The Report committee for Ryan Allen Brown

Certifies that this is the approved version of the following report:

Inertial Solution for Accurately Assessing location Coordinates

(ISAAC)

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Christine Julien

_____

Suzanne Barber

Inertial Solution for Accurately Assessing location Coordinates

(ISAAC)

By

Ryan Allen Brown, B.S.E.E.

Report

Presented to the Faculty of the Graduate School

Of the University of Texas at Austin

In Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2010

Dedication

This report is dedicated to my son Isaac.

Acknowledgements

I would like to thank Dr. Christine Julien for being my supervisor during this project and planting the seed for the need for an improved location support system. I would also like to thank Dr. Suzanne Barber for being my second reader and creating the Option III Master's of Software Engineering Program. It is this program that has allowed me to pursue my master's degree while working full-time.

Inertial Solution for Accurately Assessing location Coordinates

(ISAAC)

by

Ryan Allen Brown, M.S.E.

The University of Texas at Austin, 2010

Supervisor: Christine Julien

Accurately determining one's location has long been a persistent problem in navigation and has reappeared in recent years in the field of mobile computing. The ability to determine a device's location indoors is needed for both automation and efficient communication in collaborative robotic and sensor networks. Technologies such as indoor GPS transmitters and Cricket have been employed, but have had limited success due to cost, accuracy, and power consumption. The Inertial Solution for Accurately Assessing location Coordinates (ISAAC) was developed as a means of filling this need without requiring infrastructure or expensive components to accurately determine position, inside or outside.

ISAAC is based on modified six-degrees-of-freedom (6DOF) dead reckoning algorithms currently being used by Unmanned Aerial Vehicles (UAV). UAVs typically have access to other types of sensors to supplement and/or replace the IMU measurements. ISAAC was implemented using a low-cost MEMS 6DOF IMU in which the onboard firmware was modified to incorporate the dead reckoning calculations and communications necessary to realize ISAAC. ISAAC was implemented as a portable unit which communicated with a host computer through an RS-232 interface.

ISAAC did not perform as well as expected; the location coordinates were very inconsistent with device movements and did not produce any useful data. The correct intermediate results of the calculations and subsequent review by a local subject-matter-expert implies that the source of the erroneous results lie with the accuracy and precision of the MEMS IMU. ISAAC presents a foundation for future work where more robust sensors and/or filtering can be used for further examination of inertial-based location systems.

Table of Contents

Introduction 1

Literature Review 1

Approach 5

Implementation 10

Results 18

Conclusions 21

Appendix A – Modified and New Firmware Source Code 22

Appendix B – DeadReck Source Code Package 34

Appendix C – Dead Reckoning Host Software Source Code 38

Bibliography 42

Vita 44

List of Figures

Figure 1: ISAAC Prototype Block Diagram 10

Figure 2: CHR-6d Inertial Measurement Unit [14] 11

Figure 3: Bodhilabs VPack Boost Regulator [16] 12

Figure 4: Windows Configuration Utility 14

Figure 5: ISAAC Dead Reckoning Host 17

List of Tables

Table 1: Accelerometer Electrical Characteristics [14] 11

Table 2: Gyroscope Electrical Characteristics [14] 12

Table 3: Inertial Sensor Bias Values 13

Table 4: Location Request Message Structure 15

Table 5: Response Message Structure 15

Introduction

Accurately determining a mobile device's location while indoors has long been a problem in mobile computing. This ability is needed to support energy-efficient location-based communication routing algorithms, location-based services such as home automation and/or site mapping, and industrial robotic applications such as manufacturing or warehouse operations. Technologies such as indoor GPS transmitters and Cricket have been employed but have had limited success due to cost, accuracy, and power consumption.

The Inertial Solution for Accurately Assessing location Coordinates (ISAAC) is an approach to determining a device's location that requires no infrastructure or expensive components to accurately determine position, inside or outside. ISAAC is comprised of only inertial sensors which enables the unit to be used in a wide variety of applications such as being mounted within a vehicle or on a person where traditional dead reckoning devices use application-specific sensors (e.g. pedometers, wheel rotation, airspeed) for determining location. This system is based on six degrees of freedom (6DOF) dead reckoning algorithms currently being used by unmanned aerial vehicles. These algorithms have been modified to accommodate the mobile device's ability to move in any direction regardless of orientation.

The envisioned system architecture for applications which employ ISAAC is one in which an ISAAC unit is affixed to a host (machine, vehicle, mobile device, or node) and interfaced through a bi-directional RS-232 connection. ISAAC would function as an independent unit and respond to location queries by the host unit. To provide application flexibility in this architecture, the host has the responsibility of converting the ISAAC location coordinates to meaningful information (e.g. location on a map).

The purpose of this project is to implement an ISAAC prototype using a low-cost Micro-Electro-Mechanical Systems (MEMS) Inertial Measurement Unit (IMU) and evaluate/characterize the performance of this prototype. This paper provides a literature survey of related work, a description to the approach taken to complete this project, detailed descriptions of the prototype hardware and software, evaluation results, and a conclusion including suggestions for future work.

Literature Review

Accurately determining one's location has been a persistent problem in navigation for the past few centuries. Many various methods have been developed to solve this problem including astronomical navigation, compass and pace count, signal triangulation, and the Global Positioning System (GPS). This section briefly summarizes various location-determination systems developed for use with robotics, sensor networks, and aircraft. The uses, features, and limitations of GPS are fairly well-known and have been omitted from this section.

Patents

Patents US 7,522,999 [1] and US 5,612,688 [2] describe inertial-based apparatuses that are used for waypoint navigation. Both of these patents prescribe many of the same inertial sensors employed in ISAAC, but were developed for a very specific application: monitoring the return vector to a set origin (waypoint). In addition, these inventions are comprised of additional hardware in the moving devices to display the return vector to the origin. Patent US 5,612,688 also prescribes a charging base which initiates location tracking, sets the waypoint origin, and charges the sensed device's battery. ISAAC differs from these to inventions in that ISAAC does not provide a means to display location coordinates, and does not require external hardware for initialization and/or setting the origin.

Patent US 7,587,277 [3] describes an apparatus which applies magnetometer measurements to the gyroscope measurements to improve pitch, roll, and yaw measurements. The intent of ISAAC is to calculate location based using only gyroscope and accelerometer sensors. However, the use of magnetometer measurements will be a suggestion for future work.

Patent US 7,400,246 [4] describes an apparatus used to locate workers in a mine. This system is wearable and provides a means of wirelessly transmitting velocity telemetric data in TCP/IP format. This patent differs from ISAAC in that the apparatus consists of a wireless transmitter that is used to send inertial sensor data to a central computer where it is then processed to determine the location of the apparatus. This patent lists potential sources of error (e.g. Earth spin, Earth Loop Error, Shuler Oscillation Error, etc.) associated with inertial sensors that could also effect the accuracy of ISAAC.

Aviation Evaluations

Berman et al. [5] discuss candidates for GPS backup in aviation navigation and perform a trade-off study comparing the candidates' performance. The candidates examined were stand-alone dead reckoning (air speed and magnetic compass only), dead reckoning with inertial sensors, and stand-alone inertial sensors. The results of this trade-off study indicated that inertial sensors did not increase dead reckoning accuracy unless the sensors used were of "tactical quality." This system differs with ISAAC in that inertial sensors are being used evaluated as potential GPS backup solutions. The results of this paper raise concern of the potential accuracy of ISAAC.

Brown et al. [6] evaluate the performance of a low cost MEMS-based integrated GPS/Inertial Navigation System (INS) developed for unmanned aerial vehicles (UAVs). During this evaluation, it was determined that because of the poor quality of the MEMS sensors, navigation performance rapidly degrades when GPS is lost. To counter this degradation, the authors incorporated altimeter readings, dead reckoning (heading and speed) sensors, video sensor (to track motion of objects in images), and communication time of arrival (TOA) information into their navigation system. ISAAC does not incorporate the use of these additional sensors (including GPS) which are outside of the intent of ISAAC to be an inertial-only location system. This paper does, however, suggest those MEMS inertial sensors are not sufficient for dead reckoning without the aid of additional sensors.

Pedometer-Based Location Systems

Chung et al. [7] describe two methods aimed at improving dead-reckoning accuracy: precision calibration of the gyroscope (single axis), and the implementation of an indirect feedback Kalman filter that fuses FOG sensor data with the mobile robot's odometry system. They conclude that using both methods concurrently improves accuracy 9-fold. Though the navigation system evaluated is very different from ISAAC, sensor calibration and using advanced Kalman filtering techniques will be considered during the implementation of ISAAC.

Fuke et al. [8] use inertial sensors to extend the classic dead reckoning approach to the case of a mobile robot moving on uneven terrain. This system uses gyros and accelerometers for attitude determination. This information is then coupled with odometry readings to calculate the rover's location. This system differs from ISAAC in that ISAAC uses only inertial sensors to determine its location.

Dr. Tom Judd of Point Research Corporation [9] developed a lightweight miniature Dead Reckoning Module (DRM) for navigation by personnel traveling on foot. This system consists of a three-dimensional compass and an electronic pedometer that calculates position through standard dead reckoning algorithms. Data is reported from the DRM through a standard RS-232 serial interface. ISAAC is similar to this DRM in that position data is calculated and reported via an RS-232 serial interface, but ISAAC uses only inertial sensors for position calculation; DRM employs pedometers to calculate distance traveled and compasses to determine direction of travel.

Randell et al. [10] compare dead reckoning measurement techniques in hopes of finding a configuration suitable for use by pedestrians. This study compared different technologies in different mounting configurations through different scenarios. In each case study, step count was used for speed and was measured using accelerometers or pedometers. In addition, two and three dimensional compasses (with a rate gyroscope for error correction) were used to measure direction. The techniques analyzed in this study differ with ISAAC in that ISAAC does not require step count, and uses 3-axes accelerometers and rate gyroscopes for all of its velocity measurements.

RF-Based Location Systems

Niculescu et al. [11] propose a positioning system that works as an extension of both distance vector routing and GPS positioning. In this positioning system, the approximate location for all nodes in a network is calculated where some nodes are capable of determining their location. The locations of the other nodes are calculated using triangulation of estimated distances to the "landmark" nodes using one of the three following methods: "DV-Hop" where triangulation distances are estimated using an average size for one hop, "DV-Distance" where triangulation distances are estimated based on received signal strength, or "Euclidean" which uses known distances from neighboring nodes to form a quadrilateral for location

estimation. This system differs from ISAAC in that it is based on inertial measurements of the nodes' movements and not based on communication with other nodes. However, ISAAC-equipped nodes could potentially be deployed in APS as "landmark" nodes.

Priyantha et al. [12] propose Cricket, a location support system that is comprised of RF/UHF beacons distributed at known locations, and nodes capable of receiving RF and UHF signals. Distance is determined by measuring the time delay between receiving a RF beacon and a UHF beacon. The distance from the beacon is then calculated using the speed of light, speed of sound, and the time delay. This system is primarily designed to be used indoors and requires the use of expensive RF and UHF hardware on each node. ISAAC differs with Cricket in that ISAAC requires no additional infrastructure equipment and is implemented using only inertial sensors.

Event-Based Location Systems

Stoleru et al. [13] propose Spotlight, an event-based system used to calculate location. In Spotlight, an event (e.g., bright light) is generated over a specific area (containing sensor nodes) by a device located at a known location. The sensor nodes detect the events and report back to the Spotlight device the timestamps when the events were detected. The spotlight device then computes the location of the sensor nodes. This system requires an external device to both generate an event and calculate node locations, which may not be feasible in some [tactical] environments. This system differs from ISAAC in that ISAAC does not require any external hardware or events to operate.

Approach

The purpose of this project was to develop a location system using inertial sensors, or specifically an inertial measurement unit (IMU). This project was comprised of four efforts: prototype hardware development, host software development, prototype firmware development, and prototype evaluation.

Prototype Hardware

The original hardware prototype development approach for this project was to develop an ISAAC printed circuit board (PCB) with all of the necessary components (e.g., gyroscopes, accelerometers, microcontroller) to implement an IMU. After some research, I began also researching off-the-shelf IMUs to see if any comparable devices were available. During this research, I found the CH Robotics CHR-6d IMU [14]. This IMU contained the gyroscopes and accelerometers I had already identified for use in this project, but also contained a higher-precision analog to digital converter than the one I had selected. In addition, this unit contained a pre-programmed microcontroller whose source code and corresponding compiler where available for free. At $125 (and no PCB development time), this IMU was the perfect solution. The ISAAC prototype (see "Implementation" section) was then constructed to provide power and facilitate communications to this IMU. Once the prototype was constructed, it was tested using the CH Robotics Windows Configuration Utility [17] to ensure that the unit was functioning correctly before modifying any of its firmware.

Host Software

The host software was developed to be very simple and only provide the raw data coordinates necessary to evaluate the ISAAC prototype. The approach for developing this software was to enable a user to request the location coordinates, establish communication with the ISAAC prototype, and display the ISAAC location coordinates. This application is discussed in more detail in the "Implementation" section of this report.

Prototype Firmware

The approach to developing the prototype firmware was to implement ISAAC within the existing IMU firmware construct without inhibiting or degrading the IMU's ability to perform its pre-programmed (non-ISAAC) functions. This approach was easily achieved through setting the unit to operate in "listen" mode. This mode waits for a request over the serial interface, and once a request is made, the IMU responds as directed (e.g., it sends sensor values). Within this mode, I was able to enable ISAAC location requests/processing while still allowing legacy requests to be made and honored. The ISAAC firmware was developed and evaluated iteratively so that each algorithm/calculation (discussed in the next section) could be evaluated and proven before proceeding to the next. The firmware modifications are discussed in more detail in the "Implementation" section of this report.

Location Algorithms

The ISAAC prototype converts six degrees of freedom (6DOF) inertial measurements to local position coordinates using aircraft navigation algorithms. These algorithms essentially transform the IMU measurements from the free-body reference frame to a more meaningful earth-based reference frame, then integrates these transformed rates/accelerations to positional coordinates. The inertial measurements are acquired from a 6DOF IMU which measures acceleration through three orthogonal axes and the rotation about these same axes. The local location coordinates that ISAAC calculates are defined in the east-north-up earth-referenced coordinate system as calculated from the free-body axes determined when the prototype is turned on[1]. The algorithms/calculations necessary to compute these location coordinates are derived and described in the following sections:

Pre-Processing

Before, attaining the inertial measurements and calculating the positioning coordinates, the IMU is first allowed to stabilize, then the sensor measurements are zeroed (i.e. bias values reset), and finally the sensor measurements are filtered and fed into the ISAAC location coordinates calculations. FIR filter

[1] For proper operation, the prototype must be initialized so that the IMU PCB is flat-side-down as pictured in Figure 2.

values were optimized for the ISAAC prototype using the Windows Configuration Utility [17]. It was determined that the optimal filter settings were 32 taps, at 140 Hz corner frequencies. These are the manufacturer's default values and appeared to result in the best performance.

Convert Sensor Measurements to Meaningful Values

The CHR-6d datasheet [14] prescribes the scaling to convert the sensor measurements to meaningful values. These scale factors are identified by "GYRO_SCALE_FACTOR" and "ACCEL_SCALE_FACTOR" in the CHR-6d source code [17] and have the values 0.02014 and 0.0001678, respectively. These calculations are shown below:

Rotational Velocity (°/s) = GYRO_SCALE_FACTOR × (Gyro Measurement)

Acceleration (°/s^2) = ACCEL_SCALE_FACTOR × (Accelerometer Measurement)

Calculate Free-Body Angular Velocities

Next, the three-dimensional free-body angular velocities were determined by calculating the midpoint between the previous gyro measurement and the current. This was accomplished in firmware by storing the old gyro values as static variables and performing the following calculation for each gyro axis:

AV (rads/s) = Previous AV + (New AV - Previous AV) / 2

Calculate Euler Angular Velocities

To provide meaningful values, these free-body referenced angular velocities (p, q, r) were then transformed to a coordinate system based on the nearest tangential plane to the earth (i.e., east-north-down) coordinate plane. That is, time-derivatives of the Euler angles (ψ = yaw, φ = roll, θ = pitch) were calculated using the following formulas from [18]:

$$\dot{\psi} = \frac{q \sin\varphi + r \cos\varphi}{\cos\theta}$$

$$\dot{\theta} = q \cos\varphi - r \sin\varphi$$

$$\dot{\varphi} = p + (q \sin\varphi + r \cos\varphi) \tan\theta$$

Calculate Euler Angles

Next, the three-dimensional earth-referenced Euler angular velocities were converted to angles by integrating the accelerations over time. To improve the accuracy of the calculated angles, the angular velocity midpoint (average) values from the last measurement to the current measurement were used in the calculations. In firmware, this was performed by storing the angles and old angular velocity values as static variables and updating them with each calculation iteration as follows:

Angle (rads) = (Previous Angle) + (Angular Velocity) × (Elapsed Time Since Last Measurement)

Calculate Three-Dimensional Free-Body Accelerations

The three-dimensional free-body accelerations were then determined by calculating the midpoint between the previous accelerometer measurement and the current. This was accomplished in firmware by storing the old accelerometer values as static variables and performing the following calculation for each accelerometer axis:

Accel (m/s/s) = Previous Accel + (New Accel - Previous Accel) / 2

Calculate Three-Dimensional Free-Body Velocities

Next, the free-body accelerations were converted to the corresponding velocities by simply multiplying the accelerations by time and adding the product to the previous velocity values (static local variables).

Velocity (m/s) = (Previous Velocity) + (Acceleration) × (Elapsed Time Since Last Measurement)

Calculate Three-Dimensional Earth-Referenced Velocities

The free-body velocities (u, v, w) were then converted to earth-referenced velocities using the formulas below from [18]. In addition, velocity in the Z axis was offset to account for gravity; the IMU measures gravity as a 1 g acceleration downward.

$$\dot{x}_e = \{u_e \cos\theta + (v_e \sin\varphi + w_e \cos\varphi)\sin\theta\}\cos\psi - (v_e \cos\varphi - w_e \sin\varphi)\sin\psi$$

$$\dot{y}_e = \{u_e \cos\theta + (v_e \sin\varphi + w_e \cos\varphi)\sin\theta\}\sin\psi + (v_e \cos\varphi - w_e \sin\varphi)\cos\psi$$

$$\dot{z}_e = -u_e \sin\theta + (v_e \sin\varphi + w_e \cos\varphi)\cos\theta$$

Calculate Three-Dimensional Earth-Referenced Location Coordinates

Finally, the location coordinates were calculated by multiplying the earth-referenced velocities by time and adding the result to the previous coordinates. The Z axis coordinate is inverted to transform the coordinate values from “east-north-down” to “east-north-up” so that positive Z coordinate values point up. The results were stored to a global variable which could be read by the packet handler routine to report in response to location queries. The following calculation was performed for each location coordinate:

$$\text{Location (m)} = (\text{Previous Location}) + (\text{Velocity}) \times (\text{Elapsed Time Since Last Measurement})$$

Implementation

Hardware

The ISAAC prototype was intended to provide a fully-functional 6DOF test bed that was both easy to implement and available at a low cost. To realize these design goals, the components/sensors necessary to construct an IMU, and many commercially-available IMUs were evaluated against the following high-level functional requirements:

1. The unit must be capable of measuring 3-axes of rotation and corresponding 3-axes of acceleration.
2. The unit must have sufficient onboard processing capacity to both read the inertial sensors and calculate meaningful inertial values from these measurements.
3. The unit must be capable of serially communicating (ideally RS-232) with a host (e.g., a personal computer).

Through this research, the CHR-6D [14] Inertial Measurement Unit (IMU) was selected. This unit met all of the aforementioned requirements at a low cost and CH Robotics supplies a development kit to modify their onboard firmware. In addition, this unit contained the same sensors selected for the initial homemade IMU. Once this unit was selected, the ISAAC prototype (see Figure 1 below) was developed to provide power to the IMU and enable communications with an external host. These components are further described in the following paragraphs.

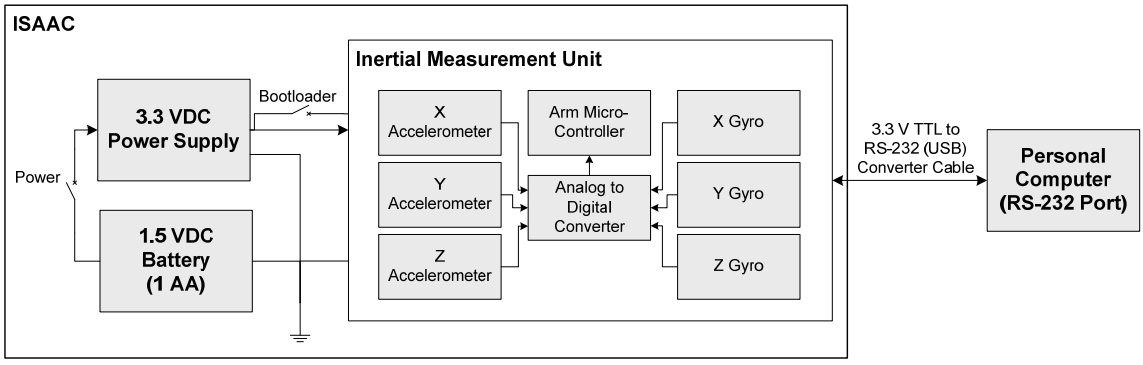

Figure 1: ISAAC Prototype Block Diagram

Inertial Measurement Unit

The IMU (see Figure 2 below) is the primary component of the ISAAC prototype. The IMU performs all of the inertial measurements, performs advanced software filtering, calculates the location coordinates (with firmware changes), and outputs those coordinates through a 3.3V TTL 115200 baud Universal Asynchronous Receiver Transmitter (UART). In addition, the IMU is capable of being reprogrammed in the circuit through the use of a boot loader.

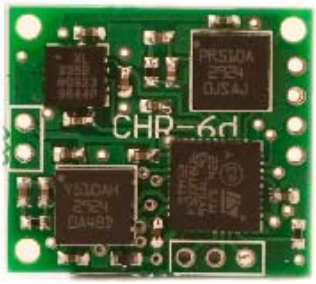

Figure 2: CHR-6d Inertial Measurement Unit [14]

The IMU is comprised of a 32-bit ARM Cortex processor, three orthogonal accelerometers, and three orthogonal gyroscopes, and an analog to digital converter to read the inertial sensors. The accelerometers (see Table 1 for specifications) and MEMS gyroscopes (see Table 2 for specifications) are measured around the same axes (e.g., the "X" axis gyroscope measures rotation around the corresponding linear acceleration axis) and are "oversampled through the onboard analog to digital converter to provide 16-bit resolution to the processor" [14].

Table 1: Accelerometer Electrical Characteristics [14]

Parameter	Test Condition	Min.	Typ.	Max.	Unit
Measurement Range		± 3.6	± 3.6		
Sensitivity	Vdd = 3 V	270	300	330	mV/g
Sensitivity Change Due to Temperature	Vs = 3V		0.01		% / °C
X,Y Noise Density			150		µg / rt Hz
Z Noise Density			300		µg / rt Hz
X,Y 0g Voltage	Vs= 3V	1.35	1.5	1.65	V
Z 0g Voltage	Vs = 3V	1.2	1.5	1.8	V

Table 2: Gyroscope Electrical Characteristics [14]

Parameter	Test Condition	Min.	Typ.	Max.	Unit
Measurement Range	4x OUT (amplified)		± 100		°/s
	OUT (not amplified)		± 400		°/s
Sensitivity	4x OUT (amplified)		10		mV / °/s
	OUT (not amplified)		2.5		mV / °/s
Sensitivity Change Due to Temperature			0.03		% / ° /s
Zero-Rate Level			1.23		V
Reference Voltage			1.23		V
Zero-Rate Level Change vs. Temperature	Delta from 25 °C		0.02		°/ s /°C

TTL to RS-232 Converter Cable

A 3.3V TTL RS-232 UART to USB converter is necessary for a personal computer to communicate with the CHR-6d IMU. Typically, a converter is not required for host devices capable of 3.3V UART serial communication. This communication interface is needed to reprogram and configure the onboard ARM processor and receive the location coordinates. The FTDI Chip TTL-232R-3V3-AJ [15] cable was chosen for the ISAAC prototype because the 3.3V converter is built into the cable, thus reducing the number of prototype components.

Power Supply

The CHR-6d IMU requires regulated 3.3 VDC power capable of providing at least 56 mA [14]. The Bodhilabs VPack Boost Regulator [16] was chosen as the power supply because of its relatively small footprint (for the ISAAC prototype), its ability to function with one standard AA battery, and the fact that it provides regulated 3.3 VDC, 100 mA current output. This power supply is shown below in Figure 3.

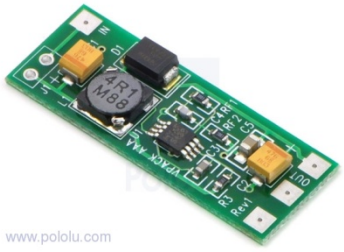

Figure 3: Bodhilabs VPack Boost Regulator [16]

Firmware

ISAAC is, at the lowest level, a modified IMU. The main difference between ISAAC and an ordinary IMU is how the inertial data is fused and utilized. This difference is accommodated strictly in the IMU onboard processor's firmware. The firmware supplied with the CHR-6d IMU is very feature rich and flexible to meet the needs of many different types of users. In addition, all CHR-6d firmware and the corresponding C compiler (Ride7) are available for free download. This firmware served as the framework to develop the ISAAC prototype. The following sections describe the modifications made to the existing firmware and the new code developed to transform the IMU into an ISAAC prototype.

Modifications

The first step in developing the ISAAC firmware was to optimize the prototype hardware performance using the supplied Windows Configuration Utility [17] (see Figure 4 below). This utility facilitates IMU setup, calibration, and configuration, and was used to determine bias values for each sensor, filter corner frequencies, number of filter taps, and see the effects of these changes in real time. Once these values were optimized, they were hard-coded into the "GetConfiguration" function located in chr6d_config.c [17]. The optimal values used for this project are FIR corner frequencies of 140 Hz, 32 FIR filter taps for each sensor and the bias values shown in Table 3 below. The "GetConfiguration" function and the associated modifications are shown in Appendix A (flagged as "Ryan").

Table 3: Inertial Sensor Bias Values

Sensor	Bias (Hexadecimal)
X Accelerometer	0x7E5E
Y Accelerometer	0x80B6
Z Accelerometer	0x849E
X Gyro	0x6089
Y Gyro	0x6057
Z Gyro	0x60E6

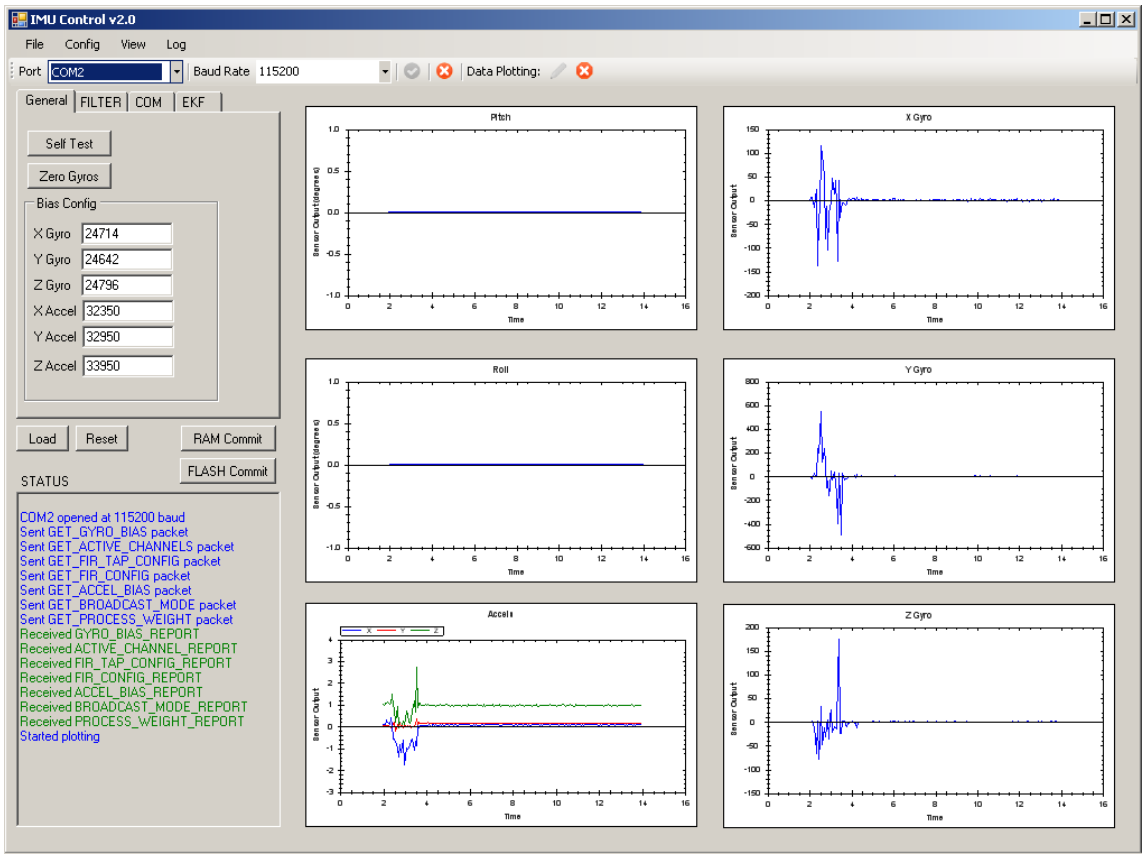

Figure 4: Windows Configuration Utility

After the prototype hardware measurements were optimized and hard-coded, the firmware was modified to enable querying and reporting of the ISAAC location coordinates. First the "GetConfiguration" function located in chr6d_config.c [17] was modified to be in listen (vs. broadcast) mode by default. This setting reduces power consumption and increases location calculation accuracy by decreasing the computation between measurements and power needed to continually send messages. Next, ISAAC messages were defined to be consistent with the native CHR-6d message structure and protocol. The request message and response message structures are shown below in Table 4 and Table 5, respectively. To comply with the existing message structure definitions, the constants GET_LOCATION_COORDINATES (0x09) and LOCATION_COORDINATES_REPORT (0xBF) were defined in chr6d_usart.c [17]. Once the messages were defined, the "ProcessPacket" function in chr6d_packet_handler.c [17] was modified to accept (recognize the packet type) and respond to the request

message. The response message is generated by the "SendLocationCoordinatesPacket" function discussed in the "new code" section. This function and its modifications are shown in Appendix A (flagged as "Ryan").

Table 4: Location Request Message Structure

's'	'n'	'p'	0x09	0x01	0x5A
Message Preamble			Packet Type	Checksum MSB	Checksum LSB

Table 5: Response Message Structure

's'	'n'	'p'	0xBF	0x0C	12-Byte Data Payload Sent in X, Y, Z order, MSB to LSB. All 32 Bits of Each Floating-Point Value are Sent.	0x??	0x??
Message Preamble			Packet Type	Packet Length		Checksum MSB	Checksum LSB

Finally, the firmware was modified to continually calculate the location coordinates. This was done by inserting a call to the "DeadReck_CalculateCoordinates" function (discussed in the "new code" section) in the "main" function in main.c [17]. The main function serves as the program entry point and contains an infinite loop to maintain continuous program execution. This function and its modifications are shown in Appendix A (flagged as "Ryan").

New Code

In addition to the aforementioned firmware modifications, two new firmware functions were developed to realize ISAAC. The first function added was the "SendLocationCoordinatesPacket" function. The purpose of this function is to package and send the current location coordinates using the message structure shown in Table 5 above. To be consistent with the CH Robotics source code, this function was added to chr6d_packet_handler.c [17].

The second, and most important function developed for ISAAC is the "DeadReck_CalculateCoordinates" function. This function performs all of the coordinate calculations discussed in the "Approach" section and is continuously called from the "main" function in main.c [17]. In addition, this function allows the IMU to stabilize for one second following startup and zeroes the measured sensor values after this stabilization period. At a very high level, this function does the following:

1. Zero sensor measurements after one second stabilization period

2. Read sensor measurements (from the FIR filter outputs)
    a. Convert measurements to meaningful values
3. Read time since last sensor read
    a. Reset timer
4. Calculate free-body angular velocities
5. Calculate Euler angular velocities
6. Calculate Euler angles
7. Calculate thee-dimensional free-body accelerations
8. Calculate three-dimensional free-body velocities
9. Calculate three-dimensional earth-referenced velocities
10. Calculate three-dimensional earth-referenced location coordinates

This function and all of its essential data members are located in the new files DeadReck.c and DeadReck.h shown in Appendix B.

Host Software

The host software was developed as a means to retrieve location coordinates from the ISAAC prototype. To reduce complexity of the ISAAC prototype, the host application uses the location coordinates to create the meaningful, application-specific information such as a location on a map, motion vectors, or raw coordinate data. For simplicity, raw data coordinates were selected to be displayed in the host application for this project.

This "Dead Reckoning Host" application was developed in C using LabWindows/CVI. With this application, the user can poll the ISAAC prototype by clicking the "Get Coordinates" button (see Figure 5 below). Once this button is clicked, the application sends a location request message to the ISAAC prototype through the RS-232 serial interface. The ISAAC prototype then responds by sending the location message over the serial interface. The location packet contains 32-bit floating-point X, Y, and Z coordinate values. Finally the host application decodes the message and displays the three-dimensional location

coordinates (in meters). This application serves as the primary means for evaluating the ISAAC prototype. The complete source listing for the Dead Reckoning Host can be found in Appendix C.

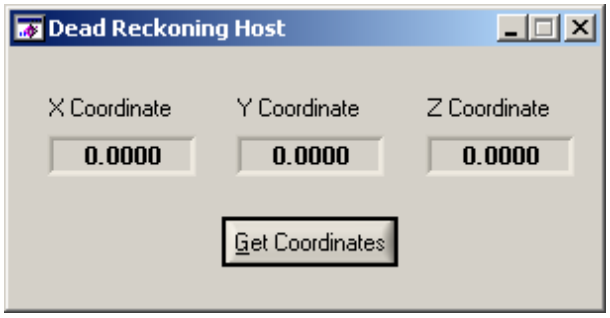

Figure 5: ISAAC Dead Reckoning Host

Results

The evaluation of the ISAAC was performed iteratively through firmware development to ensure each calculation was working properly before progressing to the next. To reduce overhead with the bench testing, the results were sent in the location messages. This enabled the existing host software and message-handling firmware modifications to work during bench testing with no additional modifications. The performance of the basic IMU functions was verified using the Windows Configuration Utility [17] prior to testing the modified firmware. The results of the bench testing are as follows (in order of development):

Calculate Free-Body Angular Velocities

There was significant noise in the free-body angular velocity calculations. When this unit was initialized and held very still, the angular velocities were all approximately 0.045 radians/second (2.57 degrees/second) with oscillations from 0.035 to 0.055 radians/second. This noise could not be corrected through the Windows Configuration Utility [17] FIR or bias settings, or by increasing the IMU stabilization period in firmware. It was observed from the configuration utility that the (filtered) gyroscope measurements were very noisy. This would account for the errors in these calculations. It was also observed that the angular velocities changed correctly in response to various movements of the prototype.

Calculate Euler Angular Velocities

As with the free-body angular velocities, noise was observed with the Euler angular velocities. The Euler angular velocities oscillated between +0.010 (0.57 degrees/second) and -0.10 radians/second when the unit was at rest. However, the velocities appeared to be changing correctly in response to motion. This is to be expected since these values are calculated using the free-body velocities.

Calculate Euler Angles

As observed with the Euler angular velocities, there were significant errors (noise) in the values. The angles were observed to drift at a rate of 0.01 radians/second (0.57 degrees/second), but did appear to

calculate the correct values as the prototype was rotated. This performance is to be expected because these values are calculated using the Euler angular velocities.

Calculate Three-Dimensional Free-Body Accelerations

Gravity (1 g down) was used as the controlled test input for these calculations. The ISAAC prototype was rotated to various positions in 90° increments to verify proper acceleration measurement. It was observed in each axis that there was a +/- 0.026 g fluctuation in the measured values. These fluctuations could not be corrected through the Windows Configuration Utility [17] FIR or bias settings.

Calculate Three-Dimensional Free-Body Velocities

This evaluation was performed similarly to the free-body acceleration evaluations. Gravity was used to as the input acceleration, and the prototype was rotated about its axes at 90° increments. The velocities appeared to respond well to the rotations but exhibited drift of approximately 0.01 meters/second. This is attributed to the noise in the inertial acceleration measurements.

Calculate Three-Dimensional Earth-Referenced Velocities

The earth-referenced velocity calculations were slightly modified to enable testing with gravity. These equations, as coded, remove gravity so that only actual prototype movement is captured, which would prevent me from using gravity as my controlled input. These calculations did not show any predictable, consistent behavior. The formulas and corresponding firmware were checked by a local aircraft autopilot control-law expert, and no errors were found in either the formulas or source code.

Calculate Three-Dimensional Earth-Referenced Location Coordinates

These calculations encompass the results from all of the aforementioned calculations. This test yielded similar results as the earth-referenced velocity calculations, but also had continually-increasing drift in the measurements. In addition, there was no predictable or consistent behavior seen when changing the orientation of the unit. These results are due to the use of low-cost (MEMS) inertial components; inertial

measurement errors are compounded as they propagate through the calculations and transformations, which cause the end results to be useless.

Conclusions

ISAAC was intended to be a low-cost means to determine one's location using inertial sensors. The purpose of this project was to implement and test an ISAAC prototype, and in a sense, determine the feasibility of such a system. The ISAAC prototype did not perform very well and did not provide any useful location data. These erroneous results are consistent with the findings from the MEMS inertial sensor trade studies described in the related work. The poor results of this project do not necessarily conclude that determining one's location with inertial sensors is impossible, just that a low-grade MEMS IMU should not be used in such applications.

There are many high-precision inertial sensors available; some are even packaged with magnetometers to further increase their accuracy. A potential future project associated with ISAAC would be to implement the same algorithms on one of these more precise systems to demonstrate the feasibility of using inertial sensors for determining one's location. In addition, more advanced filtering techniques (e.g., Kalman filters) should be researched, and applied, to aid in reducing noise and stability problems.

Appendix A – Modified and New Firmware Source Code

```
/*****************************************************************
* Function Name  : GetConfiguration
* Input          : None
* Output         : None
* Return         : None
* Description    : Fills the gConfig structure with IMU configuration
*                  data. If configuration data has been written to
*                  flash, then load it. If not, then use factory
*                  defaults.
*****************************************************************/
void GetConfiguration( void )
{
    // If flash has not been programmed yet, then use default
    //   configuration.  Otherwise, load configuration from flash
    if( GET_ACCEL_X_OFFSET() == 0xFFFF )
    {
        // Gyro and accelerometer default biases
        gConfig.x_accel_bias = 0x7E5E; //0x802F; Ryan
        gConfig.y_accel_bias = 0x80B6; //0x802F; Ryan
        gConfig.z_accel_bias = 0x849E; //0x802F; Ryan
        gConfig.x_gyro_bias = 0x6089;  //0x5F6B; Ryan
        gConfig.y_gyro_bias = 0x6053;  //0x5F6B; Ryan
        gConfig.z_gyro_bias = 0x60E6;  // 0x5F6B; Ryan

        // Gyro and accelerometer default FIR corner frequencies
        gConfig.x_accel_corner = FIR_CORNER_140HZ;
        gConfig.y_accel_corner = FIR_CORNER_140HZ;
        gConfig.z_accel_corner = FIR_CORNER_140HZ;
        gConfig.x_gyro_corner = FIR_CORNER_140HZ;
        gConfig.y_gyro_corner = FIR_CORNER_140HZ;
        gConfig.z_gyro_corner = FIR_CORNER_140HZ;

        // Gyro and accelerometer default FIR filter tap lengths
        gConfig.x_accel_taps = FIR_TAPS_32;
        gConfig.y_accel_taps = FIR_TAPS_32;
        gConfig.z_accel_taps = FIR_TAPS_32;
        gConfig.x_gyro_taps = FIR_TAPS_32;
        gConfig.y_gyro_taps = FIR_TAPS_32;
        gConfig.z_gyro_taps = FIR_TAPS_32;

        // Channel enable/disable flags
        gConfig.x_accel_enabled = CHANNEL_ENABLED;
        gConfig.y_accel_enabled = CHANNEL_ENABLED;
        gConfig.z_accel_enabled = CHANNEL_ENABLED;
        gConfig.x_gyro_enabled = CHANNEL_ENABLED;
        gConfig.y_gyro_enabled = CHANNEL_ENABLED;
        gConfig.z_gyro_enabled = CHANNEL_ENABLED;
        gConfig.pitch_enabled = CHANNEL_DISABLED;
        gConfig.roll_enabled = CHANNEL_DISABLED;
```

```
    gConfig.accel_variance = 5000;
    gConfig.process_variance = .01;

    gConfig.acc_vector[0] = 0;
    gConfig.acc_vector[1] = 0;
    gConfig.acc_vector[2] = -7270;

    // Broadcast mode or listen mode
    gConfig.broadcast_enabled = MODE_LISTEN; //MODE_BROADCAST; Ryan
    gConfig.broadcast_rate = 237;
}
else
{

    // Gyro and accelerometer default biases
    gConfig.x_accel_bias = GET_ACCEL_X_OFFSET();
    gConfig.y_accel_bias = GET_ACCEL_Y_OFFSET();
    gConfig.z_accel_bias = GET_ACCEL_Z_OFFSET();
    gConfig.x_gyro_bias = GET_GYRO_X_OFFSET();
    gConfig.y_gyro_bias = GET_GYRO_Y_OFFSET();
    gConfig.z_gyro_bias = GET_GYRO_Z_OFFSET();

    // Gyro and accelerometer default FIR corner frequencies
    gConfig.x_accel_corner = GET_ACCEL_X_CORNER();
    gConfig.y_accel_corner = GET_ACCEL_Y_CORNER();
    gConfig.z_accel_corner = GET_ACCEL_Z_CORNER();
    gConfig.x_gyro_corner = GET_GYRO_X_CORNER();
    gConfig.y_gyro_corner = GET_GYRO_Y_CORNER();
    gConfig.z_gyro_corner = GET_GYRO_Z_CORNER();

    // Gyro and accelerometer default FIR filter tap lengths
    gConfig.x_accel_taps = GET_ACCEL_X_TAPS();
    gConfig.y_accel_taps = GET_ACCEL_Y_TAPS();
    gConfig.z_accel_taps = GET_ACCEL_Z_TAPS();
    gConfig.x_gyro_taps = GET_GYRO_X_TAPS();
    gConfig.y_gyro_taps = GET_GYRO_Y_TAPS();
    gConfig.z_gyro_taps = GET_GYRO_Z_TAPS();

    // Channel enable/disable flags
    gConfig.x_accel_enabled = IS_ACCEL_X_ENABLED();
    gConfig.y_accel_enabled = IS_ACCEL_Y_ENABLED();
    gConfig.z_accel_enabled = IS_ACCEL_Z_ENABLED();
    gConfig.x_gyro_enabled = IS_GYRO_X_ENABLED();
    gConfig.y_gyro_enabled = IS_GYRO_Y_ENABLED();
    gConfig.z_gyro_enabled = IS_GYRO_Z_ENABLED();

    gConfig.pitch_enabled = IS_PITCH_ENABLED();
    gConfig.roll_enabled = IS_ROLL_ENABLED();

    gConfig.accel_variance = 5000;
    gConfig.process_variance = getProcessNoise();

    gConfig.acc_vector[0] = 0;
    gConfig.acc_vector[1] = 0;
```

```
        gConfig.acc_vector[2] = -7270;

        // Broadcast mode or listen mode
        gConfig.broadcast_enabled = GET_TRANSMIT_MODE();
        gConfig.broadcast_rate = GET_BROADCAST_RATE();
    }
}


/*****************************************************************
* Function Name  : ProcessPacket
* Input          : USARTPacket*
* Output         : None
* Return         : None
* Description    : Takes a packet received over the UART and processes
*                  it, calling the appropriate functions to act on the
*                  new data.
*****************************************************************/
void ProcessPacket( USARTPacket* new_packet )
{
    USARTPacket response_packet;
    int result;
    int temp;

    // Determine the packet type
    switch( new_packet->PT )
    {
// ---------------------------------------------------------------------
// SET_FIR_CORNERS packet - Sets the 3-dB corner frequency of each
// channel.
// Corner freq. = (x-1)*10 Hz.  If x = 0, or x = 1, then the FIR filter
// is disabled for that channel.
// ---------------------------------------------------------------------
   case SET_FIR_CORNERS:
      gConfig.x_accel_corner = (uint8_t)new_packet->packet_data[5];
      if( gConfig.x_accel_corner < 2 ) {
          gConfig.x_accel_corner = FIR_DISABLED;
      }

      gConfig.y_accel_corner = (uint8_t)new_packet->packet_data[4];
      if( gConfig.y_accel_corner < 2 ) {
          gConfig.y_accel_corner = FIR_DISABLED;
      }

      gConfig.z_accel_corner = (uint8_t)new_packet->packet_data[3];
      if( gConfig.z_accel_corner < 2 ) {
          gConfig.z_accel_corner = FIR_DISABLED;
      }

      gConfig.x_gyro_corner = (uint8_t)new_packet->packet_data[2];
      if( gConfig.x_gyro_corner < 2 ) {
          gConfig.x_gyro_corner = FIR_DISABLED;
      }
```

```
      gConfig.y_gyro_corner = (uint8_t)new_packet->packet_data[1];
      if( gConfig.y_gyro_corner < 2 ) {
          gConfig.y_gyro_corner = FIR_DISABLED;
      }

      gConfig.z_gyro_corner = (uint8_t)new_packet->packet_data[0];
      if( gConfig.z_gyro_corner < 2 ) {
          gConfig.z_gyro_corner = FIR_DISABLED;
      }

      SendCommandSuccessPacket( SET_FIR_CORNERS );
      break;

// ---------------------------------------------------------------------
// SET_FIR_TAPS packet - Sets the number of taps used in each FIR
// filter.
// 0 = 8 taps, 1, = 16 taps, 2 = 32 taps, 3 = 64 taps
// ---------------------------------------------------------------------
   case SET_FIR_TAPS:
      gConfig.x_accel_taps =(uint8_t)(new_packet->packet_data[1]&0x03);
      gConfig.y_accel_taps = (uint8_t)((new_packet-> packet_data[1] >>
         2) & 0x03);
      gConfig.z_accel_taps = (uint8_t)((new_packet->packet_data[1] >>
         4) & 0x03);
      gConfig.x_gyro_taps = (uint8_t)((new_packet->packet_data[1] >> 6)
         & 0x03);
      gConfig.y_gyro_taps = (uint8_t)((new_packet->packet_data[0])&
         0x03);
      gConfig.z_gyro_taps = (uint8_t)((new_packet->packet_data[0] >> 2)
         & 0x03);

      SendCommandSuccessPacket( SET_FIR_TAPS );
      break;

// ---------------------------------------------------------------------
// SET_ACTIVE_CHANNELS packet - Sets channels whose data will be
// transmitted
// when a GET_DATA packet is sent, or when Broadcast Mode is enabled
// ---------------------------------------------------------------------
   case SET_ACTIVE_CHANNELS:
      gConfig.x_accel_enabled = (uint8_t)( (new_packet->packet_data[0])
         & 0x01);
      gConfig.y_accel_enabled = (uint8_t)( (new_packet->packet_data[0]
         >> 1) & 0x01);
      gConfig.z_accel_enabled = (uint8_t)( (new_packet->packet_data[0]
         >> 2) & 0x01);
      gConfig.x_gyro_enabled = (uint8_t)( (new_packet->packet_data[0]
         >> 3) & 0x01);
      gConfig.y_gyro_enabled = (uint8_t)( (new_packet->packet_data[0]
         >> 4) & 0x01);
      gConfig.z_gyro_enabled = (uint8_t)( (new_packet->packet_data[0]
         >> 5) & 0x01);
      gConfig.roll_enabled = (uint8_t)( (new_packet->packet_data[0] >>
         6) & 0x01);
```

```
      gConfig.pitch_enabled = (uint8_t)( (new_packet->packet_data[0] >>
         7) & 0x01);

      SendCommandSuccessPacket( SET_ACTIVE_CHANNELS );

     break;

// ---------------------------------------------------------------------
// SET_SILENT_MODE packet - Turns off broadcast mode.
// ---------------------------------------------------------------------
   case SET_SILENT_MODE:
      DisableBroadcastMode( );

      SendCommandSuccessPacket( SET_SILENT_MODE );
      break;

// ---------------------------------------------------------------------
// SET_BROADCAST_MODE packet - turns on broadcast mode with the
// frequency specified in packet_data[0].
// ---------------------------------------------------------------------
   case SET_BROADCAST_MODE:
      EnableBroadcastMode( new_packet->packet_data[0] );

      SendCommandSuccessPacket( SET_BROADCAST_MODE );
      break;

// ---------------------------------------------------------------------
// SET_X_GYRO_BIAS packet - Sets bias term for x-axis rate gyro.
// ---------------------------------------------------------------------
   case SET_X_GYRO_BIAS:
      gConfig.x_gyro_bias = ((uint16_t)new_packet->packet_data[0] << 8)
      | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_X_GYRO_BIAS );
      break;

// ---------------------------------------------------------------------
// SET_Y_GYRO_BIAS packet - Sets bias term for y-axis rate gyro.
// ---------------------------------------------------------------------
   case SET_Y_GYRO_BIAS:
      gConfig.y_gyro_bias = ((uint16_t)new_packet->packet_data[0] << 8)
      | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_Y_GYRO_BIAS );
      break;

// ---------------------------------------------------------------------
// SET_Z_GYRO_BIAS packet - Sets bias term for z-axis rate gyro.
// ---------------------------------------------------------------------
   case SET_Z_GYRO_BIAS:
      gConfig.z_gyro_bias = ((uint16_t)new_packet->packet_data[0] << 8)
      | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_Z_GYRO_BIAS );
```

```
      break;

// ---------------------------------------------------------------------
// SET_X_ACCEL_BIAS packet - Sets bias term for x-axis accelerometer
// ---------------------------------------------------------------------
   case SET_X_ACCEL_BIAS:
      gConfig.x_accel_bias = (((uint16_t)new_packet->packet_data[0] <<
      8) & 0x0FF00) | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_X_ACCEL_BIAS );
       break;

// ---------------------------------------------------------------------
// SET_Y_ACCEL_BIAS packet - Sets bias term for y-axis accelerometer
// ---------------------------------------------------------------------
   case SET_Y_ACCEL_BIAS:
      gConfig.y_accel_bias = ((uint16_t)new_packet->packet_data[0] <<
      8) | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_Y_ACCEL_BIAS );
      break;

// ---------------------------------------------------------------------
// SET_Z_ACCEL_BIAS packet - Sets bias term for z-axis accelerometer
// ---------------------------------------------------------------------
   case SET_Z_ACCEL_BIAS:
      gConfig.z_accel_bias = ((uint16_t)new_packet->packet_data[0] <<
      8) | ((uint16_t)new_packet->packet_data[1] & 0x0FF);

      SendCommandSuccessPacket( SET_Z_ACCEL_BIAS );
      break;

// ---------------------------------------------------------------------
// ZERO_RATE_GYROS packet - starts gyro zero command
// ---------------------------------------------------------------------
   case ZERO_RATE_GYROS:
      StartGyroCalibration();

// No COMMAND_COMPLETE packet is sent from here.  When the gyro zeroing
// is finished, a GYRO_BIAS_REPORT packet is sent from within the
// filtering code.
      break;

// ---------------------------------------------------------------------
// SELF_TEST packet - starts automatic self-test command
// ---------------------------------------------------------------------
   case SELF_TEST:
      StartSelfTest();
      break;

// ---------------------------------------------------------------------
// SET_PROCESS_WEIGHT packet - used to set the diagonal elements in the
// measurement covariance matrix for the EKF.
// ---------------------------------------------------------------------
```

```
   case SET_PROCESS_WEIGHT:
      temp = (uint16_t)((new_packet->packet_data[0] << 8) | new_packet-
      >packet_data[1]);

      gConfig.process_variance = (float)temp*.0000762939;

      // Set process noise matrix
      gEstimatedStates.Q.data[0][0] = (double)gConfig.process_variance;
      gEstimatedStates.Q.data[1][1] = double)gConfig.process_variance;
      gEstimatedStates.Q.data[2][2] = (double)gConfig.process_variance;

      SendCommandSuccessPacket( SET_PROCESS_WEIGHT );
      break;

// ---------------------------------------------------------------------
// SET_ACCEL_REF_VECTOR packet - used to set the vector the IMU will
// use as a reference point for pitch = roll = 0.  The current
// orientation of the IMU is assumed to be perfectly parallel to the
// ground.
// ---------------------------------------------------------------------
   case SET_ACCEL_REF_VECTOR:
      gConfig.acc_vector[0] = gSensorData.accel_x;
      gConfig.acc_vector[1] = gSensorData.accel_y;
      gConfig.acc_vector[2] = gSensorData.accel_z;

      gAccVect.data[0] = gConfig.acc_vector[0];
      gAccVect.data[1] = gConfig.acc_vector[1];
      gAccVect.data[2] = gConfig.acc_vector[2];

      SendCommandSuccessPacket( SET_ACCEL_REF_VECTOR );
      break;

// ---------------------------------------------------------------------
// WRITE_TO_FLASH packet - saves all settings to flash.
// ---------------------------------------------------------------------
   case WRITE_TO_FLASH:

   // Disable the ADC temporarily - the WRITE_TO_FLASH command takes
   // longer to complete than it takes to fill an input buffer with
   // data.
      ADC_SoftwareStartConvCmd(ADC1, DISABLE);

   // If BROADCAST_MODE is enabled, turn it off while writing
   // configuration to flash
      if( gConfig.broadcast_enabled == MODE_BROADCAST )
      {
          // Disable Timer 2
          TIM_Cmd( TIM2, DISABLE );

          result = WriteConfigurationToFlash( );

          // Enable Timer 2
          TIM_Cmd( TIM2, ENABLE );
      }
```

```
      else
      {
          result = WriteConfigurationToFlash( );
      }

      ADC_SoftwareStartConvCmd(ADC1, ENABLE);

      if( result == FLASH_COMPLETE )
      {
          SendCommandSuccessPacket( WRITE_TO_FLASH );
      }
      else
      {
          SendCommandFailedPacket( WRITE_TO_FLASH, result );
      }

      break;

// ---------------------------------------------------------------------
// GET_DATA packet - ignored if IMU is in broadcast mode.  If in listen
// mode, return data from all active channels.
// ---------------------------------------------------------------------
   case GET_DATA:
      if( !gConfig.broadcast_enabled )
      {
          SendDataPacket();
      }
      break;

/ ---------------------------------------------------------------------
// GET_GYRO_BIAS packet - Sends a packet containing the bias values
// for each gyro channel.
// ---------------------------------------------------------------------
   case GET_GYRO_BIAS:
      SendGyroBiasPacket();
      break;

// ---------------------------------------------------------------------
// GET_ACCEL_BIAS packet - Sends a packet containing the bias values
// for each accelerometer annel.
// ---------------------------------------------------------------------
   case GET_ACCEL_BIAS:
      SendAccelBiasPacket();
      break;

// ---------------------------------------------------------------------
// GET_FIR_CONFIG packet - Sends a packet describing the corner
// frequency configuration for each individual channel's FIR filter
// ---------------------------------------------------------------------
   case GET_FIR_CONFIG:
      SendFIR_Packet();
      break;
```

```
// ---------------------------------------------------------------------
// GET_FIR_TAP_CONFIG packet - Returns a packet containing the number
// of taps used in each sensor's FIR filter.
// ---------------------------------------------------------------------
   case GET_FIR_TAP_CONFIG:
      SendFIR_TapPacket();
      break;

// ---------------------------------------------------------------------
// GET_ACTIVE_CHANNELS packet - Returns a packet specifying whether
// each sensor channel is active or inactive.
// ---------------------------------------------------------------------
   case GET_ACTIVE_CHANNELS:
      SendActiveChannelPacket();
      break;

   case GET_BROADCAST_MODE:
      SendTransmitModePacket();
      break;

   default:
   // The packet was not recognized.  Send an UNRECOGNIZED_PACKET
   // packet
      response_packet.PT = UNRECOGNIZED_PACKET;
      response_packet.length = 1;
      response_packet.packet_data[0] = new_packet->PT;
      response_packet.checksum = ComputeChecksum( &response_packet );

      SendTXPacketSafe( &response_packet );

      break;

// ---------------------------------------------------------------------
// GET_PROCESS_WEIGHT packet - Returns the value used in the diagonals
// of the accel measurement covariance weight
// ---------------------------------------------------------------------
   case GET_PROCESS_WEIGHT:
      SendProcessWeightPacket();
      break;

// ---------------------------------------------------------------------
// GET_LOCATION_COORDINATES packet - Returns the location coordinates
// (X,Y,Z)
//    added by Ryan
// ---------------------------------------------------------------------
   case GET_LOCATION_COORDINATES:
      SendLocationCoordinatesPacket();
      break;
   }
}
```

```
/*******************************************************************
* Function Name  : main
* Input          : None
* Output         : None
* Return         : None
* Description    : Entry point for CHR-6d firmware
*******************************************************************/
int main(void)
{
    unsigned int i;
    unsigned int j;
    int new_data;

// Initialize the IMU clocks, ADC, DMA controller, GPIO pins, etc.
// Initialize_IMU();

// Fill gConfig structure from flash, or use default values if flash
// has not been initialized.
    GetConfiguration();

// Pre-initialize FIR input buffer to zero
    for( i=0; i < CHANNEL_COUNT; i++ )
    {
        for( j=0; j < (MAX_INPUT_SIZE); j++ )
        {
            gFIR_Input[i][j] = 0;
        }
    }

// Start ADC1 Software Conversion.  ADC1 and ADC2 are configured to
// sample simultaneously, and to
// use DMA1 to transfer data to memory.
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);

// Start TIM1 if in broadcast mode
    if( gConfig.broadcast_enabled == MODE_BROADCAST )
    {
        EnableBroadcastMode( gConfig.broadcast_rate );
    }

// Initialize EKF covariances
// Set process noise matrix
    gEstimatedStates.Q.data[0][0] = (double)gConfig.process_variance;
    gEstimatedStates.Q.data[1][1] = (double)gConfig.process_variance;
    gEstimatedStates.Q.data[2][2] = (double)gConfig.process_variance;

// Set measurement noise matrix (this matrix can be determined
// experimentally)
    gEstimatedStates.Racc.data[0][0] = (double)gConfig.accel_variance;
    gEstimatedStates.Racc.data[1][1] = (double)gConfig.accel_variance;
    gEstimatedStates.Racc.data[2][2] = (double)gConfig.accel_variance;

    gAccVect.data[0] = gConfig.acc_vector[0];
    gAccVect.data[1] = gConfig.acc_vector[1];
```

```
    gAccVect.data[2] = gConfig.acc_vector[2];

// Start timer3 for prediction period tracking
    TIM_SetCounter(TIM3,0);
    TIM_Cmd(TIM3, ENABLE);

// Main program loop
    while(1)
    {
   // Filter new data if available (new_data = 0 if there was no new
   // data ready, 1 otherwise)
        new_data = process_input_buffers( gFIR_Input, gDecimatedOutput,
                        gFIR_Output );

        if( new_data )
        {
            // Retrieve data and convert to NED coordinate axes
            gSensorData.accel_x = -gFIR_Output[1];
            gSensorData.accel_y = gFIR_Output[0];
            gSensorData.accel_z = -gFIR_Output[2];

            gSensorData.gyro_x = gFIR_Output[4];
            gSensorData.gyro_y = gFIR_Output[3];
            gSensorData.gyro_z = -gFIR_Output[5];

            // next line modified by Ryan
            // EKF_EstimateStates( &gEstimatedStates, &gSensorData );
            // next line added by Ryan
            DeadReck_CalculateCoordinates( &gSensorData );
        }

        // Check if a packet has been received over the UART
        if( gRXPacketReceived )
        {
            ProcessPacket( (USARTPacket*)&
                        gRXPacketBuffer[gRXPacketBufferStart] );

         // Increment RX packet buffer pointer.  The RX and TX buffer
         // is circular.  The buffer is empty if the "start" and "end"
         // pointers point to the same location.
            gRXPacketBufferStart++;
            if( gRXPacketBufferStart >= RX_PACKET_BUFFER_SIZE )
            {
                   gRXPacketBufferStart = 0;
            }

         // If there are no more packets ready for processing, clear
         // the gRXPacketReceived flag. If there is another packet
         // ready, leave the flag set so that the other RX packet will
         // be processed the next time through the main loop.
            if( gRXPacketBufferStart == gRXPacketBufferEnd )
            {
                   gRXPacketReceived = 0;
            }
```

```
        }
    }
}


/*********************************************************************
* Function Name  : SendLocationCoordinatesPacket()
* Input          : None
* Output         : None
* Return         : None
* Description    : Sends a location coordinates packet over the UART
*                  transmitter.
* Added by Ryan
*********************************************************************/
void SendLocationCoordinatesPacket()
{
    USARTPacket NewPacket;
    long* LCptr;

    NewPacket.PT = LOCATION_COORDINATES_REPORT;
    NewPacket.length = 12;

    LCptr = (long*)(&LocationCoordinates.X);
    NewPacket.packet_data[0] = ((*LCptr >> 24) & 0x0FF);
    NewPacket.packet_data[1] = ((*LCptr >> 16) & 0x0FF);
    NewPacket.packet_data[2] = ((*LCptr >> 8) & 0x0FF);
    NewPacket.packet_data[3] = (*LCptr & 0x0FF);

    LCptr = (long*)(&LocationCoordinates.Y);
    NewPacket.packet_data[4] = ((*LCptr >> 24) & 0x0FF);
    NewPacket.packet_data[5] = ((*LCptr >> 16) & 0x0FF);
    NewPacket.packet_data[6] = ((*LCptr >> 8) & 0x0FF);
    NewPacket.packet_data[7] = (*LCptr & 0x0FF);

    LCptr = (long*)(&LocationCoordinates.Z);
    NewPacket.packet_data[8] = ((*LCptr >> 24) & 0x0FF);
    NewPacket.packet_data[9] = ((*LCptr >> 16) & 0x0FF);
    NewPacket.packet_data[10] = ((*LCptr >> 8) & 0x0FF);
    NewPacket.packet_data[11] = (*LCptr & 0x0FF);

    NewPacket.checksum = ComputeChecksum( &NewPacket );

    SendTXPacketSafe( &NewPacket );
                                    }
```

Appendix B – DeadReck Source Code Package

DeadReck.h

```
// Structure for holding location coordinates
typedef struct __LocationCoord {
    float X;
    float Y;
    float Z;
} LocationCoord;

extern LocationCoord LocationCoordinates;

   void DeadReck_CalculateCoordinates( RawSensorData* gSensorData );
```

DeadReck.C

```
#include <math.h>
#include "chr6d_states.h"
#include "DeadReck.h"

#define radsConverter (3.14159 / 180.0)
#define timeConverter 0.000001

LocationCoord LocationCoordinates;

/*********************************************************************
* Function Name  : DeadReck_CalculateCoordinates
* Input          : Sensor Data
* Output         : None
* Return         : None
* Description    : Reads IMU sensor values and calculates the location
*                  coordinates
*********************************************************************/
void DeadReck_CalculateCoordinates( RawSensorData* SensorData )
{
    // initialization variables
    static uint16_t initialized = 0;
    static uint16_t waitCount = 0;
    // calculated angle values
    static float roll = 0.0;  // in rads
    static float pitch = 0.0; // in rads
    static float yaw = 0.0;   // in rads
    // old anglular velocities
    static float oldQ = 0.0;   // in rads/second
    static float oldR = 0.0;   // in rads/second
    static float oldP = 0.0;   // in rads/second
    // old acceleration values (to calculate velocity)
    static float oldAccelU = 0.0; // in meters/second/second
    static float oldAccelV = 0.0; // in meters/second/second
    static float oldAccelW = 0.0; // in meters/second/second
```

```
// current free-body velocity values
static float u = 0.0; // in meters/second/second
static float v = 0.0; // in meters/second/second
static float w = 0.0; // in meters/second/second
// time variables
uint16_t timerValue = 0;
uint16_t waitTime = 0;
float time = 0;
// measured gyro values
float gyroQ = 0.0;   // in rads/second
float gyroR = 0.0;   // in rads/second
float gyroP = 0.0;   // in rads/second
// calculated gyro rates
float q = 0.0;   // in rads/second
float r = 0.0;   // in rads/second
float p = 0.0;   // in rads/second
// euler rates
float eulerRollRate = 0.0;   // in rads/second
float eulerPitchRate = 0.0;  // in rads/second
float eulerYawRate = 0.0;    // in rads/second
// measured accelerometer values
float aclrmtrU = 0.0; // in meters/second/second
float aclrmtrV = 0.0; // in meters/second/second
float aclrmtrW = 0.0; // in meters/second/second
// calculated acceleration values
float accelU = 0.0; // in meters/second/second
float accelV = 0.0; // in meters/second/second
float accelW = 0.0; // in meters/second/second
// earth-referenced frame velocities
float velX = 0.0; // in meters/second
float velY = 0.0; // in meters/second
float velZ = 0.0; // in meters/second

int intermediateResult = 0; // used for determining angle

if( initialized ) // Calculate location coordinates
{
    // get IMU measurements
    gyroP = (float)GYRO_SCALE_FACTOR*(float)SensorData->gyro_x *
            radsConverter;
    gyroQ = (float)GYRO_SCALE_FACTOR*(float)SensorData->gyro_y *
            radsConverter;
    gyroR = (float)GYRO_SCALE_FACTOR*(float)SensorData->gyro_z *
            radsConverter;

    aclrmtrU = (float)ACCEL_SCALE_FACTOR*(float)SensorData->
        accel_x;
    aclrmtrV = (float)ACCEL_SCALE_FACTOR*(float)SensorData->
        accel_y;
    aclrmtrW = (float)ACCEL_SCALE_FACTOR*(float)SensorData->
        accel_z;

    // get elapsed time (uS) since last sensor read
    TIM_Cmd(TIM3, DISABLE);
```

```
timerValue = TIM_GetCounter(TIM3);

// reset timer
TIM_SetCounter(TIM3,0);
TIM_Cmd(TIM3, ENABLE);
time = (float)timerValue * timeConverter; // time value to use

// calculate angular velocities (midpoint)
q = oldQ + (gyroQ - oldQ) / 2;
r = oldR + (gyroR - oldR) / 2;
p = oldP + (gyroP - oldP) / 2;
oldQ = gyroQ;
oldR = gyroR;
oldP = gyroP;

// calculate current euler rates
eulerYawRate   = (q * sin(roll) + r * cos(roll)) / cos(pitch);
eulerPitchRate = q * cos(roll) - r * sin(roll);
eulerRollRate  = p + (q * sin(roll) + r * cos(roll)) *
                tan(pitch);

// calculate current euler angles
yaw   += eulerYawRate * time;
roll  += eulerRollRate * time;
pitch += eulerPitchRate * time;

// calculate current accelerations (midpoint)
accelU = oldAccelU + (aclrmtrU - oldAccelU) / 2;
accelV = oldAccelV + (aclrmtrV - oldAccelV) / 2;
accelW = oldAccelW + (aclrmtrW - oldAccelW) / 2;
oldAccelU = aclrmtrU;
oldAccelV = aclrmtrV;
oldAccelW = aclrmtrW;

// calculate free-body velocities
u += accelU * time;
v += accelV * time;
w += accelW * time;

// calculate earth referenced velocities
velX = (u * cos(pitch) + (v * sin(roll) + w * cos(roll)) *
        sin (pitch )) * cos(yaw) - (v * cos(roll) - w *
        sin(roll)) * sin(yaw);
velY = (u * cos(pitch) + (v * sin(roll) + w * cos(roll)) *
        sin (pitch )) * sin(yaw) + (v * cos(roll) - w *
        sin(roll)) * cos(yaw);
velZ = -1 * u * sin(pitch) + (v * sin(roll) + w *
        cos(roll)) * cos(pitch);

// remove gravity from Z
velZ = velZ - 9.7935 * time; // gravity in Austin, Tx

// calculate 3-D displacements
LocationCoordinates.X += velX * time;
```

```
    LocationCoordinates.Y += velY * time;
    LocationCoordinates.Z += -1 * velZ * time; // change NED to NEU

}
else    // wait for IMU to stabilize
{
    if( waitCount < 20 ) // wait is 20 * 50,000 uS = 1S
    {
        if( TIM_GetCounter(TIM3) > 50000 )
        {
            // reset timer
            TIM_Cmd(TIM3, DISABLE);
            TIM_SetCounter(TIM3,0 );
            TIM_Cmd(TIM3, ENABLE);

            // increment wait counter
            waitCount += 1;
        }
    }
    else
    {
        // calibrate gyros
        StartGyroCalibration();

        // set initialized flag
        initialized = 1;

        // reset timer
        TIM_Cmd( TIM3, DISABLE );
        TIM_SetCounter( TIM3,0 );
        TIM_Cmd( TIM3, ENABLE );
    }
}
                                }
```

Appendix C – Dead Reckoning Host Software Source Code

```
#include <rs232.h>
#include <userint.h>
#include <utility.h>
#include "MainPanel.h"

#define COMPORT 2


/*****************************************************************
* Function Name  : GetChecksum( unsigned char* message, short length )
* Input          : Message and message length
* Output         : None
* Return         : Calculated checksum
* Description    : Returns the calcuated checksum from a given message
* Added by Ryan
*****************************************************************/
unsigned short GetChecksum( unsigned char* message, short length )
{
  unsigned short checkSum = 0;
  unsigned short index = 0;

  for( index = 0; index < length; index++ )
  {
    checkSum += message[index];
  }

  return checkSum;
}


/*****************************************************************
* Function Name  : MainPanel_CB( int panel, int event,
*                  void* callbackData, int eventData1, int eventData2 )
* Input          : Standard CVI callback information
* Output         : None
* Return         : None
* Description    : Event handler for the mainpanel which closes the
*                  application
* Added by Ryan
*****************************************************************/
int CVICALLBACK MainPanel_CB( int panel, int event, void* callbackData,
                                    int eventData1, int eventData2 )
{
      if( event == EVENT_CLOSE )
      {
            QuitUserInterface( 0 );
      }
      return 0;
}
```

```
/*****************************************************************
* Function Name  : GetCoordinates(int panel, int control, int event,
*                   void *callbackData, int eventData1, int eventData2)
* Input          : Standard CVI callback information
* Output         : None
* Return         : None
* Description    : Event handler for the "Get Coordinates" button
* Added by Ryan
*****************************************************************/
int  CVICALLBACK GetCoordinates(int panel, int control, int event,
                void *callbackData, int eventData1, int eventData2)
{
  // command to send
  char  command[ 7 ] = { 's', 'n', 'p', 0x09, 0, 0, 0 };
  unsigned char  response[ 19 ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 0, 0 };
  int error = 0;
  float tempFloat;
  int* tempFloatPtr = (int*)( &tempFloat );
  unsigned short checksum = 0;
  unsigned short newChecksum = 0;

  if( event == EVENT_COMMIT )
  {
    FlushInQ( COMPORT );
    FlushOutQ( COMPORT );

    checksum = GetChecksum( command, 5 );
    command[5] = ( checksum >> 8 ) & 0x0FF;
    command[6] = checksum & 0x0FF;

    error = ( ComWrt( COMPORT, command, 7 ) != 7 );
    if( !error )
    {
      error = ComRd( COMPORT, response, 19 ) != 19;
    }
    *tempFloatPtr = 0;
    if( !error )
    {
      checksum = GetChecksum( response, 17 );
      newChecksum = response[17];
      newChecksum = (newChecksum << 8) | response[18];

      if( checksum == newChecksum )
      {
        *tempFloatPtr = response[5];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[6];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[7];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[8];
        SetCtrlVal( panel, PANEL_X_COORD, tempFloat );

        *tempFloatPtr = response[9];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[10];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[11];
```

```
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[12];
        SetCtrlVal( panel, PANEL_Y_COORD, tempFloat );

        *tempFloatPtr = response[13];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[14];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[15];
        *tempFloatPtr = ( *tempFloatPtr << 8 ) | response[16];
        SetCtrlVal( panel, PANEL_Z_COORD, tempFloat );
      }
    }
  }

  return 0;
}


/*********************************************************************
* Function Name  : SerialPort_Init()
* Input          : None
* Output         : None
* Return         : None
* Description    : Initializes the serial port
* Added by Ryan
*********************************************************************/
void SerialPort_Init()
{
  OpenComConfig( COMPORT,"COM2", 115200, 0, 8, 1, 2500, 2500 );
  SetComTime( COMPORT, 0.25 );  // set timeout
  FlushInQ( COMPORT );    // clean input buffer
  FlushOutQ( COMPORT );  // clean output buffer
}


/*********************************************************************
* Function Name  : SerialPort_Close()
* Input          : None
* Output         : None
* Return         : None
* Description    : Closes the serial port
* Added by Ryan
*********************************************************************/
void SerialPort_Close()
{
      CloseCom( COMPORT );
}
```

```
/*****************************************************************
* Function Name  : main (int argc, char *argv[] )
* Input          : None
* Output         : None
* Return         : None
* Description    : The main exectuable in this application
* Added by Ryan
*****************************************************************/
int main (int argc, char *argv[] )
{
  int thereIsAnother;
  int panel; // main panel handle

  if(InitCVIRTE (0, argv, 0) == 0)
    return -1; // out of memory
  if( CheckForDuplicateAppInstance( ACTIVATE_OTHER_INSTANCE,
                                    &thereIsAnother ) < 0 )
    return -1; // could not interface with O/S
  if( thereIsAnother )
    return 0; // prevent duplicate instance

  // initialize the serial port
  SerialPort_Init();

  // load and display host panel
  panel= LoadPanel( 0, "MainPanel.uir", PANEL );
  DisplayPanel( panel );

  // wait for user callbacks or exit
  RunUserInterface();

  // shutdown program
  SerialPort_Close();
  DiscardPanel(panel);

  return 0;
}
```

Bibliography

[1] D. Wence and K. Scott. “Inertial Waypoint Finder.” U.S. Patent 7 522 999, Apr. 21, 2009.

[2] H. Masudaya. “Apparatus for Search for Sensed Object.” U.S. Patent 5 612 688, Mar. 18, 1997.

[3] R. Wells, “Inertial/Magnetic Measurement Device.” U.S. Patent 7 587 277, Sep. 8, 2009.

[4] R.M. Wells, “Inertial Sensor Tracking System.” U.S. Patent 7 400 246, Jul. 15, 2008.

[5] Z. Berman and J.D. Powell. “The Role of Dead Reckoning and Inertial Sensors in Future General Aviation Navigation” in *Proceedings of IEEE Position Location and Navigation Symposium*, 1998, pp. 510-517.

[6] A.K. Brown and Y. Lu. “Performance Test Results of an Integrated GPS/MEMS Inertial Navigation Package,” in *Proceedings of ION GNSS 2004*, Sep. 2004, pp.

[7] H.Chung and J. Borenstein. “Accurate Mobile Robot Dead-Reckoning with a Precision-Calibrated Fiber Optic Gyroscope.” *IEEE Transactions on Robotics and Automation*, vol. 17, Feb. 2001, pp. 80-84.

[8] Y. Fuke and E. Krotkov. “Dead Reckoning for a Lunar Rover on Uneven Terrain,” in *Proceeding of the 1996 IEEE International Conference on Robotics and Automation,* 1996, pp. 411-416.

[9] C.T. Judd. “A Personal Dead Reckoning Module.” Presented at Institute of Navigation’s ION GPS ’97, Kansas City, Missouri, Sep. 1997.

[10] C. Randell, C. Djiallis, and H. Muller. “Personal Position Measurement Using Dead Reckoning,” in *Proceedings of the Seventh International Symposium on Wearable Computers*, 2003, pp. 166-173.

[11] D. Niculescu and B. Nath. “Ad Hoc Positioning System,” in *IEEE GlobeCom*, November, 2001.

[12] N.B. Priyantha, A. Chakroborty, and H. Balakrishnan. “The Cricket Location-Support System,” in *the 6th Annual International Conference on Mobile Computing and Networking Proceedings*, 2000, pp. 32-43.

[13] R. Stoleru, T. He, J, Stankovic, and D Luebke. “A High-Accuracy, Low-Cost Localization System for Wireless Sensor Networks,” in *SESSION: Sensornet Services*, 2005, pp. 13-26.

[14] CH Robotics. “CHR-6d Digital Inertial Measurement Unit Product Datasheet Rev. 1.4.” Internet: http://www.chrobotics.com/docs/chr6d_datasheet.pdf, February 26, 2010 [March 3, 2010].

[15] Future Technology Devices International Ltd. “TTL-232R TTL to USB Serial Converter Range of Cables Datasheet.” Internet: http://www.ftdichip.com/Documents/DataSheets/Modules/DS_TTL-232R_CABLES_V201.pdf, August 28, 2008 [March 3, 2010].

[16] Pololu Robotics & Electronics. “Bodhilabs VPack Boost Regulator – Rectangular.” Internet: http://www.pololu.com/catalog/product/795, [March 3, 2010].

[17] CH Robotics. “CHR-6d Resources.” Internet: http://www.chrobotics.com/downloads.php, [March 3, 2010].

[18] M. Rauw. “FDC 1.2 – A SIMULINK Toolbox for Flight Dynamics and Control Analysis.” Internet: http://home.wanadoo.nl/dutchroll/manual.html, September 15, 2005 [March 21, 2010].

Vita

Ryan Allen Brown, the son of Rodney Brown and Lois Mahoney, was born in Angeles City, Philippines on June 3, 1977. After completing his work at Westwood High School, Austin, Texas in 1995, he entered the U.S. Army as a Radio and Communications Systems Repairer. He was honorably discharged from the Army in 1999 and began pursuing his bachelor's degree in Electrical and Computer Engineering while working as an electronics technician. In 2005, he graduated from the University of Texas and began working at 5-D Systems as a Systems/Software Engineer for unmanned aerial systems. After working three years as a Systems/Software Engineer, he returned to the University of Texas to pursue his master's degree in Software Engineering. Ryan Brown is married to Elizabeth Brown, and they have two children: Ivy and Isaac.

Permanent Address: 10231 Missel Thrush Dr.
Austin, Tx 78750

This report was typed by the author.