2 Member Function Documentation 38
5.6 IMU Class Reference 39
5.6.1 Detailed Description 42
5.6.2 Member Typedef Documentation 42
5.6.3 Member Enumeration Documentation 42
5.6.4 Constructor & Destructor Documentation 44
5.6.5 Member Function Documentation 46
5.6.6 Member Data Documentation 67
5.7 IMUBase Class Reference 71
5.7.1 Detailed Description 72
5.7.2 Member Function Documentation 72

5.8 Port Class Reference 74
5.8.1 Detailed Description 75
5.8.2 Constructor & Destructor Documentation 75
5.8.3 Member Function Documentation 76
5.8.4 Member Data Documentation 79
5.9 PortNotify Class Reference 80
5.9.1 Detailed Description 80
5.9.2 Member Function Documentation 80
5.10 Print Class Reference 81
5.10.1 Detailed Description 81
5.10.2 Member Function Documentation 82
5.11 IMU::regWrite Struct Reference 88
5.11.1 Detailed Description 88
5.11.2 Member Data Documentation 88
5.12 ring_buffer Struct Reference 89
5.12.1 Detailed Description 89
5.12.2 Member Data Documentation 89
5.13 TimerCntr Class Reference 90
5.13.1 Detailed Description 90
5.13.2 Constructor & Destructor Documentation 91
5.13.3 Member Function Documentation 92
5.13.4 Member Data Documentation 96
5.14 TimerNotify Class Reference 97
5.14.1 Detailed Description 97
5.14.2 Member Function Documentation 98

6 File Documentation 98
6.1 clksystem.cpp File Reference 98
6.1.1 Define Documentation 99
6.1.2 Function Documentation 99
6.2 clksystem.cpp 101
6.3 clksystem.h File Reference 103
6.3.1 Function Documentation 103
6.4 clksystem.h 104
6.5 CmdProcessor.cpp File Reference 104
6.6 CmdProcessor.cpp 104
6.7 CmdProcessor.h File Reference 107
6.8 CmdProcessor.h 107
6.9 cpp_hacks.cpp File Reference 108
6.9.1 Function Documentation 108
6.10 cpp_hacks.cpp 109
6.11 cpp_hacks.h File Reference 109
6.11.1 Function Documentation 109
6.12 cpp_hacks.h 110
6.13 Documentation.html File Reference 110
6.14 Documentation.html 110
6.15 fifo.cpp File Reference 110
6.15.1 Function Documentation 111
6.16 fifo.cpp 111
6.17 fifo.h File Reference 113

6.17.1 Function Documentation 113
6.18 fifo.h 114
6.19 GyroAcc.cpp File Reference 115
6.19.1 Function Documentation 116
6.19.2 Variable Documentation 118
6.20 GyroAcc.cpp 119
6.21 HardwareSerial.cpp File Reference 121
6.21.1 Define Documentation 122
6.21.2 Function Documentation 122
6.22 HardwareSerial.cpp 124
6.23 HardwareSerial.h File Reference 128
6.24 HardwareSerial.h 129
6.25 I2C_Master.h File Reference 129
6.26 I2C_Master.h 130
6.27 IMU.cpp File Reference 132
6.27.1 Variable Documentation 133
6.28 IMU.cpp 133
6.29 IMU.h File Reference 145
6.30 IMU.h 146
6.31 IMUManager.cpp File Reference 149
6.31.1 Define Documentation 150
6.31.2 Variable Documentation 151
6.32 IMUManager.cpp 151
6.33 NewDel.cpp File Reference 159
6.33.1 Function Documentation 159
6.34 NewDel.cpp 160
6.35 NewDel.h File Reference 160
6.35.1 Function Documentation 160
6.36 NewDel.h 161
6.37 Port.cpp File Reference 161
6.37.1 Define Documentation 162
6.37.2 Function Documentation 162
6.37.3 Variable Documentation 163
6.38 Port.cpp 163
6.39 Port.h File Reference 166
6.40 Port.h 166
6.41 Print.cpp File Reference 167
6.42 Print.cpp 167
6.43 Print.h File Reference 170
6.43.1 Define Documentation 171
6.44 Print.h 172
6.45 TimerCntr.cpp File Reference 173
6.45.1 Define Documentation 173
6.45.2 Function Documentation 174
6.46 TimerCntr.cpp 175
6.47 TimerCntr.h File Reference 180
6.48 TimerCntr.h 180

1 Main Page

Browning Research Field Emitter Control and Measurement System.

Introduction This code is written in C++. The AVR tools support a limited set of C++ capabilities so there are no fancy constructs such as templates. C++ allows the high level features to be encapsulated into a class and used where needed. In most cases these classes are built around hardware resources. There is a class to work with IO Ports, one for HardwareSerial, etc.

Compiling The compiler and debug environment for the AVR tools is freely available. Several options exist, the simplest on is the AVR Studio. This tool can be downloaded from Atmel's web site. The tool runs on a Windows PC only.

For Unix or Macs there are freely available GNU toolchains. These do not include a GUI, but command line builds work just find.

Controller Board Hardwarew The hardware consists of the following comonents:

- Controller Board.
- Emitter Control Board
- Current Monitor Board

Controller Board Board for controlling all other components and interfacing to host computer.

Emitter Control Board Contains N-Channel FETS to control the current into the emitter elements.

Microprocessor The procssor on the board ia an Atmel ATxmega 128A1.

Some important links for this device are:

- `Product Datasheet`
- `Product Manual`
- `Product Website`

The product manual is very similar to the datasheet, however the manual contains register definitions. These are very important when configuring the hardware resources available within the ATxmega.

2 Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

CmdProcessor 5

Fifo 15

I2C_Master 27

I2CNotify 37

- **IMU** 39

IMUBase 71

- **IMU** 39

Port 74

PortNotify 80

Print 81

- **HardwareSerial** 19

IMU::regWrite 88

ring_buffer 89

TimerCntr 90

TimerNotify 97

- **IMU** 39

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

CmdProcessor 5

Fifo (Fifo Class for unsigned 8 bit values) 15

HardwareSerial (HardwareSerial implementation) 19

I2C_Master 27
I2CNotify 37
IMU 39
IMUBase 71
Port 74
PortNotify 80
Print 81
IMU::regWrite 88
ring_buffer 89
TimerCntr 90
TimerNotify 97

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

clksystem.cpp 98
clksystem.h 103
CmdProcessor.cpp 104
CmdProcessor.h 107
cpp_hacks.cpp 108
cpp_hacks.h 109
Documentation.html 110
fifo.cpp 110
fifo.h 113
GyroAcc.cpp 115
HardwareSerial.cpp 121
HardwareSerial.h 128

I2C_Master.h **129**

IMU.cpp **132**

IMU.h **145**

IMUManager.cpp **149**

NewDel.cpp **159**

NewDel.h **160**

Port.cpp **161**

Port.h **166**

Print.cpp **167**

Print.h **170**

TimerCntr.cpp **173**

TimerCntr.h **180**

5 Class Documentation

5.1 CmdProcessor Class Reference

`#include <CmdProcessor.h>`

Public Member Functions

- CmdProcessor (HardwareSerial *pHW)

  Number of valid parameters.

- ~CmdProcessor ()

  Destructor. Release memory allocated in constructor.

- bool checkCommands ()
- char * cmdTerm ()

  Return pointer to termination string.

- void cmdTerm (char *)
- void resetCmd ()

  Clear the command status values so a new command can be started.

- const char * cmdDelim ()

  Return current delimiter string.

- void cmdDelim (const char ∗)
- const char ∗ getCmd ()

  Return the command string.

- uint8_t paramCnt ()

  Return the number of parameters parsed from the current command.

Parameter Extraction Functions

getParam is overloaded on the variable type, this means that each possible type has a unique function. The type of the parameter you are seeking will determine the exact function that is called, which will then do the right thing to convert the string paramter value to an unsigned int, double etc.

- void getParam (uint8_t idx, uint8_t &p)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, uint16_t &p)

  Parse the index parameter into a unsigned 16 bit integer.

- void getParam (uint8_t idx, long &l)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, int &p)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, double &f)

  Parse the index parameter into a double.

- void getParam (uint8_t idx, char ∗&p, uint8_t maxlen=128)

  Parse the index parameter into a string with the length specified.

Protected Member Functions

- void processCmd ()

Protected Attributes

- HardwareSerial ∗ _pHW
- char ∗ _pTokens [10]

  Store the serial object.

- char ∗ _pCmd

  List of command tokens.

- char ∗ _pCmdString

Command buffer.

- uint8_t _cmdPos

  Current command.

- bool _validCmd

  Current position during serial read.

- char * _pCmdTerm

  Indicates a current valid command.

- char * _pCmdDelim

  Store command terminator.

- uint8_t _paramCnt

  Current command parameter delimiter.

5.1.1 Detailed Description

Definition at line 7 of file CmdProcessor.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 CmdProcessor::CmdProcessor (HardwareSerial * *pHW*)

Number of valid parameters.

Construct a new CmdProcessor. Pass in reference to the HardwareSerial class to use for command processing. Store the serial pointer and then initialize the internal data strings used during command input processing and output processing.

Definition at line 11 of file CmdProcessor.cpp.

References _pCmd, _pCmdDelim, _pCmdString, _pCmdTerm, _pHW, and resetCmd().

```
{
    _pHW = pHW;

    _pCmdString = (char*)malloc(128);
    _pCmd = 0;
    _pCmdTerm = (char*)malloc(3);
    strcpy(_pCmdTerm,"\n\r");
    _pCmdDelim = (char*)malloc(3);
    strcpy(_pCmdDelim," \t");
    resetCmd();
}
```

5.1.2.2 CmdProcessor::∼CmdProcessor ( )

Destructor. Release memory allocated in constructor.

Definition at line 25 of file CmdProcessor.cpp.

References _pCmdDelim, _pCmdString, _pCmdTerm, _pHW, and HardwareSerial::end().

```
{
    if (_pHW) {
        _pHW->end();
    }
    free(_pCmdString);
    free(_pCmdDelim);
    free(_pCmdTerm);
}
```

5.1.3 Member Function Documentation

5.1.3.1 bool CmdProcessor::checkCommands ( )

Read new characters from the serial port

Read any new characters into the command buffer. Look for the command terminator. If the terminator is found, store the command, process the command buffer and return 1 to indicate that a new command is availble. If a full command is not yet present, then return zero.

Definition at line 68 of file CmdProcessor.cpp.

References _cmdPos, _pCmdString, _pCmdTerm, _pHW, HardwareSerial::available(), Print::print(), processCmd(), and HardwareSerial::read().

```
{
    while (_pHW->available() > 0) {
        unsigned char c = _pHW->read();
        if (strchr(_pCmdTerm,c) != 0) {
            if (_cmdPos > 0) {
                // Done with this command.
                _pCmdString[_cmdPos] = 0; // Null terminate command
                processCmd();
                return 1;
            } else {
                _pHW->print("Ok\n");
            }
        } else {
            _pCmdString[_cmdPos++] = c;
        }
    }
    return 0;
}
```

5.1.3.2 const char ∗ CmdProcessor::cmdDelim ( )

Return current delimiter string.

Definition at line 48 of file CmdProcessor.cpp.

References _pCmdDelim.

```
{
    return _pCmdDelim;
}
```

5.1.3.3 void CmdProcessor::cmdDelim (const char ∗ *d*)

Set new delimiter string. Free memory, allocate new memory and copy new value.

Definition at line 55 of file CmdProcessor.cpp.

References _pCmdDelim.

```
{
    free(_pCmdDelim);
    _pCmdDelim = (char*)malloc(strlen(d) + 1);
    strcpy(_pCmdDelim,d);
}
```

5.1.3.4 char ∗ CmdProcessor::cmdTerm ( )

Return pointer to termination string.

Definition at line 36 of file CmdProcessor.cpp.

References _pCmdTerm.

```
{ return _pCmdTerm; }
```

5.1.3.5 void CmdProcessor::cmdTerm (char ∗ *t*)

Set a new command terminator. Free memory for previous value, allocate new memory and save the new value.

Definition at line 40 of file CmdProcessor.cpp.

References _pCmdTerm.

```
{
    free(_pCmdTerm);
    _pCmdTerm = (char*)malloc(strlen(t) + 1);
    strcpy(_pCmdTerm,t);
}
```

5.1.3.6 const char * CmdProcessor::getCmd ( )

Return the command string.

Definition at line 123 of file CmdProcessor.cpp.

References _pCmd.

```
{
    return _pCmd;
}
```

5.1.3.7 void CmdProcessor::getParam (uint8_t *idx*, double & *f*)

Parse the index parameter into a double.

Definition at line 176 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        uint8_t nScans;
        nScans = sscanf(_pTokens[idx],"%lf", &p);
        //p = atof(_pTokens[idx]);
    }
}
```

5.1.3.8 void CmdProcessor::getParam (uint8_t *idx*, uint8_t & *p*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 154 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.9 void CmdProcessor::getParam (uint8_t *idx*, uint16_t & *p*)

Parse the index parameter into a unsigned 16 bit integer.

Definition at line 146 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.10 void CmdProcessor::getParam (uint8_t *idx*, int & *p*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 161 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.11 void CmdProcessor::getParam (uint8_t *idx*, char ∗& *p*, uint8_t *maxlen* = 128)

Parse the index parameter into a string with the length specified.

Definition at line 186 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        strncpy(p,_pTokens[idx],maxlen);
    }
}
```

5.1.3.12 void CmdProcessor::getParam (uint8_t *idx*, long & *l*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 168 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        l = atol(_pTokens[idx]);
    }
}
```

5.1.3.13 uint8_t CmdProcessor::paramCnt ( )

Return the number of parameters parsed from the current command.

Definition at line 129 of file CmdProcessor.cpp.

References _paramCnt.

```
{
    return _paramCnt;
}
```

5.1.3.14 void CmdProcessor::processCmd ( ) [protected]

Process the commands in the command buffer Split the command into parameters based on the command delimiter. The maximum number of command tokens is 10.

Definition at line 92 of file CmdProcessor.cpp.

References _paramCnt, _pCmd, _pCmdDelim, _pCmdString, _pTokens, and _validCmd.

Referenced by checkCommands().

```
{
    // See if the command delimiter exists in the
    // command. if it does not, then the command
    // is the entire string.
    if (strpbrk(_pCmdString,_pCmdDelim)) {
        _pCmd = strtok(_pCmdString,_pCmdDelim);
        char* pTok = strtok(0,_pCmdDelim);
        int i = 0;
        while (i < 10 && pTok) {
            _pTokens[i++] = pTok;
            pTok = strtok(0,_pCmdDelim);
        }
        _paramCnt = i;
        _validCmd = true;
    } else {
        _pCmd = _pCmdString;
        _paramCnt = 0;
        _validCmd = true;
    }
}
```

5.1.3.15 void CmdProcessor::resetCmd ( )

Clear the command status values so a new command can be started.

Definition at line 115 of file CmdProcessor.cpp.

References _cmdPos, _paramCnt, and _validCmd.

Referenced by CmdProcessor().

```
{
    _cmdPos = 0;
    _validCmd = false;
    _paramCnt = 0;
}
```

5.1.4 Member Data Documentation

5.1.4.1 uint8_t CmdProcessor::_cmdPos [protected]

Current command.

Definition at line 14 of file CmdProcessor.h.

Referenced by checkCommands(), and resetCmd().

5.1.4.2 uint8_t CmdProcessor::_paramCnt [protected]

Current command parameter delimiter.

Definition at line 18 of file CmdProcessor.h.

Referenced by getParam(), paramCnt(), processCmd(), and resetCmd().

5.1.4.3 char* CmdProcessor::_pCmd [protected]

List of command tokens.

Definition at line 12 of file CmdProcessor.h.

Referenced by CmdProcessor(), getCmd(), and processCmd().

5.1.4.4 char* CmdProcessor::_pCmdDelim [protected]

Store command terminator.

Definition at line 17 of file CmdProcessor.h.

Referenced by cmdDelim(), CmdProcessor(), processCmd(), and ~CmdProcessor().

5.1.4.5 char* CmdProcessor::_pCmdString [protected]

Command buffer.

Definition at line 13 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), processCmd(), and ~CmdProcessor().

5.1.4.6 char* CmdProcessor::_pCmdTerm [protected]

Indicates a current valid command.

Definition at line 16 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), cmdTerm(), and ~CmdProcessor().

5.1.4.7 HardwareSerial* CmdProcessor::_pHW [protected]

Definition at line 10 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), and ~CmdProcessor().

5.1.4.8 char* CmdProcessor::_pTokens[10] [protected]

Store the serial object.

Definition at line 11 of file CmdProcessor.h.

Referenced by getParam(), and processCmd().

5.1.4.9 bool CmdProcessor::_validCmd [protected]

Current position during serial read.

Definition at line 15 of file CmdProcessor.h.

Referenced by processCmd(), and resetCmd().

The documentation for this class was generated from the following files:

- CmdProcessor.h

- CmdProcessor.cpp

5.2 Fifo Class Reference

Fifo Class for unsigned 8 bit values.

`#include <fifo.h>`

Public Types

- typedef uint8_t FifoType

Public Member Functions

- Fifo (uint8_t size)
- int8_t push (FifoType ∗)
- int8_t pop (FifoType ∗pData)
- uint8_t count ()
- bool full ()

  Return true if the fifo is full.
- bool empty ()

  Return true if the fifo is empty.
- void clear ()

  Clear the fifo by resetting the start and end pointer.

Private Attributes

- FifoType ∗ _pdata
- FifoType ∗ _start
- FifoType ∗ _end
- uint8_t _size

5.2.1 Detailed Description

Fifo Class for unsigned 8 bit values. Construct a fifo and specify the number of elements to store. The fifo constructor will allocate memory for the specified number of values. The Fifo class contains member functions for pusshing, popping and checking the status of the fifo.

Definition at line 16 of file fifo.h.

5.2.2 Member Typedef Documentation

5.2.2.1 typedef uint8_t Fifo::FifoType

Definition at line 20 of file fifo.h.

5.2.3 Constructor & Destructor Documentation

5.2.3.1 Fifo::Fifo (uint8_t *size*)

Construct the fifo object. Allocate memory for the specified number of elements and set the internal value to indicate the size of the fifo. Reset the start and end data points to their clear state. The clear function is called to maintain consitency and insure that clear() always does the right thing.

Definition at line 14 of file fifo.cpp.

References _pdata, _size, and clear().

```
{
    _size = size;
    _pdata = (FifoType*)malloc(_size * sizeof(FifoType));
    clear();
}
```

5.2.4 Member Function Documentation

5.2.4.1 void Fifo::clear ( )

Clear the fifo by resetting the start and end pointer.

Definition at line 22 of file fifo.cpp.

References _end, _pdata, and _start.

Referenced by Fifo(), and FifoTest().

```
{
    _start = _end = _pdata;
}
```

5.2.4.2 uint8_t Fifo::count ( )

Return the number of elements currently in the fifo if the end and start pointers are the same then the fifo is empty and count == 0. If they differ, then we need to check for wrap-around in order to properly determine the size. In the following examples a = marks empty spots, while an x marks filled spots.

```
              s                            e
       ========xxxxxxxxxxxxxxxxxxxxxxxxxxxx=========================
```

In this case end > start, so count is equal to the distance between them or $end - start$.

```
              e                            s
       xxxxxxxxx==========================xxxxxxxxxxxxxxxxxxxxxxxxxxx
```

In this case end < start, so data wraps around. The total count is equal to the size of the buffer, minus the number of blank spots, or $size - (start - end)$.

The total number of possible elements that can be stored is size -1, so

Definition at line 50 of file fifo.cpp.

References _end, _size, and _start.

Referenced by FifoTest().

```
{
    if (_end == _start) return 0;
    if (_end > _start) {
        return _end - _start;
    }
    return _size - (_start - _end);
}
```

5.2.4.3 bool Fifo::empty ( )

Return true if the fifo is empty.

Definition at line 66 of file fifo.cpp.

References _end, and _start.

Referenced by pop().

```
{
    return (_start == _end);
}
```

5.2.4.4 bool Fifo::full ( )

Return true if the fifo is full.

Definition at line 60 of file fifo.cpp.

References _end, _size, and _start.

Referenced by push().

```
{
    return (_start - _end)  == 1 || (_end - _start) == _size;
}
```

5.2.4.5 int8_t Fifo::pop (FifoType ∗ *pD*)

Remove the top value from the Fifo. We do not have exceptions in this simple C++ implementation, so this function is not able to do anything to indicate that the called tried to pop a value from an empty fifo. In that case, a zero value is returned, which is not unique so the caller will have to insure that pop is never called on an empty fifo.

Definition at line 92 of file fifo.cpp.

References _pdata, _size, _start, and empty().

Referenced by FifoTest().

```
{
    if (empty()) {
        return -1; // Nothing else to do
    }
    *pD = *(_start++);
    if ((_start - _pdata) > _size) {
        _start = _pdata;
    }
    return 0;
}
```

5.2.4.6 int8_t Fifo::push (FifoType ∗ *d*)

Push a new value onto the fifo. This function returns 0 if the operation succeeds, and a negative value if the operation fails.

Definition at line 74 of file fifo.cpp.

References _end, _pdata, _size, and full().

Referenced by FifoTest().

```
{
    if (full()) return -1;
    *(_end++) = *d;

    // Wrap the end back to the beginning.
    if ((_end - _pdata) > _size) {
        _end = _pdata;
    }

        return 0;
}
```

5.2.5 Member Data Documentation

5.2.5.1 FifoType* Fifo::_end [private]

Definition at line 25 of file fifo.h.

Referenced by clear(), count(), empty(), full(), and push().

5.2.5.2 FifoType* Fifo::_pdata [private]

Definition at line 23 of file fifo.h.

Referenced by clear(), Fifo(), pop(), and push().

5.2.5.3 uint8_t Fifo::_size [private]

Definition at line 26 of file fifo.h.

Referenced by count(), Fifo(), full(), pop(), and push().

5.2.5.4 FifoType* Fifo::_start [private]

Definition at line 24 of file fifo.h.

Referenced by clear(), count(), empty(), full(), and pop().

The documentation for this class was generated from the following files:

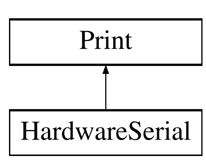
- fifo.h
- fifo.cpp

5.3 HardwareSerial Class Reference

HardwareSerial implementation.

```
#include <HardwareSerial.h>
```

Inheritance diagram for HardwareSerial:



Public Member Functions

- HardwareSerial (USART_t ∗usart, PORT_t ∗port, uint8_t in_bm, uint8_t out_-bm)
- ∼HardwareSerial ()
- void begin (long baudrate, int8_t bscale=0)
- void begin2x (long baudrate, int8_t bscale=0)
- void end ()
- uint8_t available (void)
- int read (void)
- void flush (void)
- virtual void write (uint8_t)
- void enable (bool bEn)

Interrupt Handlers

There are three possible interrupts for the USART. Receive done, Transmit done and Data Register Ready.

- void rxc ()
- void dre ()
- void txc ()

Protected Attributes

- ring_buffer ∗ _rx_buffer
- USART_t ∗ _usart
- PORT_t ∗ _port
- uint8_t _in_bm
- uint8_t _out_bm
- uint8_t _bsel
- int8_t _bscale
- long _baudrate
- bool _bEn

5.3.1 Detailed Description

HardwareSerial implementation. This class was originally copied form the Arduino source directory but has been modified somewhat to customize it for the CFA project.

Ths class wraps the hardware serial resource in the ATXmega The class handles an interupt driven receive with a fixed size receive buffer of 128 bytes. The current implementation uses a synchronous send, but a buffered send would be a great enhancement for performance purposes.

Definition at line 23 of file HardwareSerial.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 HardwareSerial::HardwareSerial (USART_t * *usart,* PORT_t * *port,* uint8_t *in_bm,* uint8_t *out_bm*)

Definition at line 112 of file HardwareSerial.cpp.

References _baudrate, _bEn, _bscale, _bsel, _in_bm, _out_bm, _port, _rx_buffer, _usart, RX_BUFFER_SIZE, and SetPointer().

```
{
    _rx_buffer = (ring_buffer*)malloc(RX_BUFFER_SIZE+2*sizeof(int));
    _usart      = usart;
    _port       = port;
    _in_bm      = in_bm;
    _out_bm     = out_bm;
    _bsel       = 0;
    _bscale     = 0;
    _baudrate   = 9600;
    _bEn        = true;
    SetPointer(_usart,this);
}
```

5.3.2.2 HardwareSerial::∼HardwareSerial ( )

Definition at line 130 of file HardwareSerial.cpp.

References _rx_buffer, _usart, end(), and SetPointer().

```
{
    end();
    free(_rx_buffer);
    _rx_buffer = 0;
    SetPointer(_usart,0);
}
```

5.3.3 Member Function Documentation

5.3.3.1 uint8_t HardwareSerial::available (void)

Definition at line 213 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::head, RX_BUFFER_SIZE, and ring_buffer::tail.

Referenced by CmdProcessor::checkCommands().

```
{
    return (RX_BUFFER_SIZE + _rx_buffer->head - _rx_buffer->tail) %
      RX_BUFFER_SIZE;
}
```

5.3.3.2 void HardwareSerial::begin (long *baudrate*, int8_t *bscale* = 0)

Definition at line 140 of file HardwareSerial.cpp.

References _baudrate, _bscale, _in_bm, _out_bm, _port, and _usart.

Referenced by main().

```
{
    uint16_t BSEL;
    _bscale = bscale;
    _baudrate = baud;

    float fPER = F_CPU;
    float fBaud = baud;

    _port->DIRCLR = _in_bm;  // input
    _port->DIRSET = _out_bm; // output

    // set the baud rate
    if (bscale >= 0) {
        BSEL = fPER/((1 << bscale) * 16 * baud) - 1;
        //BSEL = F_CPU / 16 / baud - 1;
    } else {
        bscale = -1 * bscale;
        BSEL = (1 << bscale) * (fPER/(16.0 * fBaud) - 1);
    }

    _usart->BAUDCTRLA = (uint8_t)BSEL;
    _usart->BAUDCTRLB = ((bscale & 0xf) << 4) | ((BSEL & 0xf00) >> 8);

    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = USART_RXCINTLVL_HI_gc;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
}
```

5.3.3.3 void HardwareSerial::begin2x (long *baudrate*, int8_t *bscale* = 0)

Definition at line 173 of file HardwareSerial.cpp.

References _baudrate, _bscale, _in_bm, _out_bm, _port, _usart, and SetPointer().

```
{
    uint16_t baud_setting;
    _bscale = bscale;
    _baudrate = baud;

    // TODO: Serial. Fix serial double clock.
    long fPER = F_CPU * 4;

    _port->DIRCLR = _in_bm;  // input
```

```
    _port->DIRSET = _out_bm; // output

    // set the baud rate using the 2X calculations
    _usart->CTRLB |= 1 << 1; // the last 1 was the _u2x value
    baud_setting = fPER / 8 / baud - 1;

    _usart->BAUDCTRLA = (uint8_t)baud_setting;
    _usart->BAUDCTRLB = baud_setting >> 8;


    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
    SetPointer(_usart,this);
}
```

5.3.3.4 void HardwareSerial::dre ( )

Definition at line 100 of file HardwareSerial.cpp.

```
{
}
```

5.3.3.5 void HardwareSerial::enable (bool *bEn*)

Definition at line 248 of file HardwareSerial.cpp.

References _bEn.

Referenced by main().

```
{
    _bEn = bEn;
}
```

5.3.3.6 void HardwareSerial::end ( )

Definition at line 203 of file HardwareSerial.cpp.

References _usart, and SetPointer().

Referenced by CmdProcessor::~CmdProcessor(), and ~HardwareSerial().

```
{
    SetPointer(_usart,(HardwareSerial*)0);

    // disable Rx and Tx
    _usart->CTRLB &= ~(USART_RXEN_bm | USART_TXEN_bm);
    // disable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;
}
```

5.3.3.7 void HardwareSerial::flush (void)

Definition at line 230 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::head, and ring_buffer::tail.

```
{
    // don't reverse this or there may be problems if the RX interrupt
    // occurs after reading the value of rx_buffer_head but before writing
    // the value to rx_buffer_tail; the previous value of rx_buffer_head
    // may be written to rx_buffer_tail, making it appear as if the buffer
    // were full, not empty.
    _rx_buffer->head = _rx_buffer->tail;
}
```

5.3.3.8 int HardwareSerial::read (void)

Definition at line 218 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::buffer, ring_buffer::head, RX_BUFFER_SIZE, and ring_buffer::tail.

Referenced by CmdProcessor::checkCommands().

```
{
    // if the head isn't ahead of the tail, we don't have any characters
    if (_rx_buffer->head == _rx_buffer->tail) {
        return -1;
    } else {
        unsigned char c = _rx_buffer->buffer[_rx_buffer->tail];
        _rx_buffer->tail = (_rx_buffer->tail + 1) % RX_BUFFER_SIZE;
        return c;
    }
}
```

5.3.3.9 void HardwareSerial::rxc ( )

Definition at line 94 of file HardwareSerial.cpp.

References _rx_buffer, _usart, and store_char().

```
{
    unsigned char c = _usart->DATA;
    store_char(c,_rx_buffer);
}
```

5.3.3.10 void HardwareSerial::txc ( )

Definition at line 104 of file HardwareSerial.cpp.

```
{
}
```

5.3.3.11 void HardwareSerial::write (uint8_t *c*) [virtual]

Implements Print.

Definition at line 240 of file HardwareSerial.cpp.

References _bEn, and _usart.

```
{
    if (_bEn) {
        while ( !(_usart->STATUS & USART_DREIF_bm) );
        _usart->DATA = c;
    }
}
```

5.3.4 Member Data Documentation

5.3.4.1 long HardwareSerial::_baudrate [protected]

Definition at line 33 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.2 bool HardwareSerial::_bEn [protected]

Definition at line 34 of file HardwareSerial.h.

Referenced by enable(), HardwareSerial(), and write().

5.3.4.3 int8_t HardwareSerial::_bscale [protected]

Definition at line 32 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.4 uint8_t HardwareSerial::_bsel [protected]

Definition at line 31 of file HardwareSerial.h.

Referenced by HardwareSerial().

5.3.4.5 uint8_t HardwareSerial::_in_bm [protected]

Definition at line 29 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.6 uint8_t HardwareSerial::_out_bm [protected]

Definition at line 30 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.7 PORT_t* HardwareSerial::_port [protected]

Definition at line 28 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.8 ring_buffer* HardwareSerial::_rx_buffer [protected]

Definition at line 26 of file HardwareSerial.h.

Referenced by available(), flush(), HardwareSerial(), read(), rxc(), and ~HardwareSerial().

5.3.4.9 USART_t* HardwareSerial::_usart [protected]

Definition at line 27 of file HardwareSerial.h.

Referenced by begin(), begin2x(), end(), HardwareSerial(), rxc(), write(), and ~HardwareSerial().

The documentation for this class was generated from the following files:

- HardwareSerial.h
- HardwareSerial.cpp

5.4 I2C_Master Class Reference

```
#include <I2C_Master.h>
```

Public Types

- enum DriverState {
  sIdle, sBusy, sError, sArb,
  sIDScan, sIDCheck }
- enum DriverResult {
  rOk, rFail, rArbLost, rBussErr,
  rNack, rBufferOverrun, rUnknown, rTimeout }
- enum ErrorType {
  eNone = 0, eDisabled = -1, eBusy = -2, eNack = -3,
  eArbLost = -4, eBusErr = -5, eTimeout = -6, eSDAStuck = -7,
  eSCLStuck = -8, eUnknown = -9 }
- typedef enum I2C_Master::ErrorType ErrorType

Public Member Functions

- I2C_Master (TWI_t *twi)
- ~I2C_Master ()
- void begin (uint32_t freq)
- void end ()
- ErrorType Write (uint8_t ID, uint8_t *Data, uint8_t nBytes)
- ErrorType WriteSync (uint8_t ID, uint8_t *Data, uint8_t nBytes)
- ErrorType Read (uint8_t ID, uint8_t nBytes)
- ErrorType ReadSync (uint8_t ID, uint8_t nBytes)
- ErrorType WriteRead (uint8_t ID, uint8_t *wrData, uint8_t nWriteBytes, uint8_t nReadBytes)
- ErrorType WriteReadSync (uint8_t ID, uint8_t *wrData, uint8_t nWriteBytes, uint8_t nReadBytes)
- void master_int ()
- void slave_int ()
- void WriteHandler ()
- void ReadHandler ()

- void ArbHandler ()
- void ErrorHandler ()
- void MasterFinished ()
- int testack (uint8_t ID)
- void dumpregs ()
- I2C_Master::DriverResult Result ()
- I2C_Master::DriverState State ()
- uint8_t ReadData (uint8_t *pData, uint8_t maxcnt)
- uint8_t ReadData (uint8_t index)
- uint8_t nReadBytes ()
- ErrorType CheckID (uint8_t ID)
- void Stop ()
- ErrorType ForceStartStop ()
- ErrorType WigglePin (uint8_t cnt, uint8_t pinSel, uint8_t otherState)
- void CleanRegs ()
- void loop ()
- bool busy ()
- void * isReserved ()
- bool Reserve (void *)
- void NotifyMe (I2CNotify *pMe)
- bool IsIdle ()

Protected Member Functions

- uint8_t busState ()
- void showstate ()

Private Attributes

- TWI_t * _twi
- PORT_t * _twiPort
- bool _bEnabled
- DriverState _State
- DriverResult _Result
- void * _pReserved
- I2CNotify * _pNotifyClient
- uint8_t _DeviceID
- uint8_t _nBytesWritten
- uint8_t _nWriteBytes
- uint8_t _nReadBytes
- uint8_t _nBytesRead
- uint8_t * _WriteData
- uint8_t _wrBufferLen
- uint8_t * _ReadData
- uint8_t _rdBufferLen
- uint8_t _idScanCurrent
- uint8_t _IDList [128]
- bool _ScanComplete

5.4.1 Detailed Description

Definition at line 25 of file I2C_Master.h.

5.4.2 Member Typedef Documentation

5.4.2.1 typedef enum I2C_Master::ErrorType I2C_Master::ErrorType

5.4.3 Member Enumeration Documentation

5.4.3.1 enum I2C_Master::DriverResult

Enumerator:

rOk

rFail

rArbLost

rBussErr

rNack

rBufferOverrun

rUnknown

rTimeout

Definition at line 37 of file I2C_Master.h.

```
             {
    rOk,
    rFail,
    rArbLost,
    rBussErr,
    rNack,
    rBufferOverrun,
    rUnknown,
    rTimeout
} DriverResult;
```

5.4.3.2 enum I2C_Master::DriverState

Enumerator:

sIdle

sBusy

sError

sArb

sIDScan

sIDCheck

Definition at line 28 of file I2C_Master.h.

```
                  {
    sIdle,
    sBusy,
    sError,
    sArb,
    sIDScan,
    sIDCheck
} DriverState;
```

5.4.3.3 enum I2C_Master::ErrorType

Enumerator:

eNone

eDisabled

eBusy

eNack

eArbLost

eBusErr

eTimeout

eSDAStuck

eSCLStuck

eUnknown

Definition at line 78 of file I2C_Master.h.

```
                            {
    eNone           = 0,
    eDisabled       = -1,
    eBusy           = -2,
    eNack           = -3,
    eArbLost        = -4,
    eBusErr         = -5,
    eTimeout        = -6,
    eSDAStuck       = -7,
    eSCLStuck       = -8,
    eUnknown        = -9
} ErrorType;
```

5.4.4 Constructor & Destructor Documentation

5.4.4.1 I2C_Master::I2C_Master (TWI_t * *twi*)

5.4.4.2 I2C_Master::∼I2C_Master ( )

5.4.5 Member Function Documentation

5.4.5.1 void I2C_Master::ArbHandler ( )

5.4.5.2 void I2C_Master::begin (uint32_t *freq*)

Referenced by main(), and IMU::Reset().

5.4.5.3 uint8_t I2C_Master::busState ( ) [protected]

5.4.5.4 bool I2C_Master::busy ( )

Referenced by IMU::Run().

5.4.5.5 ErrorType I2C_Master::CheckID (uint8_t *ID*)

Referenced by IMU::CheckIDs(), and IMU::QueryChannels().

5.4.5.6 void I2C_Master::CleanRegs ( )

5.4.5.7 void I2C_Master::dumpregs ( )

5.4.5.8 void I2C_Master::end ( )

Referenced by IMU::Reset().

5.4.5.9 void I2C_Master::ErrorHandler ( )

5.4.5.10 ErrorType I2C_Master::ForceStartStop ( )

Referenced by IMU::ForceStartStop().

5.4.5.11 bool I2C_Master::IsIdle ( ) [inline]

Definition at line 153 of file I2C_Master.h.

References _twi.

```
{
    return (_twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_gm)
        == TWI_MASTER_BUSSTATE_IDLE_gc;
}
```

5.4.5.12 void* I2C_Master::isReserved ( )

5.4.5.13 void I2C_Master::loop ( )

5.4.5.14 void I2C_Master::master_int ( )

5.4.5.15 void I2C_Master::MasterFinished ( )

5.4.5.16 void I2C_Master::NotifyMe (I2CNotify ∗ *pMe*)

Referenced by IMU::IMU(), and IMU::Reset().

5.4.5.17 uint8_t I2C_Master::nReadBytes ( )

5.4.5.18 ErrorType I2C_Master::Read (uint8_t *ID*, uint8_t *nBytes*)

5.4.5.19 uint8_t I2C_Master::ReadData (uint8_t ∗ *pData*, uint8_t *maxcnt*)

Referenced by IMU::Rd(), IMU::ReadWord(), IMU::StoreAccData(), and IMU::StoreGyroData().

5.4.5.20 uint8_t I2C_Master::ReadData (uint8_t *index*)

5.4.5.21 void I2C_Master::ReadHandler ( )

5.4.5.22 ErrorType I2C_Master::ReadSync (uint8_t *ID*, uint8_t *nBytes*)

5.4.5.23 bool I2C_Master::Reserve (void ∗)

5.4.5.24 I2C_Master::DriverResult I2C_Master::Result ( )

5.4.5.25 void I2C_Master::showstate ( ) [protected]

5.4.5.26 void I2C_Master::slave_int ( )

5.4.5.27 I2C_Master::DriverState I2C_Master::State ( )

5.4.5.28 void I2C_Master::Stop ( )

Referenced by IMU::ResetDevices().

5.4.5.29 int I2C_Master::testack (uint8_t *ID*)

5.4.5.30 ErrorType I2C_Master::WigglePin (uint8_t *cnt*, uint8_t *pinSel*, uint8_t *otherState*)

Referenced by IMU::FailRecovery().

5.4.5.31 ErrorType I2C_Master::Write (uint8_t *ID*, uint8_t * *Data*, uint8_t *nBytes*)

5.4.5.32 void I2C_Master::WriteHandler ( )

5.4.5.33 ErrorType I2C_Master::WriteRead (uint8_t *ID*, uint8_t * *wrData*, uint8_t *nWriteBytes*, uint8_t *nReadBytes*)

Referenced by IMU::RdAsync(), and IMU::WrAsync().

5.4.5.34 ErrorType I2C_Master::WriteReadSync (uint8_t *ID*, uint8_t * *wrData*, uint8_t *nWriteBytes*, uint8_t *nReadBytes*)

Referenced by IMU::Rd().

5.4.5.35 ErrorType I2C_Master::WriteSync (uint8_t *ID*, uint8_t ∗ *Data*, uint8_t *nBytes*)

Referenced by IMU::Wr().

5.4.6 Member Data Documentation

5.4.6.1 bool I2C_Master::_bEnabled [private]

Definition at line 51 of file I2C_Master.h.

5.4.6.2 uint8_t I2C_Master::_DeviceID [private]

Definition at line 58 of file I2C_Master.h.

5.4.6.3 uint8_t I2C_Master::_IDList[128] [private]

Definition at line 73 of file I2C_Master.h.

5.4.6.4 uint8_t I2C_Master::_idScanCurrent [private]

Definition at line 72 of file I2C_Master.h.

5.4.6.5 uint8_t I2C_Master::_nBytesRead [private]

Definition at line 62 of file I2C_Master.h.

5.4.6.6 uint8_t I2C_Master::_nBytesWritten [private]

Definition at line 59 of file I2C_Master.h.

5.4.6.7 uint8_t I2C_Master::_nReadBytes `[private]`

Definition at line 61 of file I2C_Master.h.

5.4.6.8 uint8_t I2C_Master::_nWriteBytes `[private]`

Definition at line 60 of file I2C_Master.h.

5.4.6.9 I2CNotify* I2C_Master::_pNotifyClient `[private]`

Definition at line 55 of file I2C_Master.h.

5.4.6.10 void* I2C_Master::_pReserved `[private]`

Definition at line 54 of file I2C_Master.h.

5.4.6.11 uint8_t I2C_Master::_rdBufferLen `[private]`

Definition at line 67 of file I2C_Master.h.

5.4.6.12 uint8_t* I2C_Master::_ReadData `[private]`

Definition at line 66 of file I2C_Master.h.

5.4.6.13 DriverResult I2C_Master::_Result `[private]`

Definition at line 53 of file I2C_Master.h.

5.4.6.14 bool I2C_Master::_ScanComplete `[private]`

Definition at line 74 of file I2C_Master.h.

5.4.6.15 DriverState I2C_Master::_State [private]

Definition at line 52 of file I2C_Master.h.

5.4.6.16 TWI_t* I2C_Master::_twi [private]

Definition at line 49 of file I2C_Master.h.

Referenced by IsIdle().

5.4.6.17 PORT_t* I2C_Master::_twiPort [private]

Definition at line 50 of file I2C_Master.h.

5.4.6.18 uint8_t I2C_Master::_wrBufferLen [private]

Definition at line 65 of file I2C_Master.h.

5.4.6.19 uint8_t* I2C_Master::_WriteData [private]

Definition at line 64 of file I2C_Master.h.

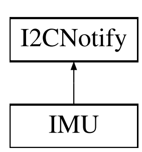
The documentation for this class was generated from the following file:

- I2C_Master.h

5.5 I2CNotify Class Reference

```
#include <I2C_Master.h>
```

Inheritance diagram for I2CNotify:



Public Member Functions

- virtual void I2CWriteDone ()=0
- virtual void I2CReadDone ()=0
- virtual void I2CBusError ()=0
- virtual void I2CArbLost ()=0
- virtual void I2CNack ()=0

5.5.1 Detailed Description

Definition at line 14 of file I2C_Master.h.

5.5.2 Member Function Documentation

5.5.2.1 virtual void I2CNotify::I2CArbLost ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.2 virtual void I2CNotify::I2CBusError ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.3 virtual void I2CNotify::I2CNack ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.4 virtual void I2CNotify::I2CReadDone ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.5 virtual void I2CNotify::I2CWriteDone ( ) `[pure virtual]`

Implemented in IMU.

The documentation for this class was generated from the following file:

- I2C_Master.h