TELEPRESENCE: DESIGN, IMPLEMENTATION AND STUDY OF
AN HMD-CONTROLLED AVATAR WITH A
MECHATRONIC APPROACH

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

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

by
Darren Michael Chan

June 2015

© 2015
Darren Michael Chan
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Telepresence: Design, Implementation and Study of
an HMD-controlled Avatar with a Mechatronic
Approach

AUTHOR: Darren Michael Chan

DATE SUBMITTED: June 2015

COMMITTEE CHAIR: Jane Zhang, Ph.D.
Professor of Electrical Engineering

COMMITTEE MEMBER: Bryan Mealy, Ph.D.
Associate Professor of Electrical Engineering

COMMITTEE MEMBER: John Ridgely, Ph.D.
Professor of Mechanical Engineering

ABSTRACT

Telepresence: Design, Implementation and Study of an HMD-controlled Avatar with a Mechatronic Approach

Darren Michael Chan

Telepresence describes technologies that allow users to remotely experience the sensation of being present at an event without being physically present. An avatar exists to represent the user whilst in a remote location and is tasked to collect stimuli from its immediate surroundings to be delivered to the user for consumption. With the advent of recent developments in Virtual Reality technology, viz., head-mounted displays (HMDs), new possibilities have been enabled in the field of Telepresence.

The main focus of this thesis is to develop a solution for visual Telepresence, where an HMD is used to control the direction of a camera's viewpoint, such that the user's head is tracked by the avatar, while providing visual feedback to the user. The design and development of the device follows a mechatronic approach, where a real time operating system (RTOS) is used in conjunction with a microcontroller for mechanical actuator control.

The first-generation prototype, *HOG-1* (**H**MD-**O**perated **G**imbal, rev. 1), developed for this thesis serves as a foundation for study; the implementation and analysis of the prototype contributes to the state of the art by providing a clearer glimpse of hardware and software requirements that are necessary to construct an improved model. Additionally, qualitative and quantitative measurements are developed in the process of this research.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
1 INTRODUCTION	1
1.1 Motivation.....	2
1.2 Recent Developments in Related Technology.....	3
1.3 Objectives	4
1.4 Thesis Overview	5
2 BACKGROUND	6
2.1 Mechatronics.....	6
2.2 Telepresence	6
2.3 Avatar	8
2.4 The YPR Coordinate System.....	10
3 PROTOTYPE CONCEPT	11
3.1 High Level Diagram and Description.....	12
3.2 Constraints	12
4 HARDWARE SELECTION AND DESIGN	13
4.1 HMD Selection	13
4.2 Camera Selection	14
4.3 Actuator Selection	15
4.4 Gimbal Frame Selection	18
4.5 Sensor Selection.....	19
4.5.1 Inertial Measurement Unit	19
4.5.2 Position Encoder (Absolute) Selection	20
4.6 Motor Drivers	26
4.7 Microcontroller Selection and Board Design	29
4.8 Prototype Integration	32

5.	SOFTWARE DESIGN AND IMPLEMENTATION	35
5.1	PC Client.....	35
5.1.1	Dependencies	35
5.1.2	HMD Data Acquisition	37
5.1.3	Data Formatting and Serial Protocol Settings.....	38
5.1.4	Client Algorithms.....	42
5.1.5	Vision Processor	44
5.2	Controller Design.....	49
5.2.1	Real Time Operating System	49
5.2.2	Controller Tasks, Timing and Priority	52
5.2.3	The Read HMD and Read IMU Tasks.....	55
5.2.3.1	Data Parsing	55
5.2.3.2	Reading from the Absolute Encoder	59
5.2.4	The Motor Controller Task	63
5.2.4.1	Microcontroller Setup	64
5.2.4.2	Motor Controller Theory	67
5.2.4.3	Motor Controller Implementation	80
6	RESULTS, ANALYSIS AND CONCLUSIONS.....	87
6.1	Device Complications.....	87
6.2	Quantitative Analysis.....	88
6.2.1	Analysis of Yaw-axis Tracking and Error Allowance	92
6.2.2	Analysis of Pitch-axis Tracking.....	96
6.2.3	Analysis of Roll-axis Tracking	98
6.3	Qualitative Study	100
6.4	Viability as a Mobile Device	102
7	FUTURE WORK.....	103
7.1	Summary of Improvements for a Second Iteration.....	103
7.2	An Improved Testing Method	105
7.3	Other Considerations	106
7.4	Final Remarks	107

BIBLIOGRAPHY	108
--------------------	-----

APPENDICES

A: Full Schematic of Controller Board.....	110
B: PC Client C++ Code	111
C: Vision Processor Python 2.7 Code.....	113
D: Embedded Code Documentation	114
E: Embedded C++ Code (HOG-1)	140

LIST OF TABLES

Table	Page
Table 1. Voltage Requirements for each of the Avatar Components	33
Table 2. Common baud rates and the corresponding transmission times for 220 bit-length data sets – It was experimentally determined that a baud rate of 57600 <i>bps</i> produced the best performance.	41
Table 3. Grey Code vs. Binary – Decimal values shown with equivalent Grey Code and Binary values.....	60
Table 4. ATmega2560 Pins vs. Parameters – Table of the microcontroller pins shown with their corresponding parameters. [2].....	66
Table 5. Yaw Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Yaw axis.	95
Table 6. Pitch Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Pitch axis.....	97
Table 7. Roll Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Roll axis.....	100

LIST OF FIGURES

Figure	Page
Figure 1. Differences between Virtual Reality and Telepresence – In Virtual Reality, the environment is generated and updated as the avatar interacts with the environment. In Telepresence, the environment is purely physical, where the avatar directly interacts with it.	9
Figure 2. The Yaw Pitch Roll Coordinate System [12] – Illustration of the Yaw, Pitch and Roll coordinate system.....	10
Figure 3. High-level Telepresence Concept Diagram – A high level diagram of the visual Telepresence system showing the relationship between high level components.	12
Figure 4. HMD [1] – The Oculus Rift DK2 head-mounted display.	13
Figure 5. Excitation Table vs. Motor Windings – Tabulated (simplified) excitation sequence of 3-phase DC motor for clockwise rotation and its corresponding winding sequence (right).....	16
Figure 6. The Three-axis Mechanical Gimbal – This diagram shows the critical components. A webcam and IMU (not fully visible) are loaded to the gimbal.....	18
Figure 7. A 3-bit Absolute Encoder Disk – Dark shaded regions represents a low signal, and white areas represent a high signal. An n th-bit resolution encoder with have an n th number of rings. One slice represents one measured position of the encoder.	22
Figure 8. KOYO TRD-NA256NW Datasheet Part 1[8] – The encoder is specified to have 8-bit resolution, or 256 possible grey code values for one revolution of the encoder shaft. The encoder is also designed for NPN output.	23
Figure 9. KOYO TRD-NA256NW Datasheet Part 2 [8] – Electrical specifications for the KOYO TRD-NA series encoders and connection diagram table for models with various resolutions. Specs highlighted in yellow correspond to specifications relating to the TRD-NA256NW model.....	24
Figure 10. The Mounted Yaw Axis Motor – The Yaw axis motor and rotary encoder are mounted to the base plate, and are stationary relative to the rest of the mechanical gimbal. The Yaw motor and encoder shafts are connected via flexible shaft coupler.....	25

Figure 11. Encoder to Microcontroller Interface [8] – Circuit schematic of the 1K-ohm pull-up resistor bank for NPN output interface to a microcontroller.	25
Figure 12. Encoder to Microcontroller Interface (Physical Implementation).....	26
Figure 13. SN754410 Quadruple Half H-Bridge Driver Specifications – Critical specifications are highlighted in yellow. This driver meets the power requirements of the motors discussed in Section 4.3.....	27
Figure 14. SN754410 Quadruple Half H-Bridge Driver Circuit – PWM1, PWM2 and PWM3 ports are wired to the timer output pins of the microcontroller. L1, L2 and L3 represent each of the phase windings for a 3-phase DC motor.	28
Figure 15. Schematic of Controller Board drawn using Eagle PCB design suite. A larger version of this schematic can be found in Appendix A.....	30
Figure 16. PCB Layout – Top view of the PCB layout of the controller board, designed in Eagle.	30
Figure 17. Controller Board (Physical Implementation) – Physical implementation of the controller board, with references to critical components.....	31
Figure 18. The HOG-1 Prototype – The fully assembled HOG-1 prototype, showing critical components.....	32
Figure 19. Overview of Connected Components that Form the HOG-1 – Orange colored blocks correspond to components that are mounted on the avatar, while yellow blocks correspond to components that the user directly interacts with.....	34
Figure 20. Terminal Window of HMD Test Program – A test program that extracts the YPR data from the Oculus Rift HMD and outputs the data to a terminal window.....	37
Figure 21. A High level Diagram of the PC Client – The routine inside the <i>while(True)</i> container is looped while the program runs, and if an available serial port is open.....	42
Figure 22. Example of a Stereoscopic Image in SBS Configuration – For a small IPD, the differences in the two halves are not extremely apparent. However, differences are more noticeable by comparing the corners of the images. (from Wikimedia commons) [7]	45
Figure 23. Process for Converting an Image to SBS Format – The Region of Interest (ROI) is segmented from the original image and projected to each half of the HMD screen (SBS-Configured). Since the ROI is identical for each half of the SBS-configured image, the result produces non-stereoscopic viewing.....	46

Figure 24. Block Diagram of the Vision Processor – The routine inside the <i>while(True)</i> container is looped while the program runs, and if the webcam is available.	47
Figure 25. Results of the Vision Processor – An image captured by the Logitech C920 webcam and produced by the Vision Processor.....	49
Figure 26. Task Timing Example – Task timing diagram of 3 tasks, each with 2ms timing requirements in two execution cycles, scheduled by an RTOS. The solid black lines indicate the task timing requirements	50
Figure 27. Task diagram of the HOG-1 Controller – Shared variables of tasks are connected by dotted lines.....	52
Figure 28. State diagram for Read HMD task.	56
Figure 29. State diagram for Read IMU task.....	56
Figure 30. Example of grey code to binary conversion.	60
Figure 31. Encoder Positions Mapped to LUT – This diagram depicts 7 positions out of 256 possible positions.....	63
Figure 32. 3-Phase Sine Wave Generation via PWM – Three sets of sine waves generated by PWM signals, captured on an oscilloscope. Each signal is spaced 120 degrees out of phase to create 3-phase DC voltage, which are used to power the 3-phase DC motors.	69
Figure 33. Schematic of a Primitive DAC – A simple RC low-pass filter created to act as a DAC.	70
Figure 34. 3-Phase Sine Wave Generation via PWM (Filtered) – Three sets of sine waves generated by PWM signals, captured on an oscilloscope. Each signal is spaced 120 degrees out of phase to create 3-phase DC voltage, which are used to power the 3-phase DC motors.....	70
Figure 35. PID Controller Diagram – The PID control system, showing the HOG-1 components.	78
Figure 36. The Motor Controller Task State Diagram for Yaw Axis Motor Control – For the complete system, this algorithm is replicated for the Pitch and Roll axis motors.	85
Figure 37. Yaw Axis Tracking: 15-30-15 Test Results.....	90
Figure 38. Pitch Axis Tracking: 15-30-15 Test Results.	91
Figure 39. Roll Axis Tracking: 15-30-15 Test Results.....	91

Figure 40. Yaw Axis Tracking: Slow Mode	92
Figure 41. Yaw Axis Tracking: Normal Operation	94
Figure 42. Yaw Axis Tracking: Fast Mode	94
Figure 43. Slow Mode Test Results for Tracking of the Pitch Axis.....	96
Figure 44. Normal Operation test results for tracking of the Pitch axis.	96
Figure 45. Fast Mode test results for tracking of the Pitch axis.	97
Figure 46. Slow Mode test results for tracking of the Roll axis.	98
Figure 47. Slow Operation test results for tracking of the Roll axis.....	99
Figure 48. Fast Mode test results for tracking of the Roll axis.....	99
Figure 49. Diagram for an Improve Test Method – High level diagram of the proposed testing procedure to be implemented in the Motor Controller task.....	106

1 INTRODUCTION

Developments in electronic displays and vision systems continue to push the boundaries of communication, medicine, military, entertainment and consumer devices. Within the past decade, lesser charted areas of research, viz., Virtual Reality (VR), Augmented Reality (AR) and Telepresence have gained attention along-side the advancement of digital video and graphics systems. Recent innovations by Sony and Oculus VR include VR Head Mounted Displays (HMDs) available to consumers at marketable prices. With the advent of Virtual Reality technology in full development, it follows that closely related fields, viz. Augmented Reality and Telepresence also benefit.

The field of discussion in this thesis revolves around Telepresence; Telepresence describes technologies that allow humans to gain the sensation of presence at an event without being physically present. In other words, it describes the means by which a user can experience the sensation of presence, remotely. Though Telepresence extends to visual, auditory and other experiences relating to the human senses, this thesis introduces a novel implementation of a Telepresence device that revolves around visual experience, or *Visual Telepresence*.

1.1 MOTIVATION

Explosive Ordinance Disposal (EOD) plays a significant role in saving military personnel and civilian lives. The men and women who operate in EOD teams to remove dangerous explosive devices regularly take life-threatening risks. Some special devices have been developed to lower the risks of severe injury and/or death due to accidental or unintended explosive discharge. Some examples of these counter-explosive devices include: bomb suits, containment chambers, disruption systems and small charges. However, all of these systems require the disarmer(s) to be within close proximity of the explosive, which correlates to missions with high-cost risks. A more recently developed EOD technique uses remotely controlled robots to closely observe the bomb site in place of live subjects, thus mitigating the risk of losing human lives. However, due to the complex nature of bomb disposal, EOD robots do not generally offer a complete disarming solution, because current systems lack precision and intuitive controls to make them very effective. In current implementations, the robot is controlled using a portable computing device, where controls are less than ideal because all of the robot's onboard devices (i.e. camera, arms, drivetrain, etc.) are controlled by a single keyboard or touchscreen, and perhaps with the addition of a joystick. The robot contains more devices to be controlled than the operator can control at one given time, since the operator only has two hands to work with. The integration of a Telepresence device, such as one that controls the direction that a camera is facing, by tracking the operator's head movements, allows for a more complete control interface for the user because he is no longer required to use his hands to control the camera direction.

1.2 RECENT DEVELOPMENTS IN RELATED TECHNOLOGY

First Person View (FPV) is a method that describes the piloting of unmanned remote controlled vehicles from a first person perspective. In a typical configuration, the operator receives visual feedback via display panel that is retrieved from a camera mounted on the vehicle. The operator is given a sense of position, direction and speed of the vehicle as it is being piloted in real time. In more advanced setups, stereo camera rigs are mounted on FPV vehicles to create the sensation of Telepresence for the operator. However, the camera's orientation is typically stable where the camera's direction can only be controlled by turning the entire FPV vehicle.

Omnidirectional (or 360-degree) cameras, made popular by Google's street view, have also garnered attention in the consumer market. These devices "stitch" together multiple, synchronized camera streams from different viewpoints to create a single 360-degree field of view (FOV) panoramic video feed. Omnidirectional cameras are often used for high FOV applications, e.g., panoramic photography, robotic vision and security systems. In FPV applications, 360-degree cameras offer a software-based solution for camera viewpoint control, rather than employing mechanical gimbals. One disadvantage of this device is that due to its extremely large resolution, it requires an enormous amount of processing to produce frame rates suitable for Telepresence applications – at the time of writing, no such device is able to produce more than 15 frames per second (FPS). But as technology progresses, this will become less of a problem. However, a major drawback to the omnidirectional camera is that it is impossible to implement stereo vision; stereo vision integrates two camera modules that must be set apart by some interpupillary distance (IPD) that allow the viewer to simultaneously view different angles for each eye.

Telepresence Robots [9] are also making their way into the consumer marketplace, made popular by companies like Double Robotics. These Telepresence robots feature a camera and moveable platform that can be remotely controlled through WIFI. Although the Telepresence Robot provides an alternative solution to teleconferencing, it does not create a sense of presence for which this thesis attempts to solve.

Perhaps the most similar piece of technology that relates to the work highlighted in this thesis (that is commonly known to exist), is the integration of Helmet Mounted Displays in military aircraft control systems [4]. Helmet Mounted Displays serve a similar purpose to that of VR HMDs, in that they both provide head-tracking capabilities and output visual data to the user. When used in fighter aircraft, the Helmet Mounted Display allows the user to aim his weapon using head movements. This technology shares the likeness of the methods described in this thesis, because mechanical actuators are controlled in real time via head-tracking. However, although similar, the two are vastly different in their application.

1.3 OBJECTIVES

The goal of this thesis is to contribute to the state of the art by developing an electrical-mechanical device (avatar), using mechatronic methodologies, that serves as a platform for visual Telepresence. With the implementation of this device, important characteristics and complications related to mechanical head-tracking will surface to allow for further study, and an improved second-iteration prototype can be proposed.

1.4 THESIS OVERVIEW

This chapter introduces concepts, motivation and goals associated with this thesis. Chapter 2 presents terms and definitions which will be used in this paper, and covers theory and concepts in the fields of Telepresence, Virtual Reality and Augmented Reality. Chapter 3 gives insight to current generation hardware (at the time of writing) that will be used for this project. Chapter 5 discusses design and implementation of software using mechatronic methodologies. Chapter 6 discusses performance, results and proposes improvements for a second prototype. Chapter 7 concludes this thesis and prompts future related work.

2 BACKGROUND

This chapter introduces terms and definitions specific to the work in this thesis, and provides introductory background knowledge related to Telepresence. In addition, this chapter also briefly explains some fundamental concepts related to prototype design, which will be used to explain more advanced concepts in future sections.

2.1 MECHATRONICS

Mechatronics is an interdisciplinary field that extends its reaches to Electrical Engineering, Mechanical Engineering and Computer Science. While there is no formal definition for the term, a mechatronic device typically integrates a computer and electronics for mechanical actuator control. Hence, the design of a mechatronic system encompasses both hardware and software design, where both parts have an equal emphasis. However, a unique approach, that is specific to the discipline, is that software design is organized into abstractions known as tasks and states [3]; a task is a fundamental building block that is executed by a real time operating system (RTOS) that is natively installed to the computing device. The concept of tasks will be explained in further detail in Chapter 5, where the implementation of software will be used as an example.

2.2 TELEPRESENCE

The concept of Telepresence and the technology surrounding this subject has long existed before modern developments in VR technology. For example, the telephone, invented in the 1800's has allowed two (or more) remotely placed users to communicate speech effectively in real time. Another example is the television, invented in the early

1900's, which has allowed humans to experience remote events (albeit not always in real time) concerning real people, places and things from the comfort of their homes. More recently, the introduction of the internet has made it possible for teleconferencing, wherein participants can transfer and receive speech and visual information simultaneously in real time. The underlying principle behind these innovations is that it has enabled reliable and fast communication between distant parties. In an ideal world, humans would have the capability to remotely extract as much information as possible, in the shortest amount of time. This is essentially the problem that Telepresence attempts to solve.

Similar to Virtual Reality, Telepresence revolves around an abstraction known as *presence*. Presence simply describes the degree by which a user of a Telepresence/VR device feels connected to his environment (whether the environment is simulated or not), as if actually present at the event. The nature of presence relates to how effectively the user can draw realistic experiences from his environment by means of the human senses, and in theory, will greatly improve remote communication.

Achieving presence, however, is a very complicated problem because naturally occurring physical human behavior is required to be captured as inputs to a computer, while the computer must simultaneously deliver outputs (that is recognizable by the human senses) to the user by means of devices that are typically electrical and/or mechanical.

2.3 AVATAR

An avatar is a physical or nonphysical representation of a user that directly interacts with an environment that is not immediate to the user. For example, in Virtual Reality, an environment is simulated by a computer; the user is able to directly interact with the computer, but is unable to directly interact with the simulated environment – instead, an avatar represents the user in that simulated environment and is, by proxy, controlled by the user.

In Telepresence, the environment and avatar have physical forms, where the avatar is remotely controlled by the user. An avatar within the context of Telepresence is also tasked with collecting information about its surroundings, in addition to being remotely controlled. The information is then delivered to the user in a form of output that pertains to the human senses. All of these responsibilities must be performed in real time – and for a high performance system, in such a way that the user does not detect a delayed response. Figure 1 highlights the differences between avatars used in Virtual Reality and Telepresence systems.

Figure 1. Differences between Virtual Reality and Telepresence – In Virtual Reality, the environment is generated and updated as the avatar interacts with the environment. In Telepresence, the environment is purely physical, where the avatar directly interacts with it.

The Telepresence and Virtual Reality system models have many similarities; both require the user to operate a Virtual Reality device and a computer where the avatar responds to human input. However, in Virtual Reality, the environment is simulated, such that when the avatar interacts with it, the computer must regenerate an updated model of that environment (since the environment changes when the avatar interacts with it). In Telepresence, the avatar is physical, and so any interaction that it has with its environment is real, such that the environment actually changes.

2.4 THE YPR COORDINATE SYSTEM

The Yaw, Pitch and Roll (YPR) coordinate system is heavily used in aircraft control systems, IMUs and HMDs. Hence, understanding this arrangement is pertinent to the prototype that is designed as part of this thesis. Rather than attempting to verbally describe the coordinate system, it is far easier to understand with a visual diagram (shown in Figure 2).

Figure 2. The Yaw Pitch Roll Coordinate System [12] – Illustration of the Yaw, Pitch and Roll coordinate system.

3 PROTOTYPE CONCEPT

This section describes the prototype at the conceptual level, as the basis for preliminary design. No hardware or software specifications are mentioned here, but rather the prototype device is described by how it should operate. Generalized, high level components are selected to model the workings of the device, and a high level diagram is presented at the end of this section to show the relationship between those parts. The entirety of the system is devised of two main components:

- An HMD serves to collect the user's head rotation coordinates and provide visual feedback from a camera (mounted to the avatar) via the HMD display.
- The electrical-mechanical avatar is mounted with a camera that collects a video feed of its surroundings and delivers the footage to the HMD, which can be viewed in real time. When the HMD is moved, the avatar follows; that is the avatar tracks the movements of the user's head (while wearing the HMD). The prototype for this project primarily concerns construction of the avatar, where development and/or modification of an HMD are beyond the scope of the thesis.

In the most ideal case, the avatar is to track the HMD movements perfectly, with no time delay between devices, such that a sense of presence is achieved. While perfect tracking is physically impossible, the intent of this prototype is to construct a model as a basis for a study and improvement.

3.1 HIGH LEVEL DIAGRAM AND DESCRIPTION

Figure 3. High-level Telepresence Concept Diagram – A high level diagram of the visual Telepresence system showing the relationship between high level components.

At the time of writing, no virtual reality HMD exists to be compatible with low-level devices, but instead, a PC that runs a native operating system is required to interface the two parts. An HMD transmits device data to the PC, wherein the PC decodes the data to be transferred to the avatar. The avatar responds to the received YPR coordinates by moving its actuators to track HMD's position. Lastly, a camera transmits video feedback to the PC, which is eventually transferred to the HMD display for viewing.

3.2 CONSTRAINTS

The device implemented by this thesis is constrained for use in short-range (SR) Telepresence, where the distance between the user and the avatar are within close proximity. Of course an avatar that can function in long range is infinitely more practical, but there are many more problems that must be tackled first.

4 HARDWARE SELECTION AND DESIGN

This chapter discusses hardware selection based on known specifications, and the design of additional hardware that are integral parts of the prototype described in this thesis. This chapter serves to segue into Chapter 5, which discusses software and controller design for precise mechanical actuation.

4.1 HMD SELECTION

The Oculus Rift DK2 (Development Kit 2) was selected as the HMD of choice for this prototype due to its low cost, high performance and reliability. The Oculus Rift DK2 features high resolution imaging at 1080p, and has rotational tracking with an update rate of 1000Hz, and has a considerably lower price tag (at time of writing) compared to alternative virtual reality headsets.

Figure 4. HMD [1] – The Oculus Rift DK2 head-mounted display.

The Oculus Rift headset is designed to be interfaced with a PC using one USB port for device communication, and an HDMI port for visual output. However, the Oculus Rift requires proprietary drivers that must be installed to a PC, which are compatible with a few popular operating systems, including Windows and MacOS. There is no necessary hardware modification to the Oculus Rift for integration to the prototype. However, the

device requires additional software to modify its intended purpose for the direction of this thesis, in which the details surrounding implementation will be described in chapter 5.

4.2 CAMERA SELECTION

The Oculus Rift DK2 has a screen resolution of 1080p (1920x1080 pixels), thus it is sensible to incorporate a camera into the avatar that shares the same native resolution to preserve image quality. While the avatar can benefit from two cameras to enable stereoscopic imaging, which would likely improving the user's sense of depth, only one camera was used in implementation due to budget constraints and to simplify the prototype, since its primary purpose is to serve as proof of concept for mechanical head-tracking; the addition of another camera to the prototype for future modification is a trivial process, which will be reserved for future work.

The Oculus Rift DK2 is specified to have a refresh rate of 75Hz. Therefore, it is ideal to find a camera that has a matching frame rate of 75 frames per second (FPS), while also having a matching native resolution. However, such a camera that meets these specifications costs on the order of several 1000USD (at time of writing). To meet with budget constraints, compromises to the camera specs must be made.

The Logitech C920 webcam was selected for integration to the prototype, since it features 1080p resolution at 30 frames per second (FPS), which stands to be a webcam to have one of the fastest frame rates available for its given resolution (at time of implementation).

4.3 ACTUATOR SELECTION

Grossman et al. [6] determined that maximum rotational head velocities for humans, while vigorously shaking their heads, are 780deg/s and 380deg/s for the horizontal and vertical directions, respectively.

The prototype described in this thesis is purposed for stationary situations, viz. sitting, and it is assumed that the device operator will not be shaking his head vigorously for extended periods within the duration of its use. Therefore, Grossman et al.'s findings serve as conservative estimates for the required motor speeds. Converting the maximum rotational head velocities to motor specs,

(4.3.1)

$$\frac{780 \text{ deg}}{s} = \frac{1 \text{ rotation}}{360 \text{ deg}} * \frac{60s}{1 \text{ min}} = 130 \text{ rpm maximum horizontal rotation}$$

(4.3.2)

$$\frac{380 \text{ deg}}{s} = \frac{1 \text{ rotation}}{360 \text{ deg}} * \frac{60s}{1 \text{ min}} = 63.3 \text{ rpm maximum vertical rotation}$$

From (4.3.1) and (4.3.2), it is determined that the motors controlling the yaw (horizontal) and roll axes should be specified to have a minimum of 130rpm, while the motor controlling the pitch (vertical) axis should be specified to have a minimum of 63.4rpm. While there are many flavors of actuators that are deemed sensible for the purpose of this prototype, “Brushless DC Gimbal Motors” (BLDC), or more formally, 3-Phase DC motors, were selected to be the actuators of choice.

The emergence of 3-Phase DC motors introduced an interesting alternative to servos and traditional brushed and brushless DC motors for camera stabilizing applications. 3-Phase DC motors are popular due to their ability to produce high torque in

low speed applications, while also having a compact form-factor. In many applications, 3-Phase DC motors are used in direct-drive mechanisms to eliminate gear boxes, effectively reducing cost.

3-Phase motors consist of three sets of coils that generate a rotating magnetic field as the windings are excited in sequential order. As the name suggests, 3-Phase motors operate best with three excitation voltages in sinusoid form, that oscillate at 120 degrees out of phase from each other. However, this is not required (as shown in the Winding Excitation Table in Figure 5), but the resolution of the motor is increased when the input is in sinusoidal form. The direction of the motor relies on the order in which the coils are energized, and the speed is directly proportional to the frequency of the sine waves.

Motor Winding Excitation Sequence						
A	1	1	1	0	0	0
B	1	0	0	0	1	1
C	0	0	1	1	1	0

Figure 5. Excitation Table vs. Motor Windings – Tabulated (simplified) excitation sequence of 3-phase DC motor for clockwise rotation and its corresponding winding sequence (right).

An important specification for selecting a 3-Phase DC motor is based on the number of poles, or the number of windings that are evenly distributed around the stator, Relating number of poles to speed,

(4.2.3)

$$rpm = \frac{120 \text{ deg} * f}{n_p}, \text{ where}$$

f = frequency of the three phase signal,

n_p = number of poles for the given motor,

rpm = rotation speed of the motor in revolutions per minute.

While it is known that a larger number of poles increases the resolution of the motor, and thus results in smoother motor rotation, it also decreases the overall motor speed given the same frequency of the voltage signal (4.2.3). The motors were purchased with the gimbal as part of a camera stabilization kit, in which each motor has a total of 24 poles. Testing revealed that each motor consumes approximately 5W (10V at 500mA) when fully energized.

4.4 GIMBAL FRAME SELECTION

A gimbal is a mechanical support that is inclusive of one or more rotational axes, on which an object can be mounted to be rotated. The design of the gimbal extends beyond the intentions of this thesis, and so a pre-built mechanical gimbal was purchased as part of a camera stabilization kit alongside the 3-phase DC motors. The gimbal consists of three axes, each of which has a motor mounted to provide it the ability to pivot via electrical-mechanical control. Each arm has the ability to be extended or shortened, to allow the load to be balanced, such that the torque provided by the motor is minimized. The three axis gimbal selected for this project is shown in Figure 6.

Figure 6. The Three-axis Mechanical Gimbal – This diagram shows the critical components. A webcam and IMU (not fully visible) are loaded to the gimbal.

4.5 SENSOR SELECTION

As previously discussed in section 4.1, the Oculus Rift provides head-tracking capabilities. It is important to know the form of the rotational coordinates used for input, so that a similar coordinate system can be constructed for the avatar control system to be designed. Example source code [12] from the Oculus Rift Source Developer Kit (this will be further discussed in Chapter 5) reveals that the HMD rotational coordinates can be extracted in the form of Yaw, Pitch and Roll (YPR). With this in mind, it would be more convenient to find sensors to be mounted on the avatar that also provide YPR measurements.

4.5.1 INERTIAL MEASUREMENT UNIT

An Inertial Measurement Unit (IMU) is a device that measures velocity, gravitational forces and rotational positioning using an array of sensors, including: an accelerometer, gyroscope and magnetometer. Using these sensors, the module's rotational coordinates can be calculated. The design and assembly of an IMU is beyond the scope of this thesis, and so the Razor 9DOF IMU was purchased for this project.

The Razor 9DOF IMU is a self-contained module that includes an ITG-3200 triple-axis digital-output gyroscope, an ADXL345 triple-axis accelerometer and an HMC588L triple-axis digital magnetometer. The YPR coordinates are computed via a Direction Cosine Matrix algorithm [5]; the theory behind the sensor fusion algorithm will not be discussed in detail here.

The module includes an ATMEL ATmega328 microcontroller that performs the necessary computations, such that YPR data is the output by the IMU via serial port. The

onboard microcontroller makes this module ideal for the prototype, because it offloads the processing power to a separate device from the main controller. This makes the Razor 9DOF IMU ideal for integration, where the controller board (this is discussed in chapter 5) requires intricate timing and processing.

Once the YPR data is calculated, it is pushed to the serial port to allow communication between other devices. For its intended purpose, the magnetometer measures the Earth's poles to use as a reference direction to determine its relative direction when it is rotated. However, this also implies that the magnetometer is sensitive to magnetic fields. Experimental testing revealed that magnetometer of the Razor 9DOF IMU suffers from strong magnetic interference generated by the gimbal motors. This severely altered the readings of the IMU Yaw coordinates, although the Pitch and Roll coordinates were not affected. Ultimately, it was decided that the onboard magnetometer cannot be reliable for the purpose of this project.

4.5.2 POSITION ENCODER (ABSOLUTE) SELECTION

In section 4.5.1, it was revealed that the on-board magnetometer of the Razor 9DOF IMU cannot reliably measure the avatar's Yaw orientation, since it experiences magnetic interference from the motors. Therefore, an alternative sensor that is insensitive to magnetic fields must be employed to act as a substitute for the magnetometer.

In many commercial applications, primarily manufacturing automation, encoders provide a robust method of measuring rotational position. Encoders are available in many types, shapes and sizes, with each having pros and cons. The most basic position encoder is simply a potentiometer with a known voltage range, where the analog voltages can be

mapped to a known position as the dial is swept. While cheap and easy to implement, the potentiometer as an encoder is known to be less reliable due to non-linearity in its resistance as the dial is turned, and due to voltage variation caused by changes to temperature.

The most popular encoder that provides a balance of both performance and reliability is the optical encoder; it consists of two light emitters and detectors, with a disk that consists of many slots that form a ring around it. The disk sits between the emitters and detectors, such that light is able to pass through it when the slot is perfectly in-line with the emitter and detector. When a detector senses light, it generates a digital high signal – otherwise, the output is low. Using a combination of two emitters and detectors allows a quadrature signal (two square waves, 90 degrees out of phase with each other) to be generated as the encoder is turned, such that the number of encoder “ticks” can be tracked in the controller’s memory, and incremented from its starting position to determine its relative position. However, a major drawback to this encoder is that it requires hardware interrupts to be programmed to the controller, where timing must be extremely precise; if the encoder is spun too quickly, the microcontroller cannot trigger quickly enough, which will cause the encoder position to be lost. Another drawback to the optical encoder is that it can only measure relative position; once the encoder and/or controller are powered down, the last known encoder position is lost.

The last and most robust position encoder for discussion is the absolute encoder. The absolute encoder’s precision is determined by its bit resolution, usually in powers of two. Mounted inside the encoder is a disk with n number of rings, where n is the number of bits resolution. The value of the encoder is represented by a *slice*, in which the rings are

read from the innermost ring to the outermost ring or from outermost ring to the innermost ring, depending on the manufacturer. Figure 7 depicts a 3-bit encoder disk.

Figure 7. A 3-bit Absolute Encoder Disk – Dark shaded regions represents a low signal, and white areas represent a high signal. An n th-bit resolution encoder with have an n th number of rings. One slice represents one measured position of the encoder.

A series of light detectors and emitters (one pair representing one bit) allows the current position of the encoder to be output as a digital signal in grey code form. As the disk is rotated (which is attached to the encoder shaft), the light detectors read different values to represent the new encoder position.

The obvious advantage of the absolute encoder is that the position is saved, even when the device receives no power. In addition, each of the bit readings of the encoder can be wired directly to a microcontroller for GPIO input without requiring hardware triggered interrupts or other polling techniques to operate, since there is no incremental component to be tracked to determine its position; the absolute encoder simply delivers continuous output, and can be read from the microcontroller at any point in time.

However, due to its disk geometry and physical limitations, the absolute encoder tends to be larger than the other encoders; the disk is required to be larger to represent more bit values for increased resolution. Another drawback to the absolute encoder is that they require more power to operate, since they integrate a greater number of light detectors and emitters by design. Lastly, the most disadvantageous feature of the absolute encoder is its high cost; absolute encoders are more commonly employed in industrial robots, where performance is paramount, rather than in hobbyists' projects where cost may be more of an important consideration.

Of the three encoder types presented in this section, the absolute encoder was selected for monitoring the avatar's Yaw position (since opportunity presented itself to purchase one cheaply at the time of the project implementation phase).

The prototype makes use of the KOYO TRD-NA256NW 8-bit NPN absolute encoder, which was purchased, used, for 40USD. The datasheet for the KOYO TRD-NA series absolute rotary encoders is shown below in Figures 8 and 9.

Model Number List

NPN Output

Resolution	Output Increasing Rotation Direction	With Cable		With Relay Connector	With the connector for FC2 connection (For FC)	
		Cable Length 2 m	Cable Length 5 m	Cable Length 2 m	Cable Length 2 m	Cable Length 5 m
32 (5-bit)	CW	TRD-NA32NW	TRD-NA32NW5M	TRD-NA32NWE	—	—
	CCW	TRD-NA32RNW	TRD-NA32RNW5M	TRD-NA32RNWE	—	—
64 (6-bit)	CW	TRD-NA64NW	TRD-NA64NW5M	TRD-NA64NWE	—	—
	CCW	TRD-NA64RNW	TRD-NA64RNW5M	TRD-NA64RNWE	—	—
128 (7-bit)	CW	TRD-NA128NW	TRD-NA128NW5M	TRD-NA128NWE	—	—
	CCW	TRD-NA128RNW	TRD-NA128RNW5M	TRD-NA128RNWE	—	—
180 (8-bit)	CW	TRD-NA180NW	TRD-NA180NW5M	TRD-NA180NWE	—	—
	CCW	TRD-NA180RNW	TRD-NA180RNW5M	TRD-NA180RNWE	—	—
256 (8-bit)	CW	TRD-NA256NW	TRD-NA256NW5M	TRD-NA256NWE	—	—
	CCW	TRD-NA256RNW	TRD-NA256RNW5M	TRD-NA256RNWE	—	—
360 (9-bit)	CW	TRD-NA360NW	TRD-NA360NW5M	TRD-NA360NWE	TRD-NA360NWF2 (TRD-NA360NWF)	TRD-NA360NWF2-5M (TRD-NA360NWF5M)
	CCW	TRD-NA360RNW	TRD-NA360RNW5M	TRD-NA360RNWE	—	—

Figure 8. KOYO TRD-NA256NW Datasheet Part 1[8] – The encoder is specified to have 8-bit resolution, or 256 possible grey code values for one revolution of the encoder shaft. The encoder is also designed for NPN output.

Electrical Specifications

Model Number		TRD-NA□NW	
Power Supply	Supply Voltage	10.8 to 26.4 V DC	
	Allowable Ripple	3% rms or less	
	Consumption Current**	70 mA or lower (2,048 pulses: 100 mA or lower)	
Output Code		Gray binary* ¹	
Maximum Response Frequency		20 kHz* ²	
Precision		(360/ (Resolution x 2))°	
Rotation Direction		Clockwise rotation (CW) or counterclockwise rotation (CCW)* ³	
Output	Output Form	NPN open collector output	
	Output Logic	Negative logic (Active low)	
	Residual Voltage	0.4 V or lower	1.5 V or lower
	Inflow-Outflow Current	16 mA	32 mA
	Load Supply Voltage	30 V DC or lower	
Rise / Fall Time* ⁵		Not larger than 2.0 μs (Load resistance 1kΩ)	

Output Circuit

NPN Output

Connection

Cable Type Core Wire Color	Connector Type Pin No.	2,048 Resolution	1,024/720 Resolution	512/360 Resolution	256/180 Resolution	128 Resolution	64 Resolution	32 Resolution
Blue	1	0 V	0 V	←	←	←	←	←
Brown	2	+12/24 V	+12/24 V	←	←	←	←	←
Black	3	bit 1 (2 ⁰)	bit 1 (2 ⁰)	Not connected	←	←	←	←
Red	4	bit 2 (2 ¹)	bit 2 (2 ¹)	bit 1 (2 ⁰)	Not connected	←	←	←
Orange	5	bit 3 (2 ²)	bit 3 (2 ²)	bit 2 (2 ¹)	bit 1 (2 ⁰)	Not connected	←	←
Yellow	6	bit 4 (2 ³)	bit 4 (2 ³)	bit 3 (2 ²)	bit 2 (2 ¹)	bit 1 (2 ⁰)	Not connected	←
Green	7	bit 5 (2 ⁴)	bit 5 (2 ⁴)	bit 4 (2 ³)	bit 3 (2 ²)	bit 2 (2 ¹)	bit 1 (2 ⁰)	Not connected
Purple	8	bit 6 (2 ⁵)	bit 6 (2 ⁵)	bit 5 (2 ⁴)	bit 4 (2 ³)	bit 3 (2 ²)	bit 2 (2 ¹)	bit 1 (2 ⁰)
Gray	9	bit 7 (2 ⁶)	bit 7 (2 ⁶)	bit 6 (2 ⁵)	bit 5 (2 ⁴)	bit 4 (2 ³)	bit 3 (2 ²)	bit 2 (2 ¹)
White	10	bit 8 (2 ⁷)	bit 8 (2 ⁷)	bit 7 (2 ⁶)	bit 6 (2 ⁵)	bit 5 (2 ⁴)	bit 4 (2 ³)	bit 3 (2 ²)
Black / White	11	bit 9 (2 ⁸)	bit 9 (2 ⁸)	bit 8 (2 ⁷)	bit 7 (2 ⁶)	bit 6 (2 ⁵)	bit 5 (2 ⁴)	bit 4 (2 ³)
Red / white	12	bit 10 (2 ⁹)	bit 10 (2 ⁹)	bit 9 (2 ⁸)	bit 8 (2 ⁷)	bit 7 (2 ⁶)	bit 6 (2 ⁵)	bit 5 (2 ⁴)★
Orange / white	13	bit 11 (2 ¹⁰)	Not connected	←	←	←	←	←
Shield	—	GND	GND	←	←	←	←	←

* The parenthesis marks the bit corresponding to the binary code.

★: Most significant bit

Figure 9. KOYO TRD-NA256NW Datasheet Part 2 [8] – Electrical specifications for the KOYO TRD-NA series encoders and connection diagram table for models with various resolutions. Specs highlighted in yellow correspond to specifications relating to the TRD-NA256NW model.

Due to its relative size and weight, it is less feasible to mount the encoder to the Pitch and Roll axes motors. However, the encoder does not pose a problem when mounted to the Yaw axis, since the Yaw axis motor is stationary relative to the load, and to the rest of the gimbal axes. Figure 10 shows the motor and TRD-NA256NW coupled together and mounted to the base plate.

Figure 10. The Mounted Yaw Axis Motor – The Yaw axis motor and rotary encoder are mounted to the base plate, and are stationary relative to the rest of the mechanical gimbal. The Yaw motor and encoder shafts are connected via flexible shaft coupler.

To integrate the TRD-NA256NW encoder to a microcontroller, additional electronic circuitry must be built. For an encoder with NPN output, the circuit shown in Figures 11 and 12 was constructed.

Figure 11. Encoder to Microcontroller Interface [8] – Circuit schematic of the 1K-ohm pull-up resistor bank for NPN output interface to a microcontroller.

Figure 12. Encoder to Microcontroller Interface (Physical Implementation)

Referring to Figure 11, each of the 1K-ohm pull-up resistors act to limit the current from VCC2 to ground when the NPN transistor is turned on (when light is detected based on the disk's position). These resistors also serve to set the output level to VCC2, when the transistor is turned off. In essence, this causes the bit readings to be inverted; when the microcontroller detects a high level input, the bit is low valued.

Since the supply voltage required by the encoder [8] is higher than that of the rest of the electronics in the prototype, a DC-DC boost converter is used to step the supply voltage of the encoder to 12V.

4.6 MOTOR DRIVERS

For mechanical actuation control, it is mostly impossible for stand-alone microcontrollers to deliver the amount of electrical power that actuators require. In almost all cases (aside from small hobby servos), additional motor drivers and high-powered FETs are required to interface microcontrollers with actuators.

Previously discussed in Section 4.2, 3-phase DC motors were selected as the mechanical actuators for the prototype. The integration of these motors presents a unique

problem, in that they each require 3 sets of sinusoidal voltages. Ultimately, it is the microcontroller's responsibility to drive and modify these voltages to provide some method of motor control (i.e. velocity, positioning, etc.). At the time of writing, there are a few viable options that work well to interface 3-phase DC motors with microcontrollers. One option is to generate three PWM signals to be applied to the input of three FETs (one for each phase/PWM signal) in conjunction with H-bridges (for directional control); another option is to use a motor chip/device that is specifically designed to drive 3-phase motors; it was found that both methods require three PWM signals, and are in fact very similar.

Several H-bridge chips were purchased, and tested against the motors. Ultimately, the SN754410 Quadruple Half H-Bridge Driver was selected to be used in the prototype, since it fulfills the power delivery requirements (Figure 13) and also contains four sets of H-bridges on a single chip to drive all three phases of one motor.

		MIN	MAX	UNIT
V_{CC1}	Output supply voltage range	-0.5	36	V
V_{CC2}	Output supply voltage range	-0.5	36	V
V_I	Input voltage	-0.5	36	V
V_O	Output voltage range	-3	$V_{CC2} + 3$	V
I_P	Peak output current		± 2	A
I_O	Continuous output current		± 1	A
P_D	Continuous total power dissipation at (or below) 25°C free-air temperature ⁽³⁾		2075	mW
T_A	Operating free-air temperature range	-40	85	°C
T_J	Operating virtual junction temperature range	-40	150	°C
T_{stg}	Storage temperature range		260	°C

Figure 13. SN754410 Quadruple Half H-Bridge Driver Specifications – Critical specifications are highlighted in yellow. This driver meets the power requirements of the motors discussed in Section 4.3.

Figure 14. SN754410 Quadraple Half H-Bridge Driver Circuit – PWM1, PWM2 and PWM3 ports are wired to the timer output pins of the microcontroller. L1, L2 and L3 represent each of the phase windings for a 3-phase DC motor.

Figure 14 above shows the wiring diagram to interface the SN754410 to a microcontroller. PWM1 (SIG1A), PWM2 (SIG1B) and PWM3 (SIG2A) represent the driver pins that are mapped to the timer output of a microcontroller. L1, L2 and L3 represent each of the phases of a 3-phase DC motor. The 5V_ENABLE signal powers the chip, and turns it on (mapped to EN and VCC1 inputs of the motor driver). Lastly, 12_MOTOR_POWER (mapped to VCC2) is the high voltage DC signal that is delivered to the motor; the value of 12_MOTOR_POWER can be within any range that allows the motor to operate. It was determined via experimentation and tweaking that the motors operate best when 12_MOTOR_POWER is set to a value of 10V; this provides an optimum balance of high torque and low current delivered to the motors, such that heat dissipated by the driver chips is reduced.

4.7 MICROCONTROLLER SELECTION AND BOARD DESIGN

The microcontroller acts as the “brain” for many modern control systems; they provide a large variety of mathematical functions and have the ability to carry out complex algorithms for controlling hardware and manipulating data. As such, microcontrollers come in many varieties to meet broad processing requirements ranging from simple to demanding tasks. For this project, the Atmel ATmega2560 microcontroller was selected due to its large number of GPIOs, Timers and Serial Ports.

As previously discussed in Section 4.1, the Oculus Rift is a PC-based device that will serve to collect head-tracking data. This data is to be pushed to the microcontroller (more of this will be described in chapter 5). However, for a PC to communicate with a serial device (microcontroller), a Universal Asynchronous Receiver/Transmitter (UART) must be used to translate data between USB and Serial (microcontroller) protocols.

Like the other devices described in this thesis, UARTs come in different models, the most common being the FTDI FT232RL chip for USB-to-Serial conversion; the FTDI FT232RL was selected to be used in the prototype.

Recall that the Razor 9DOF IMU (Section 4.4.1) uses serial communication for data transfer. No UART chip is necessary for the IMU and the controller board to communicate, since both devices support serial protocol by default.

Careful planning and a thorough understanding of the peripherals (and their protocols) that are to be connected to the microcontroller led to the design and assembly of the controller board. The complete circuit schematic, PCB layout and physical implementation are shown in Figures 15, 16 and 17.

Figure 15. Schematic of Controller Board drawn using Eagle PCB design suite. A larger version of this schematic can be found in Appendix A.

Figure 16. PCB Layout – Top view of the PCB layout of the controller board, designed in Eagle.

Figure 17. Controller Board (Physical Implementation) – Physical implementation of the controller board, with references to critical components.

The controller board was built as a platform for the ATmega2560 microcontroller and designed as a two-layer PCB; it was manually routed using Eagle PCB design software. The board features nine exposed PWM outputs (three PWM outputs for each 3-phase DC motors) and three SN754410 motor drivers (one for each motor), two exposed serial ports (one for the Razor 9DOF IMU), an on-board UART (FTDI FT232RL) with USB connector for PC communication, exposed GPIO pins to interface with the absolute encoder and a 3.3V regulated output (in addition to microcontroller's 5V reference) for lower-powered devices (IMU). The board also features a 16MHz crystal to set the microcontroller system clock speed.

4.8 PROTOTYPE INTEGRATION

Moving forward from the parts selection phase, the prototype is constructed and is coined *the HMD-Operated Gimbal revision 1* (HOG-1). For the rest of this thesis, HOG-1 and avatar are used synonymously.

The HOG-1 is designed to be a compact, self-contained system that is to be mounted on mobile platforms, and thus it is ideal for the avatar to be powered from a signal DC power supply. This section discusses the assembly of the prototype and the various connections made between components. Figure 18 shows the HOG-1 in its fully assembled form.

Figure 18. The HOG-1 Prototype – The fully assembled HOG-1 prototype, showing critical components.

The gimbal motors, controller board, absolute encoder and IMU have different power requirements to function; Table 1 lists each component and their corresponding voltage requirements.

Component Description	Voltage Requirement
ATmega 2560 Microcontroller	5V
Razor 9DOF IMU	3.3V
3-Phase DC Motors	10V
KOYO TRD-NA256NW Absolute Encoder	12V

Table 1. Voltage Requirements for each of the Avatar Components

The controller board has two onboard regulators: one for delivering 5VDC, and the other for 3.3VDC. The 5V regulator is an LT1117 fixed linear voltage regulator (5V version) that provides 5VDC with a current limitation of 800mA; the ATmega2560 microcontroller and its peripherals (including three SN754410 motor drivers) was measured to utilize approximately 110mA. The controller board's onboard 3.3VDC regulator is used to power the Razor 9DOF IMU, and is also an LT1117 fixed linear regulator (3.3V version) that provides 3.3VDC with a current limitation of also 800mA; the Razor 9DOF IMU is measured to utilize approximately 50mA. Measuring the power consumption of the encoder, the KOYO TRD-NA256 drew approximately 0.6W (12V at 50mA). Of all the components listed in this section, the 3-phase DC motors drew the most power, which were measured to consume 5W per motor (10V at 500mA). Thus, the motor voltage (10V) was selected to be the reference voltage for the HOG-1. Figure 19 shows the connection diagram of the components that form the HOG-1 prototype, inclusive of a boost DC-DC voltage regulator.

Figure 19. Overview of Connected Components that Form the HOG-1 – Orange colored blocks correspond to components that are mounted on the avatar, while yellow blocks correspond to components that the user directly interacts with.

5. SOFTWARE DESIGN AND IMPLEMENTATION

This chapter discusses software design for the HOG-1, and is broken into three main components: the PC Client, Vision Processor and the Controller. The PC Client describes all pieces of software that are necessary to run on a PC for the HOG-1 to function. The Vision Processor runs as an extension of the PC Client, which modifies the camera(s) (two camera for stereoscopic imaging) output to be displayed to the HMD. The Controller describes the software that is installed on the HOG-1 avatar. Much of the code for implementation is written in C++, with some Python.

5.1 PC CLIENT

The PC Client's purpose is to allow a seamless interface of the Oculus Rift to the HOG-1 controller board. However, the Oculus Rift requires other pieces of software to operate, including device drivers and the Oculus VR Runtime Service that must be installed to the PC. This section describes software dependencies, design and implementation of the PC Client. Additionally, the Vision Processor is run on a PC to enable functionality to the HOG-1 avatar, so it is also discussed in this section.

5.1.1 DEPENDENCIES

At time of writing, the Oculus Rift is not yet a commercially produced product and is still in its earlier stages of development (hence, the full name of the HMD used in this project is titled "Oculus Rift Development Kit 2"). Not much is known about the device compatibility for future revisions, but at its current stage, it is limited for use on PCs running Windows and MacOS operating systems.

The device dependencies are primarily due to proprietary drivers and the Oculus Runtime Service program, that were written specifically for more popular PC operating systems to market to a larger audience (although there is also growing support for Linux at the time of implementation). The Oculus Runtime Service provides functionality to the Oculus Rift HMD by handling the device's native IMU processes (YPR calculations), screen distortion correction and display settings. In addition, the Oculus Runtime can be used to calibrate the HMD to account for variances in inter-pupillary distance (IPD) between human individuals and to store the calibration settings in memory to be recalled. Hence, for the Oculus Rift to be used in conjunction with the HOG-1 system, a PC with an operating system that is compatible with the Oculus Rift software must serve as a platform to interface them. Due to its popularity and large support within the VR community, Window 7 was selected as the operating system for which for the PC Client will be developed.

While it is known that the Oculus Rift provides head tracking capabilities (via internal IMU) and outputs data that corresponds to head rotation and orientation to be used for generating VR content, the same data and methods also apply here. However, instead of using this data to perform computations which intended purpose is to create VR experiences, the data must be relayed to an external device, viz., HOG-1 avatar for mechanical actuator control.

Data acquisition is simplified by integrating the Oculus Source Development Kit (Oculus SDK) into the programming environment, which includes libraries and APIs (written in the C++ language) that are specifically geared for third-party developers to

develop custom VR content for the Oculus Rift. Chapter 5.1.2 will discuss in greater detail, the necessary APIs used in implementation.

5.1.2 HMD DATA ACQUISITION

The Oculus Developer Guide [12] is a document designed to help developers quickly create VR content. It contains references to common APIs and functions (found in the *LibOVR* library of the Oculus SDK) that allow data from the HMD to be extracted for use in software development. Following the suggestion of the guide (and other developers), Microsoft Visual Studio 2013 was selected as the integrated development environment (IDE) for the coding platform of the PC Client.

Following example code provided by the Oculus Developer Guide, the HMD rotational coordinates were retrieved and tested by outputting the YPR data to a terminal screen; results are shown in Figure 20. The explanation of the code will not be discussed in detail here, since it can be readily viewed in Oculus Developer Guide.

```
C:\Users\dmchan\Desktop\Oculus044\Debug\Oculus044.exe
Debug: PerformanceTimer UsingUistaOrLater = 1
Debug: NtQueryTimerResolution = Min 15600 us, Max 500 us, Current 1000 us
Debug: [NetClient] Connected to a server running version 1.2.0 (my version=1.2.0)
Debug: Setting up display shim
Debug: Failed to open file: C:\Users\dmchan\AppData\Local\Oculus\Devices.json
Debug: Using profile dmchan
Debug: [SharedMemory] Creating factory
Debug: [SharedMemory] Creating shared memory region: OURObject00 > 1184 bytes
yaw: -40.6887
pitch: -0.564644
roll: 1.61172
```

Figure 20. Terminal Window of HMD Test Program – A test program that extracts the YPR data from the Oculus Rift HMD and outputs the data to a terminal window.

In addition to extracting the YPR data from the HMD, the PC Client is to transfer them to the HOG-1 Controller. The HOG-1 controller board (discussed in Chapter 4) contains an FTDI FT232RL UART chip that allows asynchronous serial communication with a PC via USB. Including the *windows.h* and *stdio.h* header files to the PC Client source code allows serial port manipulation by the program when it is executed.

Observing Figure 20 reveals that the YPR data is stored in float-type variables. This presents a problem when transmitting the data from the PC to the Controller. Because each digit must be converted to ASCII form first before being sent via serial, long strings must be transmitted to represent the same data (one string per axis). The limitation to how quickly the Controller receives readings is dependent upon how quickly data is transmitted from the other device, which depends on the baud rate of the serial transfer.

5.1.3 DATA FORMATTING AND SERIAL PROTOCOL SETTINGS

The baud rate describes speed of data transmission from one serial device to another, and is expressed in units of bits per second. While a higher baud rate increases the speed of transmission, it is also limited by the clock speed of the two devices that are in communication with each other; higher clock speed corresponds to higher baud rates.

In serial port transmission, an ASCII character is sent one character at a time. An ASCII character represents one byte of information, in which each digit of YPR data must be converted. Asynchronous serial communication also requires a start bit to signal that one byte of data is being sent, and a stop bit to indicate that the byte transfer is complete. This is a required procedure for devices having unsynchronized clocks (each device having its own separate clock), such that some form of handshaking is necessary for them to

communicate. Therefore, a single byte of information to be transferred via serial protocol requires a total of 10 bits to be transferred.

To transfer reliable data from the PC to the Controller, compromises must be made; it is assumed that the HOG-1 Controller cannot efficiently use more than a hundredths place resolution for each of the rotational coordinates (nor is it realistic to believe that a human can detect head rotation to less than a 0.01 degree). By limiting each axis to a precision of two decimal places, YPR data for each axis can be limited to a maximum of 5 digits (when rotation is greater than 99.99 degrees).

However, there is an intrinsic complication to serial communication that must be resolved. As the PC client continuously streams data, the Controller receives said data with no meaning or correlation; that is, there is need for an intelligible way for the Controller to determine which piece of data corresponds to which axis. To resolve this issue, delimiters must be added to the data stream to organize and categorize information; data is rewritten to be output in the following format:

(5.1.2.1)

$$S_{pc} = Y\#_{pc}P\#_{pc}R\#_{pc}E, \text{ where}$$

S_{pc} = the YPR data set sent from the PC, including delimiters,

Y = Yaw delimiter,

$\#_{pc}$ = 5 digit ASCII code representing Yaw coordinates,

P = Pitch delimiter,

$\#_{pc}$ = 5 digit ASCII code representing Pitch coordinates,

R = Roll delimiter,

$\#_{pc}$ = 5 digit ASCII code representing Roll coordinates,

E = End of data set delimiter

Note that the subscript, pc , is assigned to the dataset in (5.1.2.1) to avoid confusion, since the YPR IMU dataset also has the same format. By constraining each YPR data to 5 digits, with the possibility of the values being negative (thus we must account for another total of three bytes) and with the addition of four delimiters (each 8 bits in length), the total number of bits per dataset (all three axes) can be computed:

(5.1.2.2)

$$n_s = 3 \text{ signs} * \frac{10 \text{ bits}}{\text{sign}} + 5 \text{ digits} * \frac{10 \text{ bits}}{\text{digit}} * 3 \text{ axes} + 4 \text{ delimiters} * \frac{10 \text{ bits}}{\text{delimiter}} = \frac{220 \text{ bits}}{\text{set YPR data}}, \text{ where}$$

$n_s = \text{number of bits per data set.}$

Relating the data packet to baud rate, equation 5.1.2.1 is used to calculate the transmission time:

(5.1.2.3)

$$\frac{n_s}{BR} = T_s, \text{ where}$$

$n_s = \text{number of bits per data set} \left(\frac{220 \text{ bits}}{\text{set YPR data}} \right) \text{ from 5.1.2.2,}$

$BR = \text{baud rate in bits per second,}$

$T_s = \text{time of transmission per data set.}$

Table 2 lists the supported baud rates and the corresponding transmission times:

Baud Rate (bits/s)	Minimum Transmission Time for 220 Bits (ms)
9600	22.92
19200	11.46
38400	5.73
57600	3.82
115200	1.91

Table 2. Common baud rates and the corresponding transmission times for 220 bit-length data sets – It was experimentally determined that a baud rate of 57600**bps** produced the best performance.

Once the data has been received by the Controller, the ASCII characters must be converted back to numerical form. However, it is known that the Controller cannot instantaneously carry out the decoding process. Therefore, there must be some delay between the read cycles of each data set; if the Controller begins to read at the middle of the data sequence, it will wait until the next cycle so that it can read from the beginning of the sequence.

The baud rate setting was varied after completion of the prototype to tweak performance of the HOG-1. Experimentation revealed that a baud rate value of 57600 $\frac{bits}{s}$ (for the PC and Controller) resulted in the best performance for the ATmega2560 without accruing detectable errors. The baud rate plays an important role in microcontroller timing specifications, which will further discussed in section 5.3.

5.1.4 CLIENT ALGORITHMS

Section 5.1.2 described the format of the data to be output by the PC to be sent to the Controller, as well as the serial port settings that both the PC and Controller must agree upon. This chapter discusses the PC Client's algorithmic components. Figure 21 depicts a high level diagram of the PC Client and its algorithms.

Figure 21. A High level Diagram of the PC Client – The routine inside the *while(True)* container is looped while the program runs, and if an available serial port is open.

Once the rotational coordinates are retrieved from the HMD, the values are printed to the terminal window for viewing. Each value is then multiplied by 100 and stored into an integer-type variable; recall that the rotational coordinates directly taken from the HMD readings are stored in float-type variables; by multiplying each float value by 100 and storing the results into integer-type variables, two decimal places are preserved for each coordinate value.

Next, each digit of the integer-valued coordinates must be converted to ASCII form. This is easily achieved by using the C++ function, *itoa()*, found in the *stdlib.h* header file. The *itoa()* function converts an integer value and stores it into a predefined character array. Example C++ code for the use of this function is shown below in Listing 1:

```
#include <stdlib.h>

void main(){
char buffer[20]; //The character array must first be defined.
int integer = 1234; //Some integer value is declared.

itoa(integer, buffer, 10); //itoa converts each integer digit and stores the value into buffer.

//Print out values to terminal to test.
cout<< "Output: << endl;
cout << buffer[0] << endl << buffer[1] << endl << buffer[2] << endl << buffer[3] << endl;
}
```

```
//Each value in buffer is stored and printed out as a char-type.
Output:
1
2
3
4
```

Listing 1. Example C++ usage of the *itoa()* function included in the *stdlib.h* header file – This function converts an integer into string; each digit is placed into a different index of the array.

Next, the YPR strings (separated by delimiters) are transmitted to the serial port, using the Windows API, *WriteFile()*; an excerpt of the PC Client software code is shown in Listing 2.

```
//Data sent with delimiters in format: Y#P#R#E
//hPort contains the address of the serial port to write.
//yaw_ser, pitch_ser and roll_ser are character arrays that have been defined earlier in the code.

WriteFile(hPort, "Y", 1, &byteswritten, NULL);
WriteFile(hPort, yaw_ser, 5, &byteswritten, NULL);
WriteFile(hPort, "P", 1, &byteswritten, NULL);
WriteFile(hPort, pitch_ser, 5, &byteswritten, NULL);
WriteFile(hPort, "R", 1, &byteswritten, NULL);
WriteFile(hPort, roll_ser, 5, &byteswritten, NULL);
WriteFile(hPort, "E", 1, &byteswritten, NULL);
```

Listing 2. Sample Serial Write Sequence for #Y#P#R#E data – Sample C++ code for writing YPR data and delimiters to the serial port, directly taken from the PC Client source code.

Once the YPR data has been written to the serial port, the algorithm runs again; the algorithm is looped until the serial port is closed (the FT232RL device is disconnected), or if the user manually terminates the program.

5.1.5 VISION PROCESSOR

For images to be viewed properly on an HMD it is necessary to format the video feed from the camera into side-by-side (SBS) configuration. As a first generation prototype, the Vision Processor is executed as a separate program that runs in unison with the PC Client; for the HOG-1 prototype to function, both programs, the PC Client and Vision Processor, must be running simultaneously.

The Oculus Rift DK2 consists of a single panel display, where images are configured for SBS, in such a way that each eye is isolated to one half of the screen (the left eye views the left screen, while the right eye views the right screen); each half of the screen contains a separate image. Figure 5.2.1 shows an example of an SBS stereoscopic image.

Figure 22. Example of a Stereoscopic Image in SBS Configuration – For a small IPD, the differences in the two halves are not extremely apparent. However, differences are more noticeable by comparing the corners of the images. (from Wikimedia commons) [7]

For its purpose in VR applications, each half of the HMD screen is viewed from the perspective of its respective virtual eye to create a sense of stereoscopy. That is, each eye is separated by some distance (IPD) to create different viewing angles for each eye – this naturally mimics the human vision system, and so the user is able to gain a sense of depth. This concept can be integrated into Telepresence applications whereby each half of the screen displays real-time footage of two cameras that are also separated by some IPD. However, the prototype described in this thesis makes use of one only camera – thus the video feed from the single camera must be configured for SBS configuration, where the projection on both halves of the screen is identical.

For a camera that has the same native resolution as the Oculus Rift, the dimensions for the captured images can be computed by splitting the width of the image (1920 pixels)

in four identical segments; the two centermost segments of the original image is projected on each half of the Oculus display to create a non-stereoscopic image feed. Figure 23 shows a diagram of how a single image can be converted to an SBS non-stereoscopic image, while preserving the image dimensions to be projected to the HMD.

Figure 23. Process for Converting an Image to SBS Format – The Region of Interest (ROI) is segmented from the original image and projected to each half of the HMD screen (SBS-Configured). Since the ROI is identical for each half of the SBS-configured image, the result produces non-stereoscopic viewing.

Previously discussed in Chapter 4, the Logitech C920 webcam was selected to be used in the prototype. This camera is easily integrated into software, since it is supported by popular Computer Vision libraries. The Vision Processor is written in Python 2.7, with the OpenCV and Numpy libraries.

OpenCV (standing for *Open Computer Vision*) is a library of computer programming functions, developed by Intel, and is compatible with C++ and Python, and is regarded to be the most popular tool within the Computer Vision community due to its ease of use and high quality documentation for rapid prototyping vision-related algorithms. The OpenCV libraries were created for real time video applications, and are employed in both consumer and industrial settings, which makes it a viable tool for the development of this project.

Numpy is another library that is often used in conjunction with the OpenCV libraries, because it adds Matlab-style matrix manipulation functionality, which is more intuitive for a programmer to use; images are stored in memory in the form of multi-dimensional arrays (the number of dimensions depends on its color scheme) and can be easily modified when represented in matrix form. Figure 24 shows the block diagram of the Vision Processor.

Figure 24. Block Diagram of the Vision Processor – The routine inside the *while(True)* container is looped while the program runs, and if the webcam is available.

Following Figure 24, the program first searches for an available webcam; if one is detected, the frame rate setting is applied (configured for 30FPS) – otherwise the program closes. Once the program enters the *while(True)* container, an image is captured and stored into an array. The ROI is found using the methods previously described in Figure 23. The

ROI is then copied to another matrix – it is imperative that the data uses the Numpy method, *copy()*, otherwise the corresponding matrix acts as a pointer array that stores the address of the original image. The information copied to the ROI matrix is then written to the left and right halves of the original frame and the resulting image is output to the screen. Listing 3 shows the corresponding Python 2.7 code and Figure 25 shows an example image produced by the Vision Processor.

```
import numpy as np
import cv2

width = 1920
height = 1080

left_center_col = width/4
right_center_col = 3*left_center_col

capture = cv2.VideoCapture(0) #Search for available webcam and assign to object.
Capture.set(5, 30) #Change framerate to 30 frames/s

while(True):

    ret, frame = capture.read()
    ret = capture.set(4, height) #Set the captured frame to 1080p
    ret = capture.set(3, width)

    roi = np.copy(frame[0:1080, left_center_col:right_center_col])

    frame[0:roi.shape[0], 0:roi.shape[1]] = roi
    frame[0:roi.shape[0], roi.shape[1]:2*roi.shape[1]] = roi

    cv2.imshow('HMD Output', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

capture.release()
cv2.destroyAllWindows()
```

Listing 3. Vision Processor written in Python 2.7 – Converts the video feed captured by the webcam to SBS and outputs the result to the screen using Numpy and OpenCV libraries.

Figure 25. Results of the Vision Processor – An image captured by the Logitech C920 webcam and produced by the Vision Processor.

5.2 CONTROLLER DESIGN

The Controller is the software that is uploaded to the ATmega2560 microcontroller of the HOG-1 board, and is responsible for directing actuator movement based on the data it receives from the HMD and the IMU. The Controller makes use of the C++ ME405 library, compiled by Professor John Ridgely of Cal Poly's Mechanical Engineering Department.

5.2.1 REAL TIME OPERATING SYSTEM

The Controller integrates a Real Time Operating System (RTOS) to schedule events performed by the microcontroller. These events are known as *tasks*, to which an RTOS behaves as an instrument to handle multitasking, or the running of multiple tasks at the same time – in actuality, these tasks are run in quick succession to appear as though they are running simultaneously. Figure 26 illustrates a task timing diagram:

Figure 26. Task Timing Example – Task timing diagram of 3 tasks, each with 2ms timing requirements in two execution cycles, scheduled by an RTOS. The solid black lines indicate the task timing requirements.

There are three considerations to consider when designing tasks for an RTOS:

- The Timing Requirement describes the time between execution cycles.
- Execution Time describes how long it takes to execute that task.
- The sampling rate of the sensor/timing requirement of the actuator.

The RTOS (freeRTOS) implemented in this project is set for preemptive multitasking, where the RTOS switches tasks during mid-execution depending on those tasks' timing and priority. For a single processor to switch between tasks very quickly, it is imperative that the timing requirements of each task are kept short, while being long enough for the tasks to be completed. For example, if one wanted to simultaneously rotate two motors, the movements of each motor must be broken down into smaller increments, say one degree per task. The processor should start by incrementing the rotation of the first motor by one degree, followed by incrementing the rotation of the second motor by one

degree. By allotting the timing requirements to one millisecond, for example, the motors will appear to turn at $\frac{1 \text{ revolution}}{360 \text{ degrees}} \times \frac{1 \text{ degree}}{1 \text{ ms}} \times \frac{60000 \text{ ms}}{\text{minute}} = 166.67 \text{ RPM}$ simultaneously.

An important feature of RTOS is that tasks are allowed to communicate with each other. In a Mechatronic system, an actuator typically responds to data collected by some sensor; that is, one task is responsible for collecting sensor data, and another task manages the actuator response. The ME405 library contains a class, *shared_data*, that allows the use of an external variable that can be written to (using the *put()* method), or read from (using the *get()* method) inside of any task. The shared variables are thread-safe, meaning that the program cannot interrupt the *put()* and *get()* methods – this prevents data transfer from being interrupted.

5.2.2 CONTROLLER TASKS, TIMING AND PRIORITY

Now that the hardware and Controller-to-PC interface are known, a task diagram can be constructed to organize the behavior and functionality of the software that will run on the Controller. The timing requirements can be tweaked and modified after completion of the prototype to enhance performance. Figure 27 depicts a high level task diagram of the Controller software, running an RTOS.

Figure 27. Task diagram of the HOG-1 Controller – Shared variables of tasks are connected by dotted lines.

Task diagrams are helpful for software design, because they serve to break down large and complex systems into several tasks. They provide a visual means of tracking the various pieces of data that must be communicated between tasks, while additionally providing timing specifications and priority for each of those tasks. Figure 27 shows the Controller broken into three tasks. As the name implies, the Read HMD and Read IMU tasks intercept data from the IMU and PC Client via serial port, respectively. These tasks also translate

incoming data from ASCII form (since the YPR data must first be encoded to ASCII for serial transmission) to numeric values and store the results into their respective shared. The Motor Controller task reads from the shared variables to retrieve the YPR data from the IMU and HMD to determine motor rotation direction and calculate positional error values. These error values are used to calculate gains via Proportional-Integral-Derivative (PID) controller, in order to control the speed of each motor. The PID controller and the associated theory will be explained in a later section.

The timing specification for the Read HMD and Read IMU tasks were selected to be 4ms; timing for these tasks were selected based on the rate of serial data transfer, or baud rate, from the PC to Controller and from the IMU to the Controller. Reviewing section 5.1.3, the data sent from the PC to the Controller is in the form Y#P#R#E (5.1.2.1), and the PC and Controller share a baud rate of $57600 \frac{\text{bits}}{\text{s}}$. To simplify implementation and to preserve consistency, the YPR dataset from the IMU output is in Y#P#R#E form:

(5.2.2.1)

$$S_{imu} = Y\#_{imu}P\#_{imu}R\#_{imu}E, \text{ where}$$

S_{imu} = the YPR data set sent from the IMU, including delimiters,

Y = Yaw delimiter,

$\#_{imu}$ = 5 digit ASCII code representing Yaw coordinates,

P = Pitch delimiter,

$\#_{imu}$ = 5 digit ASCII code representing Pitch coordinates,

R = Roll delimiter,

$\#_{imu}$ = 5 digit ASCII code representing Roll coordinates,

E = End of data set delimiter.

Observance of equation (5.2.2.1) reveals that the IMU YPR dataset has the exact same format as the PC YPR dataset (5.1.2.1). Additionally, the IMU-Controller baud rate is also set to $57600 \frac{\text{bits}}{\text{s}}$, thus, both the Read IMU and Read HMD tasks share the same timing requirement. From Table 2, the timing requirement can be estimated to be 4ms.

Determining the timing for the Motor Controller task for the current prototype is tricky, as well as being flawed (the specifics of this will be discussed later). For now, based on the example of the two simultaneously running motors in Section 5.3.2, it is best to keep the lowest timing possible since it directly relates to the speed that the motor can rotate; the Motor Controller task is selected to have a timing requirement of 0.1ms.

Priority designates tasks of certain order, such that one task might be given a preference in the order that it is executed before other tasks. Priority is particularly useful in cases where multiple tasks have the same timing requirement and it is imperative that those tasks be executed in a specific sequence. It should be noted that all three tasks (Read HMD, Read IMU and Motor Controller tasks) share the same priority because no one task takes precedence over the other.

In the remaining sections of this chapter, each task will be broken down into its algorithmic parts and covered in greater detail.

5.2.3 THE READ HMD AND READ IMU TASKS

The Read HMD and Read IMU tasks share the same algorithm for reading incoming YPR data from the serial port, as well as sharing the same procedure for converting YPR ASCII codes to integer values. Hence, it is appropriate to combine the discussion of these two tasks into one chapter section.

5.2.3.1 DATA PARSING

Recall that the PC Client and IMU continually output YPR data via serial in the format Y#P#R#E. It was discussed earlier in Section 5.1 that there is a need for an intelligible way for the data to be read, such that each piece of data can be mapped to its corresponding axis. The ideal case is that all the YPR coordinates are read within each task cycle – this allows all axes coordinates to be captured within one instance of time, rather than collected at separate times. The trade-off for gathering all the YPR coordinates in one sitting is that the task execution time will be longer, and the inevitable result is that it will interfere with the timing requirements of shorter tasks (Motor Controller task). However, the “read-all-at-once” method is implemented for the first revision of the Controller, where modification or alternative solutions are reserved for future work.

Another intrinsic problem exists for the methods described above – when the Read HMD and Read IMU tasks are executed, the serial data is read, but at an unknown part of the sequence. Therefore, some method of polling must be implemented for reading to be synched with the incoming data. A parsing algorithm was developed to take advantage of the delimiters in the Y#P#R#E format, which serves as the fundamental process within the

Read HMD and Read IMU tasks. Figures 28 and 29 depict the state diagram for the Read HMD and Read IMU tasks, respectively:

Figure 28. State diagram for Read HMD task.

Figure 29. State diagram for Read IMU task.

The parsing algorithm operates in two modes: Read and Convert Modes. The Read Mode allows data to be read from the serial port – once the data is collected, the process transitions to Convert Mode, where the data is converted to numeric values and written into shared variables. Once the conversion process is finished and the data is written, the task is complete. When the task is executed again, the process is repeated; in this way, the YPR values are updated each time that the task is run.

Following the algorithm shown in Figure 28, an “end_seq” flag indicates whether or not the parsing algorithm is in Read Mode. By default, the “end_seq” is set to false at the beginning of the task execution. Each bit is read by the program, but does nothing until the first delimiter, Y, is detected in the serial buffer. This acts to ensure that the sequence is read from the beginning of the Y#P#R#E sequence.

If a minus character, “-“, is detected, a “negative flag” is set to indicate that the current piece of data being read from the serial port will be negative valued. If the byte corresponds to some equivalent character that is representative of a digit (number digit in ASCII), the character is placed into its corresponding buffer. Each time that a character is placed into a buffer, an internal counter is incremented to determine the number of digits that the value should hold for that corresponding axis. A flag is triggered after detecting a delimiter, which also behaves as a designator for where the data should be placed; when the “End” delimiter is detected, the digit counter stops – this is necessary, because the length of data is unknown (a rotation angle greater than 99.99 will result in a 5 digit ASCII code, while a rotation angle less than 1 will result in a 2 digit code). This will put the task into Convert Mode (the “end_seq” flag set), such that incoming data will no longer be read. Instead, the task will call upon the *atoi()* function to convert the buffered data into integer

values; Listing 4 shows sample code for the conversion process of one axis (Yaw), and the code for the *atoi()* function. Lastly, the values will be written to their respective shared variables; the task is complete, and the flags are reset to ready the task for its next iteration.

//YAW CONVERSION PROCESS

```
if (y_count == 5) { //5 Digits
yaw = 100*(100 * atoi(y_buf[0]) + 10 * atoi(y_buf[1]) + atoi(y_buf[2]) + 0.1 * atoi(y_buf[3]) +
0.01 * atoi(y_buf[4]));
}
else if (y_count == 4) { //4 Digits
yaw = 100*(10 * atoi(y_buf[0]) + atoi(y_buf[1]) + 0.1 * atoi(y_buf[2]) + 0.01 * atoi(y_buf[3]));
}
else if (y_count == 3) { //3 Digits
yaw = 100*(atoi(y_buf[0]) + 0.1 * atoi(y_buf[1]) + 0.01 * atoi(y_buf[2]));
}
else if (y_count == 2) { //2 Digits
yaw = 100*(0.1 * atoi(y_buf[0]) + 0.01 * atoi(y_buf[1]));
}
if (y_neg) {
yaw = -1 * yaw;
}
```

//ASCII TO INTEGER CONVERSION

```
int readhmd_task::atoi(char digit){
    if (digit == '1') {
        return 1;
    }
    else if (digit == '2') {
        return 2;
    }
    else if (digit == '3') {
        return 3;
    }
    else if (digit == '4') {
        return 4;
    }
    else if (digit == '5') {
        return 5;
    }
    else if (digit == '6') {
        return 6;
    }
    else if (digit == '7') {
        return 7;
    }
    else if (digit == '8') {
        return 8;
    }
    else if (digit == '9') {
```

```

        return 9;
    }
    else{
        return 0;
    }
}

```

Listing 4. Example C++ Code for String to Integer Conversion

5.2.3.2 READING FROM THE ABSOLUTE ENCODER

Examination of Figures 28 and 29 reveals subtle differences between the Read HMD and Read IMU tasks; the Read IMU task has an additional process that reads the value of an absolute encoder. Recall that Chapter 4.5 discussed problems associated with the IMU magnetometer; due to magnetic interference caused by the gimbal motors, the IMU yaw values are unusable. The inclusion of the absolute encoder remedies this, by providing its own method of measurement to determine the Yaw values, which effectively bypasses the need for the magnetometer. The absolute encoder is incorporated into software with the addition of a new class (this is based on preference, rather than on necessity), such that an absolute encoder driver is instantiated as an object within the Read IMU task. The GPIO pins that physically connect to the encoder wires are configured for input; the KOYO TRD-NA256NW 8-bit absolute encoder (Chapter 4.5) is of the NPN type, thus the logic is inverted. The encoder outputs values in the form of grey code, which is a modification of binary. However, grey code is not understood by the compiler, so the grey code values must be converted to binary. Table 3 highlights the differences between binary and grey code.

Decimal Equivalent	Grey Code	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

Table 3. Grey Code vs. Binary – Decimal values shown with equivalent Grey Code and Binary values.

Conversion of grey code to binary is performed by using the bitwise exclusive-or operation, *XOR*, starting from the most significant bit (MSB) of the binary equivalent (where the MSB of the grey code and binary value is equivalent) with the adjacent grey code bit. Figure 5.3.4.3 provides a visual example of grey code to binary conversion.

Figure 30. Example of grey code to binary conversion.

Once the grey code is converted to binary, the binary value is automatically converted to decimal form; the compiler understands the binary, hexadecimal and decimal equivalents for unsigned integer values for the convenience of the programmer so that no conversion algorithm is needed to be written. Since the encoder is specified to have 8 bits, the encoder measures $2^8 = 256$ (valued 0–255) equally distributed positions per revolution. Therefore, the encoder has a resolution of:

(5.3.4.1)

$$\frac{\frac{360 \text{ degrees}}{\text{revolution}}}{\frac{256 \text{ increments}}{\text{revolution}}} \cong \frac{1.4 \text{ degrees}}{\text{increment}}$$

Recall that the Yaw readings from the IMU are in signed integer form, where that the output is in the range of -180–180 degrees, and 0 is specified to be the center – the value becomes more negative as the IMU is rotated counter-clockwise and becomes more positive as it is rotated clockwise. Hence, each position of the encoder should correspond to some angular value, where one position is designated to be the referenced as the center (zero degrees). To achieve the same Yaw range as the IMU, the center reference is set to position 129, such that positions 128–0 are negative valued (where position 128 = -1.4 degrees and position 0 = $-1.4 * 127 = -177.8$ degrees); positions 130–255 are positive valued (where position 130 = 1.4 degrees and position 255 = $1.4 * (255 - 129) = 176.4$ degrees). This is achieved by constructing a look up table (LUT), such that each encoder position is mapped to an angular position.

The LUT is stored in a signed-integer array, and initialized in the constructor of the absolute encoder driver class; the LUT is only run once, and is initialized at the start of the program. Listing 5 shows the code that is written to the microcontroller that initializes the LUT:

```
//Generate the LUT for encoder position to angle conversion

encoder_lut_angle[129] = 0;
encoder_lut_angle[0] = 17780;
for(i = 130; i <= 255; i++){
encoder_lut_angle[i] = 140*(i-129);
}

for(i = 128; i > 0; i--){
encoder_lut_angle[i] = -140*(129-i);
}
```

Listing 5. Example C++ Code for LUT Generation – The angles are stored in the encoder_lut_angle array, such that the index of the array corresponds to the encoder position.

Listing 5 shows that each encoder angle has been multiplied by 100; the same has been done to the Pitch and Roll angles measured by the IMU to preserve the coordinate values to two decimal places. Figure 31 shows a few calculated angles mapped to their respective encoder positions and the corresponding scaled value that is to be stored in the shared variable.

Figure 31. Encoder Positions Mapped to LUT – This diagram depicts 7 positions out of 256 possible positions.

5.2.4 THE MOTOR CONTROLLER TASK

The Motor Controller task reads from the shared variables that are written to by the Read IMU and Read HMD tasks to recover both the YPR coordinates from the IMU and the HMD. These values are used by the Motor Controller task to direct the motion of the motors, which allow the avatar to track the HMD. The IMU coordinates represent the position of the avatar, where each of the motors is driven independently by a control system to correct for rotational differences between the HMD and IMU; ideally, the motors are moved in such a way to have the IMU coordinates mimic that of the HMD coordinates.

5.2.4.1 MICROCONTROLLER SETUP

Reviewing the hardware from Chapter 4, the H-bridge drivers require three sets of PWM signals to drive a single three phase motor. The timers of the ATmega2560 must be set up to enable PWM output. The frequency of the timer relates directly to the frequency of the PWM signal and must be set to a value that is not so large that the H-bridge transistors cannot toggle quickly enough. Three separate timers are used control each of the 3-phase motors. The ATmega2560 has three special function registers (SFR) that allow the timer settings to be changed, and as such must be set. The registers include:

- TCCR n A/B (where n is the timer number) consists of two registers that are used to set the frequency of the output. Setting the COM1A1 bit of the TCCR n A register configures the timer to be reset when the counter is equal to the value stored in the Output Compare Register, OCR n N, which is required to enable PWM mode. The CS10, CS11 and CS12 bits of the TCCR n B register sets the output frequency of the PWM signal; the bits are set to configure a pre-scale value that divides the system clock by that pre-scale value.
- OCR n N (where n is the timer number and N is the channel name) is known as the Output Compare Register, which holds an 8-bit integer value (for an 8-bit timer) for the timer counter to compare to. When the timer counter is equal to the value stored in OCR n N, an interrupt is triggered, and the timer is affected depending on the mode that was set by TCCR n A register

Listing 6 shows an excerpt from the source code that sets up the PWM output:

```
//Counter Setup  
  
TCCR1A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);  
TCCR1B |= (1 << WGM12) | (1 << CS10);  
TCCR3A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);  
TCCR3B |= (1 << WGM12) | (1 << CS10);  
TCCR4A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);  
TCCR4B |= (1 << WGM12) | (1 << CS10);;
```

Listing 6. Example C++ Code for Timer Setup (AVR Microcontroller) – PWM set up for timer/counters 1, 3 and 4 of the ATmega2560 microcontroller.

The Timer- n channels A, B and C, where n corresponds to the counter number (counters 1, 3 and 4 are used to drive the motors), are set to 8-bit mode and set have a PWM frequency that is equal to the system clock, or 16MHz (set by the crystal oscillator). The PWM frequency was varied via trial and error to determine the best setting, and it was found that a frequency setting between 16MHz and 500KHz produced the best performance. However, setting the PWM frequency to anything lower than 16MHz resulted in an audibly high-pitched noise that emanated from each of the motors. Once the timers have been set, the PWM output from the timer pins of the microcontroller are ready for use. Table 4 lists the microcontroller pins that have been set for PWM output.

Microcontroller Pin	Timer No.	Channel Name	Motor and Phase Winding	Frequency	Resolution	Output Compare Register
PB5	1	A	1A	16MHz	8-bit	OCR1A
PB6	1	B	1B	16MHz	8-bit	OCR1B
PB7	1	C	1C	16MHz	8-bit	OCR1C
PE3	3	A	2A	16MHz	8-bit	OCR3A
PE4	3	B	2B	16MHz	8-bit	OCR3B
PE5	3	C	2C	16MHz	8-bit	OCR3C
PH3	4	A	3A	16MHz	8-bit	OCR4A
PH4	4	B	3B	16MHz	8-bit	OCR4B
PH5	4	C	3C	16MHz	8-bit	OCR4C

Table 4. ATmega2560 Pins vs. Parameters – Table of the microcontroller pins shown with their corresponding parameters. [2]

The microcontroller pin locations can be referred to via the HOG-1 schematic found in Chapter 4, or alternatively, in the Appendix A. The Output Compare Registers allow the PWM duty cycle to be varied. It follows that the OCR_{nN} registers (those specifically used in the this project) can hold a value between 0 and 255 (recall that the timers are configured for 8-bit resolution), where a value of 0 produces 0% duty, and a value of 255 produces 100% duty. For example, if the OCR1A is set to hold a value of 128, the microcontroller pin, PB5, will output a square wave at 50% duty.

5.2.4.2 MOTOR CONTROLLER THEORY

Unlike traditional motor drivers that require a single PWM signal to control motor speed, the speed of the 3-phase motor is not reliant upon the PWM duty cycle. Rather, PWM signals are used generate the analog behavior of sine waves, viz., the intermediate region between sine wave peaks. For now, we assume that the motor can be driven directly from the microcontroller for simplicity's sake. The goal now is to generate three sets of three sine waves that are 120 degree out of phase – each sine wave per motor winding, and one set per motor. Unfortunately, the microcontroller can only output signals in discrete time increments (the problem with all digital electronics) and digital levels (low and high voltages). However, if the frequency of a PWM signal is high enough, the average voltage that is measured over some period of time (that is, the time of measurement is much longer than the period of the PWM signal), appears to be analog. Extending this concept further, it is possible to divide a sinusoidal signal into some number of discrete steps such that it becomes possible to translate the PWM duty cycle that equates to each analog voltage of a discrete-time sinusoid. Equations 5.2.4.2.1 and 5.2.4.2.2 relate duty cycle to the average DC voltage.

(5.2.4.2.1)

$$D = \frac{T}{P}, \quad \text{where}$$

D = Duty Cycle

T = Total Time that Signal is High

P = Period of the Signal.

The average voltage (or analog voltage) is given by:

(5.2.4.2.2)

$$V_A = DV_{high} + (1 - D)V_{low}, \quad \text{where}$$

V_A = Average DC voltage,

D = Duty Cycle [0, 1],

V_{high} = Digital high voltage,

V_{low} = Digital low voltage (typically equal to 0).

A sinusoid is divided into 48 discrete time steps (chosen as a multiple of 6), and each step is assigned an 8-bit value that corresponds to the average voltage of the PWM output. These values are eventually assigned to the Output Compare Registers (OCR) of the timer to generate a PWM signal that is proportional duty cycle. Now we can derive a relationship between the duty cycle and the OCR value.

(5.2.4.2.3)

$$V_A = V_{high} \frac{OCR}{n_T}, \quad \text{where}$$

V_A = Average DC voltage,

V_{high} = Digital high voltage,

OCR = Value stored in the Output Compare Register,

n_T = Bit resolution of the timer.

Since $V_{low} = 0$ for the microcontroller, equation 5.2.4.2.2 can be simplified to:

(5.2.4.2.4)

$$V_A = DV_{high}$$

Relating (5.2.4.2.4) with (5.2.4.2.3),

(5.2.4.2.5)

$$D = \frac{OCR}{n_T}$$

Hence, the duty cycle is simply the OCR value divided by the timer resolution.

Figure 32 shows the results of three phase signal generation via PWM:

Figure 32. 3-Phase Sine Wave Generation via PWM – Three sets of sine waves generated by PWM signals, captured on an oscilloscope. Each signal is spaced 120 degrees out of phase to create 3-phase DC voltage, which are used to power the 3-phase DC motors.

Figure 32 shows the PWM equivalent of three sets of sinusoids to create 3-phase DC voltages, which are used to drive the 3-phase DC motors. Each signal is directly applied to an input of the H-bridge driver, which sources the necessary voltages that are proportional to the PWM duty cycles to each of the motor windings. To prove that the

results shown in Figure 32 represent three sine waves, a low pass filter is constructed in series with each of the signals (Figure 33); this creates a primitive digital to analog converter (DAC) that is not implemented in the prototype, but rather serves as a proof of concept for discussion in this thesis. Figure 34 shows the analog equivalent of the results shown in Figure 32.

Figure 33. Schematic of a Primitive DAC – A simple RC low-pass filter created to act as a DAC.

Figure 34. 3-Phase Sine Wave Generation via PWM (Filtered) – Three sets of sine waves generated by PWM signals, captured on an oscilloscope. Each signal is spaced 120 degrees out of phase to create 3-phase DC voltage, which are used to power the 3-phase DC motors.

Each time step of the 3-phase voltages corresponds to some positional increment of the motor. The number of increments for a 3-phase DC brushless motor is dependent upon the number poles that the motor has, but more importantly, the number of commutation steps determines the number of increments per sequence. This concept is related to stepper motor control, where the rotor is moved by energizing the motor windings in a sequence of patterns. When a winding is fully energized, the rotor is magnetically attracted to the pole. By energizing windings in a patterned sequence, the rotor can be continuously rotated. Extending this concept, the windings can be partially energized, such that the rotor is attracted to more than one pole; this allows the steps to be broken into smaller increments, and this method is commonly known as *microstepping*. The same principle is applied to this thesis, and the number of increments directly relates to the stepping resolution of the 3-phase DC motor. The corresponding step sequence is known as *step commutation*. The resolution of a 3-phase DC motor, relating to step communication is shown in equation 5.2.4.2.6:

(5.2.4.2.6)

$$\frac{n_p}{2} \times SC = NI, \text{ where}$$

n_p = number of poles,

SC = step commutation (number of states in one step sequence),

NI = motor resolution (number of steps per revolution),

The frequency of the 3-phase signals can also be determined by equation 5.2.4.2.7, since the period of a discrete sine wave is the number of time steps multiplied by the interval of time between steps:

(5.2.4.2.7)

$$f = \frac{1}{n_s * t_s}, \text{ where}$$

f = frequency of the 3 – phase signals,

n_s = number of steps per period,

t_s = time between steps.

It is generally accepted that a higher step resolution is desirable since it results in smoother motor rotation. It then becomes necessary to observe the relationship between motor speed, step resolution and time step intervals, since these variables are inherently dependent upon each other.

From the previous discussion, it was determined that the number of discrete time intervals for the 3-phase signals is also the number of steps in one commutation sequence. Therefore,

$$n_s = SC$$

Thus, equation 5.2.4.2.6 can be rewritten as:

(5.2.4.2.8)

$$\frac{n_p}{2} \times n_s = NI \text{ (5.2.5.2.7)}$$

Recalling equation 4.2.3,

$$rpm = \frac{120f}{n_p}, \text{ where}$$

rpm = motor speed in rotations per minute,

f = frequency of the 3 – phase sinusoids,

n_p = number of motor poles.

Equations 4.2.3, 5.2.4.2.6 and 5.2.4.2.8 can be combined construct a relationship between the motor speed, commutation steps and time between steps:

(5.2.4.2.9)

$$rpm = \frac{60}{NI \times t_s}$$

Equation 5.2.4.2.9 shows that if either the number of steps per commutation or the step interval time is increased, the motor speed will decrease. Hence, if the commutation step number is increased, the time interval between steps must be reduced to keep the motor speed the same. The most obvious way to increase the performance of the motors is to decrease the time interval between steps (within reason, since the motor may not be able to provide enough torque at higher speeds). Unfortunately this leads to the problem of limitations to the microcontroller – the motor is incremented with each execution of the Motor Controller task, such that shortening the timing duration is limited by the processor clock speed. This will be discussed in more detail in Chapter 6.

To recap, it was discussed that 3-phase DC motor actuation is performed in incremental steps where the frequency of the voltages delivered to the windings is proportional to the PWM duty cycle. Three sets of the PWM signals are delivered to the H-bridge motor driver that delivers higher voltage to the motor windings, since it is impossible for the microcontroller to deliver the necessary power as a stand-alone device. The PWM values follow the analog equivalent of 3-phase sinusoidal waveforms, such that each time step corresponds to a commutation step; by generating PWM outputs that follow the 3-phase sinusoidal pattern, the motor is allowed to microstep, or give the effect of smoother motor rotation for its given number of poles. Now we relate these concepts to task design.

It was previously discussed that the PWM frequency does not play a role in controlling motor speed. Rather, it is the frequency of the sinusoidal DC voltages that are delivered to the motor windings that control how quickly the motor rotates. Recall from the task example (Section 5.2.1) describing two simultaneously rotating motors, where the motor task must be written in such a way that the motor positions are incremented each time that it is executed. By varying the time steps of the task execution cycles, the periodicity of the sinusoidal voltages is proportionally varied. For example, if the timing requirement of the Motor Controller task were 1ms, and the motor controller is written to have a 48 step commutation sequence, the frequency of the sinusoidal voltages is $f = \frac{1}{1ms * 48 \text{ steps}} = 20.83Hz$. If the task requirement were changed to 2ms, the frequency of the sinusoidal voltages is halved, or 10.42Hz – the result of this is that the motor speed becomes halved.

However, it is impractical to vary the timing requirement of the Motor Controller task in real-time, since this will inevitably affect functions that run inside the task, as well as other tasks (i.e. Read IMU and Read HMD tasks). Instead, the inclusion of a counter that is native to the task is more practical to implement. The design of an internal counter exploits the timing requirement of the Motor Controller task – that is, the counter can only increment as quickly as the timing requirement for the Motor Controller task. The counter is reset when it reaches a certain threshold value, which is also the time that the motor is incremented by one step in the commutation sequence – this value can be modified independently of the counter, similar to the way that the Output Compare register is modified to change the duty cycle of PWM signals. By increasing this threshold value, the motor is slowed down. For example, if the threshold value is 2, the motor is slowed down to half its speed; this provides the same functionality as the previously discussed example, where the task timing is varied – but instead, the internal counter eliminates the need to dynamically change the task timing. When no event is triggered (the timer has not reached its threshold value), the counter continues to be incremented each time that the task is executed.

One of the main drawbacks to this method is that the task timing must be sufficiently low so that a wider range of frequencies can be produced. For example, if the Motor Controller task timing is set to 1ms, such that the fastest motor speed is synched to the task timing (that is, 1ms to produce 20.83Hz DC sinusoidal voltages, for a 24 pole motor, where $rpm = \frac{120f}{n_p} = 104.15$), the next possible slowest speed, where the threshold value is 2, resulting in half the motor speed, or 52.08rpm. Alternatively, if the timing is set to 0.1, the motor speed does not change as dramatically when the threshold is increased.

However, lowering the task requirement requires faster hardware – the prototype developed for this thesis did not account for this at the time of implementation, and thus a newer revision is in order for future work; this will be described in further detail in Chapter 7.

It was discussed that the inclusion of an internal counter that is native to the task is what is required for the motor speed to vary, and that the counter threshold value is inversely proportional to motor speed. Now, we move on to the discussion of setting the motor speed based on position error.

Error is simply the absolute difference between the motor’s goal position (the position that the motor should move to) and the motor’s actual position (where it is currently stationed):

(5.2.4.2.10)

$$err_x = goal\ position_x - actual\ position_x, \text{ where}$$

x denotes that the equation corresponds to the x – axis.

Section 5.2.3 thoroughly described the methods for collecting YPR data, in which the HMD coordinates represent the avatar’s goal position, while the IMU and encoder represent the avatar’s actual position. The basis of this thesis is to create a control system that actuates the mechanical gimbal such that the HMD and IMU coordinates are equal at all times. While there are many control methods that are available to try, the Proportional-Integral-Derivative (PID) controller was selected for this thesis due to its simplicity, reliability and its prevalent deployment in similar applications (i.e. FPV camera stabilization).

The PID controller is broken into three parts:

- The Proportional controller can be loosely described as “present-time error correction” such that the calculated error is multiplied by some proportional gain constant. As the error term is reduced, the gain is proportionally reduced.

(5.2.4.2.11)

$$K_p = \rho \times err(t), \text{ where}$$

K_p = the resultant proportional gain,

ρ = the proportional gain constant.

- The Integrator integrates error as a function of time to quickly increase the gain, and also behaves to remove steady-state error (if oscillation occurs at the goal position, the error will cancel each other out after some time).

(5.2.4.2.12)

$$K_I = I \times \int_0^t err(t) dt, \text{ where}$$

K_I = the resultant integrator gain,

I = the integrator gain constant,

t_2 = the current time,

t_1 = time at which err began,

- The Derivative controller finds the instantaneous rate of change of error to roughly estimate gain based on how quickly the system is changing.

(5.2.4.2.13)

$$K_d = \delta \times \frac{d}{dt} \text{err}(t), \text{ where}$$

K_d = the resultant derivative gain,

δ = the derivative gain constant.

The summation of the three controllers creates the PID controller:

(5.2.4.2.14)

$$G = K_p + K_i + K_d, \text{ where}$$

G = PID gain.

Figure 35 illustrates the complete PID control diagram:

Figure 35. PID Controller Diagram – The PID control system, showing the HOG-1 components.

When the PID controller is integrated to the Motor Controller task, several modifications are made, given that the system is discrete (time). Limitations must be placed on the gain and error values due to limitations on hardware (i.e. sensor and motor resolution, etc.). The continuous-time equations can be replaced by their simplified discrete-time counterparts where the time difference is assumed to be constant (due to task timing).

$$(5.2.4.2.15)$$

$$K_p = \rho \times err$$

$$(5.2.4.2.16)$$

$$K_I = I \times \sum_{t1}^{t2} err$$

$$(5.2.4.2.17)$$

$$K_d = \delta \times |current\ position - previous\ position|, \text{ where}$$

err = the error at the current time of measurement.

The most obvious difference between the continuous and discrete equations is that the time component for the derivative controller has been completely removed; the equation was intentionally modified in this way, since it is assumed that the task containing the PID controller runs in constant time increments, such that time difference can be compensated by adjusting the derivative gain variable, δ . Also, the integrator is modified in such a way that the gain is limited to a predefined value – like the integral gain variable, I , the limit requires tuning to prevent overshoot, while still providing a good balance of steady state correction.

Each motor is independently controlled via its own PID controller, since each axis of the gimbal is constrained by its own set of physical parameters – that is, each motor actuates a separate mechanical load (for example, the Yaw-axis motor is tasked with moving the largest load, since it must move majority of the gimbal components). Via trial and error, the proportional, integral and derivative gains can be modified until desirable performance is achieved (i.e. the motors reach their target destination in the quickest time without overshoot or oscillatory behavior). Although hardware is ultimately still a limiting factor to the performance of the entirety of the system, the PID functions to optimize the way that the hardware performs. The software implementation for the PID controller can be found in the Appendix E.

To stop the motor, or hold it in its current position, the commutation sequence is frozen, such that the 3-phase sinusoidal output voltages are held constant and delivered to the motor windings for the duration that the motor is decidedly stopped. When the motor is directed to move again, the motor continues where it left off in the commutation sequence.

5.2.4.3 MOTOR CONTROLLER IMPLEMENTATION

This section relates the calculated PID gain to software integration, and discusses other methods incorporated into software add improvements to the system by compensating for sensor-actuator resolution mismatch. In addition, the overall algorithm for the Motor Controller task will be presented here – however, majority of code for the Motor Controller task will not be discussed in this section, but may instead be found in the Appendix E.

In the previous section, the PID controller was introduced as the control method to be implemented in the Motor Controller task to allow the motor speed to vary depending on

YPR error. Recall that the motor speed is based on the threshold of a counter that is native to the Motor Controller task (in this case, three counters are used for each motor), such that the period of the commutation sequence (which relates directly to the period of the sinusoidal output voltage) is elongated when the threshold is increased (Section 5.2.4.1). To relate PID gain to the counter threshold, Equation 5.2.4.3.1 is implemented into software:

(5.2.4.3.1)

$$\text{Counter Threshold} = \text{round}\left(\frac{REF}{G}\right), \text{ where}$$

$$REF = \text{sensitivity factor},$$

$$G = \text{PID gain}.$$

It is common in motor control systems that the gain directly modifies the PWM duty cycle, where a larger gain corresponds to a higher duty. However, the control method presented in this thesis is more or less eccentric, because the gain is inversely proportional

to the period that relates directly to task timing. The inclusion of the REF variable as shown in equation 5.2.4.3.1 is necessary, because the counter threshold must be positive integer-valued. As it would seem, the selection of the REF variable is mostly arbitrary with a few considerations – some extreme cases are presented to highlight the significance of this variable:

- If REF is set to a value of 1, the motor can only be set with one of two possible counter threshold values: 0 or 1. In this case, the motor is only allowed two speeds.
- IF REF is set to too large of a value, the motor speed becomes less sensitive to the PID gain, G .

It was determined that the REF value is optimal when it is set to a value that closely relates to the maximum value that G can attain. Much like the components of the PID controller [15], the REF variable requires tuning until the system is optimized.

When a motor reaches its goal position, differences in motor and sensor resolution can induce oscillatory behavior. For example, if a motor has a step resolution of 1 degree/step, and an encoder is spec'd to have 0.2 degrees between positions, the motor will never reach a goal position of 0.2 degrees, but will instead, oscillate back and forth between the 0 and 1 degree positions. To remedy this issue, an allowable error (or, error allowance), based on the resolution and performance of the system is incorporated. The selection of allowable error values is based on a few considerations:

- The *resolution* of the least precise actuator/sensor, where the sensor is used to measure that actuator, can be used to determine allowable error for that control system.
- *Tunable allowable error* relates to how “good enough” the system performs, based on what is observed. In other words, a system designer asks himself whether or not the error noticeably impacts performance. This method is useful for when compromises must be made between precision and other performance parameters. Consider a motor that is coupled to an encoder, such that the speed at which the motor reaches its destination is of paramount importance; additionally, motor oscillation is to be completely avoided. While it is known that higher motor speeds result in some oscillatory behavior, however, it can also be eliminated by increasing the allowable error.

Given from the motor specifications in Chapter 4, and the number of time steps per period in each sinusoid, the motor resolution for those used in this project is given by:

(5.2.4.3.2)

$$\frac{24 \text{ poles}}{2} \times \frac{48 \text{ steps}}{\text{commutation sequence}} = \frac{576 \text{ steps}}{\text{revolution}}$$

Relating to number of degrees per revolution,

(5.2.4.3.3)

$$\frac{\text{revolution}}{576 \text{ step}} \times \frac{360 \text{ degrees}}{\text{revolution}} = \frac{0.625 \text{ degrees}}{\text{step}}$$

Of the motor/sensor pairs (one for each axis), the motor's step resolution is lower, with exception for the Yaw axis, where the 8-bit absolute encoder was used in place of the IMU's magnetometer. Calculating the resolution of the encoder in degrees:

(5.2.4.3.4)

$$\frac{360 \text{ degrees}}{256 \text{ positions}} = \frac{1.41 \text{ degrees}}{\Delta \text{position}}$$

After the prototype was complete, the error allowance for each gimbal axis was tuned for performance using a combination of the methods previously discussed. Now that each of the critical parameters and components of the Motor Controller task have been described, the Motor Controller task is presented in its entirety; a high level state diagram depicting the Motor Controller task is shown in Figure 5.2.4.3:

Figure 36. The Motor Controller Task State Diagram for Yaw Axis Motor Control – For the complete system, this algorithm is replicated for the Pitch and Roll axis motors.

When the Motor Controller task is executed, it begins by reading the shared variables that contain the HMD and IMU YPR coordinates (recall that the Read HMD and Read IMU tasks write the YPR data into their respective shared variables). Next, each error term is calculated by taking the difference of the avatar's coordinate (IMU) from the goal coordinate (HMD). Using the error terms and measured IMU coordinates, the PID gain is

calculated for each axis motor. The PID gain variables are divided by the sensitivity factor variable, REF, to determine the counter threshold values; the threshold value determines the timing at which the motor (corresponding to that counter) is incremented, and can change instantaneously with each task cycle, depending on the value of the PID gain. If an internal counter reaches the threshold value, the motor corresponding to that counter will either:

- Move clockwise if the HMD position variable is greater than the IMU position variable, and if the IMU position is within the pre-defined software limits to prevent the avatar from moving past its physical limits. If in this state, the counter variable is reset.
- Move counter-clockwise if the HMD position variable is less than the IMU position variable, and if the IMU position is within the pre-defined software limits to prevent the avatar from moving past its physical limits. If in this state, the counter variable is reset.
- Stop if the error value less than the allowable error value. If in this state, the counter variable is reset.
- Otherwise, the counter is incremented and nothing occurs; this means that the motor's response is delayed for at least one task cycle to slow down its rotation.

Not shown in Figure 36, this algorithm pertains to each gimbal, so that each motor is controlled independently using different parameters (PID gain constants), and error allowances.

6 RESULTS, ANALYSIS AND CONCLUSIONS

This chapter discusses testing and performance measures for the implementation described by the previous three chapters, and consists of four major topics: device complications, quantitative analysis, qualitative study and viability as a mobile device.

6.1 DEVICE COMPLICATIONS

As a first-generation prototype, problems were introduced relating to the selection of hardware, and thus complications exist to the overall performance of the system. These problems attribute to less reliable data collection, and so an improved method for quantitative measure is needed in addition to substantial hardware changes to the prototype, which will be discussed in detail in Chapter 7.

A thorough study of Chapter 5 reveals issues related to incompatible timing between tasks; recall that the timing requirement for the Motor Controller task is specified for 0.1ms, and the Read HMD and Read IMU tasks are specified for 4ms timing. An inherent problem with the Read HMD and Read IMU tasks is that the execution times are dependent upon the speed of data transfer between devices, viz., baud rate of asynchronous serial ports. Because the Read HMD and Read IMU tasks require a longer execution time than the timing requirement of the Motor Controller task, the Motor Controller task will inevitably be blocked during data transfer. The amount of time between Motor Controller task run cycles cannot be precisely determined, because length of the Y#P#R#E data string is unknown – that is, each positional coordinate can vary between 2 and 5 digits, depending on the device's measured position.

The Pitch-axis motor of the HOG-1 performs the best out of the three, but this is expected, due to the least amount of loading on the motor. The Pitch-axis motor is responsible for rotating the IMU and camera; the Roll-axis motor rotates the Pitch-axis motor and its load; and Yaw-axis motor rotates the entirety of the mechanical gimbal. The Yaw-axis motor experiences the most loading, and thus it requires more torque to rotate. When the PID gain is high such that the motor is electrically driven to reach its destination more quickly, the Yaw-axis motor cannot sustain its torque at high speeds, and instead experiences stalling. Hence, the PID gain for the Yaw-axis motor was decremented via trial and error such that the value is limited to allow the motor to provide enough torque to move the gimbal. While still functional, the resultant speed degrades the Yaw-axis tracking performance, which will be discussed in the following section.

6.2 QUANTITATIVE ANALYSIS

Ideally, measurement of the IMU and HMD YPR coordinates must be taken at exactly the same time to produce precise error calculation. Moreover, it is also necessary to have consistent sampling rates to determine lag times – the input-output (HMD-IMU) response times can then be accurately measured. Specifically relating to the prototype of this thesis, it is impossible to accurately and precisely measure error due to inconsistent task timing and slower than ideal serial transfer speeds. However, this thesis still attempts to construct a method to estimate error to test the device as a proof of concept.

The IMU and HMD coordinates are captured by the HOG-1 controller board and must be extracted for observance and analysis. Using an available serial port, the data can be transferred to a PC in real time, but with some repercussions – when the controller

transmits data to the PC, it takes valuable time that will cause the task timing to elongate; by sending data via serial port, the entire response of the system will change by an amount that can effectively ruin the experiment. Given few options available, this method is used to measure the performance of the avatar's head-tracking capabilities. By pure empirical observance, however, no noticeable loss of performance was humanly detectable as the prototype underwent testing.

The write-to-serial method is implemented into the Motor Controller task, where both IMU and HMD data are simultaneously used to control motor actuation. It seems most sensible then, to implement the testing feature into this task. One test consists of three iterations, where the data for each axis are independently collected. Using a serial terminal from a PC, the axis coordinates can be viewed in real-time.

A primitive test was designed to observe the avatar's head-tracking performance for three typical cases:

- In *slow mode*, the HMD is rotated slowly about the axis of interest to allow the avatar to move with it.
- In *normal operation*, the HMD is placed on the user's head, and the user is asked to look around. This test serves to measure error based on how the HOG-1 system is intended to be used.
- In *fast mode*, the HMD is shaken vigorously to see how well the avatar can track the HMD under the worst cases; given that a few of the motors are not to spec, the avatar is expected to perform poorly under this test.

Figures 37, 38 and 39 show the typical results of the HOG-1's head-tracking capabilities using the methods described above, using *15-30-15 test*. The 15-30-15 test can be described as a sequence of tests – for the first 15 seconds, the avatar is tested under slow mode, followed by 30 seconds in normal operation and, lastly, tested for 15 seconds in fast mode.

Figure 37. Yaw Axis Tracking: 15-30-15 Test Results.

Figure 38. Pitch Axis Tracking: 15-30-15 Test Results.

Figure 39. Roll Axis Tracking: 15-30-15 Test Results.

The average error per sample is highly dependent upon how the device is used. Hence, each of the test cases must be analyzed individually to draw fair conclusions. In the

subsections in the following section, the sample results shown in the Figures 37, 38 and 39 are to be deconstructed into their respective test cases.

6.2.1 ANALYSIS OF YAW-AXIS TRACKING AND ERROR ALLOWANCE

A first glance at Figure 40 shows that the avatar position follows the HMD; recall that the HMD coordinates are the inputs to the controller, whereas the avatar mimics the position as a response. Ideally the positions should overlap directly on top of each other, but this is impossible, because the avatar cannot instantaneously respond to the HMD; processing of the controller is required to appropriate a response to the actuators, which takes time.

Figure 40. Yaw Axis Tracking: Slow Mode

It can be observed from Figure 40 that the time separation between the two positions are distinctly separated by some time – by inference, the average error for each sample pair is higher. However, this separation is also caused by the allowed error value for

this axis, which was set to a value of ± 4 degrees (the concept of allowed error was discussed in Chapter 5). The error allowance for this axis was set to be relatively high, but was deemed necessary to reduce oscillation and overshoot; specific to this axis, the motor carries a large load where the momentum pushes the motor past its goal position, and so the allowed error is increased to compensate. Equation 6.2.3.1 is used to compute the average error, which also takes into account error allowances:

$$(6.2.3.1)$$

$$e(t) = |position(t)_{hmd} - position(t)_{imu}| - EA, \text{ where}$$

$$EA = \text{error allowance}$$

$$(6.2.3.2)$$

$$E(t) = \begin{cases} e(t), & e(t) > 0 \\ 0, & e(t) \leq 0 \end{cases}$$

$$(6.2.3.3)$$

$$Avg_{err} = \frac{\sum_{t=0}^T E(t)}{T}, \text{ where}$$

$$T = \text{number of samples.}$$

The mode, error allowance and average error are compared in Table 5, found at the end of this section.

Figure 41. Yaw Axis Tracking: Normal Operation

Figure 42. Yaw Axis Tracking: Fast Mode

Yaw Axis Average Error for Various Test Cases and Allowed Error		
Mode	Average Error (degrees)	Allowed Error (degrees)
Slow	5.21	±4
Normal	6.24	±4
Fast	15.78	±4

Table 5. Yaw Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Yaw axis.

Figures 41 and 42, and the data shown in Table 5, demonstrate that the avatar is less capable of tracking as the HMD is rotated more quickly. This is expected, because the motor cannot provide the necessary torque to actuate the gimbal at the speeds that are required for high performance. An interesting observation from the YPR trends is that the Yaw data is much less noisy than the data for the other axes. This is caused by the 8-bit encoder which has less precision than that of the IMU, which makes measurement of the device's position less sensitive to small movements. Additionally, the encoder is much less susceptible to environmental noise, whereas the other axes rely on the IMU's accelerometer and gyroscope. The HMD's Yaw data is gathered via the device's internal magnetometer, which is processed via computer [12]; the source code for the algorithm is not available to the public, so one can only assume that a great deal of filtering is added to the software to produce data with such little noise.

6.2.2 ANALYSIS OF PITCH-AXIS TRACKING

Figures 43, 44 and 45 show the 15-30-15 test results segmented into their respective modes.

Figure 43. Slow Mode Test Results for Tracking of the Pitch Axis.

Figure 44. Normal Operation test results for tracking of the Pitch axis.

Figure 45. Fast Mode test results for tracking of the Pitch axis.

Pitch Axis Average Error for Various Test Cases and Allowed Error		
Mode	Average Error (degrees)	Allowed Error (degrees)
Slow	2.43	± 1
Normal	3.56	± 1
Fast	5.44	± 1

Table 6. Pitch Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Pitch axis.

Qualitatively speaking, the Pitch axis performs the best of the three axes; this is expected, since the Pitch-axis motor experiences the least amount of mechanical loading, and is thus able to run at higher speeds. Given that the motor is able to operate at spec, the allowed error for this axis is reduced to ± 1 degree for improved precision and operates well under this constraint.

6.2.3 ANALYSIS OF ROLL-AXIS TRACKING

Figures 46, 47 and 48 show the 15-30-15 test results segmented into their respective modes.

Figure 46. Slow Mode test results for tracking of the Roll axis.

Figure 47. Slow Operation test results for tracking of the Roll axis.

Figure 48. Fast Mode test results for tracking of the Roll axis.

Roll Axis Average Error for Various Test Cases and Allowed Error		
Mode	Average Error (degrees)	Allowed Error (degrees)
Slow	8.56	± 1
Normal	3.56	± 1
Fast	12.32	± 1

Table 7. Roll Axis Average Error for Various Test Cases and Allowed Error – Calculated errors for each mode of the 15-30-15 test case for the Roll axis.

Like the Yaw-axis motor, the Roll-axis motor experiences mechanical loading that does not allow it to travel at ideal speeds with sufficient torque for high performance. Additionally, the Roll axis suffers from vibration as the motor rotates at higher speeds, such that the PID gain requires limiting. One unique artifact from this dataset is that the average error is higher for the slow mode test case, versus normal operation. This is accounted for by the fact that humans do not typically rotate their head side-to-side when looking around. This will be discussed in further detail in the next section.

6.3 QUALITATIVE STUDY

Much of the reasoning behind performance loss is described in terms of hardware and mechanical limitations of the avatar in the previous section. In this section related to qualitative analysis, *functionality* is used as qualitative descriptor of how well the prototype performs as a Telepresence device; *usability* describes the practicality of the device in various applications. This section brings light to the functionality and usability of the prototype in its current state based on empirical observation.

The Pitch axis motor, by an extreme amount, performs better than the others, and confirms that the performance analysis via qualitative data has some correlation to usability. The Pitch axis actuation performs well enough, where only minor improvements can be made to enhance its functionality. The Yaw and Roll axis, however, are unable to

perform well due to hardware limitations of the prototype, and thus a second iteration of the prototype is required for further analysis.

From the data shown in the previous section, there is much noise for the HMD and IMU coordinate positions. However, error allowance effectively filters out unnecessary actuator movements, where the noise does not vitally detract from the functionality of the device. To increase the precision of the system, error allowance should be minimized; by decreasing the error allowance, additional filtering is required for the HMD and IMU outputs.

A strange phenomenon for the Roll axis data set is that the average error measured for the slow mode test is substantially greater than that of the device tested in normal operation. Recalling that the PID gain for the Roll motor was intentionally limited to a value that does not allow the motor to run at spec (due to issues related to vibration), the motor cannot track the HMD as effectively. In the Roll axis slow mode test, the HMD was rotated from side to side in a greater range of motion than that of the device tested under normal operation; hence, the calculated error is higher. The data tested under normal conditions (and based on user experience) show that humans do not typically rotate their head about the roll axis as much as they do about the yaw and pitch axes when looking around.

Another important performance consideration related to the device's functional performance is vision latency, also known as *motion-to-photon* latency. Motion-to-photon latency describes the time difference between the user's movements and what he actually sees. It is well known within virtual reality community that a low motion-to-photon latency less than 20ms is required to create the illusion of presence. While this topic is beyond the

scope of this thesis, it plays a critical role in the functional test of the prototype, and thus a more robust vision system is required to make fair testing methods related to functionality.

Users who tried the device typically experienced motion sickness, where the onset time depended on how well the user was previously acquainted with using virtual reality devices. Typical complaints were made about the high actuator response times related to the Yaw axis rotation, and mechanical vibrations induced by the Roll axis motor. Additionally, the webcam used for this project exhibits high motion-to-photon latency that is greater than 100ms.

6.4 VIABILITY AS A MOBILE DEVICE

The HOG-1 running at full load was measured to consume an average of 17 watts of power; the prototype was supplied 10V, where an average of 1.7A was used to drive the controller, motors, encoder and IMU.

A typical car battery is specified for 12V, and has a nominal rating of 70 amp-hours. Assuming that a voltage regulator is used to step-down the voltage (2V at 1.7A), a conservative estimate can be made for the length of the time that the prototype can operate under full load conditions when powered by a car battery:

$$\frac{70Ahr}{1.7A} = 41.18 \text{ hours}$$

The size of the prototype is relatively small and is measured to be 5.5" x 5.5" x 10" (Length, Width, Height) and weighs under 5lbs. Due to its size, weight and low power consumption, the HOG-1 is a viable option to be mounted on mobile Telepresence robots.

7 FUTURE WORK

This chapter discusses improvements and remedies to the problems associated with the current prototype that are to be implemented in a second-iteration prototype. Also, a new method for data collection is proposed to replace the current method.

7.1 SUMMARY OF IMPROVEMENTS FOR A SECOND ITERATION

As mentioned previously, one of the main device complications is incompatible task timing. To recap, the serial transfer rate between the PC, IMU and controller prevent the Motor Controller task from executing in consistent time intervals, which attributes to slower and inconsistent motor speeds. For optimum performance, the IMU Read and HMD Read tasks must have execution times that are below the Motor Controller task timing requirement, such that the serial transfer must be sufficiently high.

It was determined after implementation and testing, that the baud rate cannot be increased to the ideal speeds for the ATmega2560 microcontroller, which was selected for this prototype. Using Equation 5.1.2.4, the required baud rate can found by using back substitution:

$$\text{Letting } T_s = 0.1ms \text{ and } n_s = 220 \frac{\text{bits}}{\text{set YPR data}},$$

$$BR = \frac{n_s}{0.5T_s} = \frac{\frac{220 \text{ bits}}{\text{set YPR data}}}{0.5 \times 10^{-3}s} = \frac{4.4Mb}{s}$$

The required baud rate for optimal performance is too high given the amount of data that needs to be transmitted. Thus it becomes necessary to reduce the size of the data stream. From qualitative testing, it was conservatively determined that humans are unable

to detect less than 1 degree precision, so the YPR data stream can be reduced to a total of three digits, or purely integer values to represent the angular coordinates. With this assumption, the data can be encoded into binary form, such that each axis can be represented by two bytes: one byte to hold a possible negative sign, and another byte to represent an unsigned integer data-type. Thus, the total number of bits that represent the #Y#P#R#E stream is calculated to be:

$$n_s = 3 \text{ signs} * \frac{10 \text{ bits}}{\text{sign}} + \frac{10 \text{ bits}}{\text{unsigned int}} \times 3 \text{ axes} + 4 \text{ delimiters} \times \frac{10 \text{ bits}}{\text{delimiter}} = \frac{100 \text{ bits}}{\text{set YPR data}}$$

The new required baud rate for optimum performance is given by:

$$BR = \frac{n_s}{0.5T_s} = \frac{\frac{100 \text{ bits}}{\text{set YPR data}}}{0.5 \times 10^{-3} s} = \frac{2Mb}{s}$$

This new spec still cannot be implemented on the current prototype due to limitations of the microcontroller, and so a new microcontroller is required; a wide variety of superior microcontrollers are available for purchase – for example, the ARM Cortex-M7 boasts a maximum baud rate of 12.5Mb/s with a clock frequency set to 100MHz [14].

In addition, the IMU utilizes the ATmega328 for onboard processing; this presents the same consequences as the ATmega2560. Thus, a new IMU, or a custom-made one is called for in the next prototype iteration.

Another drawback to the current device is that the Yaw and Roll motors cannot supply enough torque to actuate the gimbal to the desired speeds. Hence, replacement

motors are required – the specs for these motors will not be listed here since more testing is required to determine the necessary specs; however, it should be mentioned that since the Yaw motor selection is not limited by its weight, an alternative to the 3-phase DC brushless motor can be used – for example, stepper motors are heavier with the benefit of increased torque, which can be sensibly implemented to actuate the Yaw axis, but are less ideal for the Roll and Pitch axes due to their size and weight.

7.2 AN IMPROVED TESTING METHOD

Testing the of the avatar’s ability to track the HMD was a flawed procedure in this thesis, given that serial port communication increases the execution time of the Motor Controller task. However, with an improved second-iteration prototype that consists of faster hardware, the YPR coordinates of the HMD and avatar can be precisely measured given that the timing of the tasks are consistently executed in a timely manner. Even with these assumptions, writing data to the serial port still requires time to transmit, which skews the avatar’s tracking ability.

Instead, with an improved method, positional coordinates are saved to the device’s memory within the Motor Controller task to be retrieved after some time. In this method, the data is saved to the microcontroller while the device is in operation; writing the coordinate values into an array takes a negligible amount of time, and so the tracking speed is not altered. When some (predefined) number of samples has been collected, the device transmits the data to the PC for analysis. Figure 7.2.1 shows a high level diagram for this improved testing method.

Figure 49. Diagram for an Improve Test Method – High level diagram of the proposed testing procedure to be implemented in the Motor Controller task.

The time interval between HMD and IMU coordinate pairs is known, since this method relies solely on the Motor Controller task’s timing. Given that the timing is precise, more analysis can be made relating to the time separation between equal-valued points to construct generalizations about input-output latency, but will not be explored in this project because it is currently a fruitless venture for the first revision prototype.

7.3 OTHER CONSIDERATIONS

The camera used for this thesis did not account for motion-to-photon latency; an improved vision system is necessary to create the sensation of presence. More research is required to determine camera specs for the next prototype.

Additionally, more research is needed to determine a method for low latency data transmission for long range communication, the distance between the avatar and the user to be extended.

7.4 FINAL REMARKS

In summary, a second iteration of the prototype that improves upon the basis of this thesis is required to be constructed using the described methods within this section for further study. The implementation of this second prototype will allow the use of the improved testing method, which will allow for more accurate and precise data collection; ideally, this procedure will serve as a foundation for testing all future implementations that use the same mechatronic approach.

BIBLIOGRAPHY

- [1] Ats Kurvet, "Oculus Rift development kit 2," Image, Wikimedia Commons, October, 5, 2014. [Online]. Available:
https://commons.wikimedia.org/wiki/File:Oculus_Rift_development_kit_2.jpg. Accessed [June, 2, 2015].
- [2] Atmel Corporation, Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V datasheet, 2014.
- [3] Bradley, D. A., D. Dawson, N. C. Burd, and A. J. Loader. "What Is Mechatronics?" *Mechatronics* (1991): 1-15.
- [4] Cameron, A.A. "Visor Projected Helmet Mounted Displays Technology and Applications." *Microprocessors and Microsystems* 22.8 (1999): 465-75.
- [5] Georgiou, Evangelos, Peter L. Morgan, Jonathan San Diego, Jian Dai, and Michael Luck. "Self Localization Using a 9DOF IMU Sensor With a Directional Cosine Matrix." Volume 4: 18th Design for Manufacturing and the Life Cycle Conference; 2013 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (2013).
- [6] Grossman, G.e., R.j. Leigh, L.a. Abel, D.j. Lanska, and S.e. Thurston. "Frequency and Velocity of Rotational Head Perturbations during Locomotion." *Exp Brain Res Experimental Brain Research* 70.3 (1988).
- [7] Gwon, Arthur, "2014-02-08 Stereoscapy - Toronto Islands - 08 Ward Island Village," Image, Wikimedia Commons, February, 8, 2014. [Online]. Available:
http://commons.wikimedia.org/wiki/File:2014-02-08_Stereoscapy_-_Toronto_Islands_-_08_Ward_Island_Village.jpg. Accessed [June, 2, 2015].
- [8] Koyo Electronics Corporation Ltd., TRD-NA Series datasheet, 2014.
- [9] Kanigoro, Bayu, Widodo Budiharto, Jurike V. Moniaga, and Mushin Shodiq. "WEB BASED CONFERENCE SYSTEM FOR INTELLIGENCE TELEPRESENCE ROBOT: A FRAMEWORK." *Journal of Computer Science* 10.1 (2014): 10-14. 14 Oct. 2014.
- [10] Kratz, Sven, Jim Vaughan, Ryota Mizutani, and Don Kimber. "Evaluating Stereoscopic Video with Head Tracking for Immersive Teleoperation of Mobile Telepresence Robots." *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction Extended Abstracts - HRI'15 Extended Abstracts* (2015).

- [11] Livatino, S., G. Muscato, and F. Privitera. "Stereo Viewing and Virtual Reality Technologies in Mobile Robot Teleguide." *IEEE Transactions on Robotics* IEEE Trans. Robot. 25.6 (2009): 1343-355.
- [12] Oculus VR, Oculus Rift Developer Guide Version 0.6.0.0, 2015.
- [13] Ridgely, John R. "Using Tasks and Finite State Machines in C++." MS. California Polytechnic State University.
- [14] STMicroelectronics, STM32F756xx datasheet, 2014.
- [15] Wescott, Tim. "PID Without a PhD." Embedded 09 May 2006.

A: FULL SCHEMATIC OF CONTROLLER BOARD

B: PC CLIENT C++ CODE

```
#include "OVR_CAPI.h"
#include "Kernel\OVR_Math.h"
#include <iostream>
#include <conio.h>
#include <windows.h>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <atlstr.h>
#include <string>
using namespace OVR;
using namespace std;
ovrHmd hmd;
ovrFrameTiming frameTiming;
void Init()
{
    ovr_Initialize();

    hmd = ovrHmd_Create(0);

    if(!hmd) return;

    ovrHmd_ConfigureTracking(hmd, ovrTrackingCap_Orientation |
ovrTrackingCap_MagYawCorrection | ovrTrackingCap_Position, 0);
}

void Clear()
{
    ovrHmd_Destroy(hmd);

    ovr_Shutdown();
}

void Output()
{
    int16_t yaw_int;
    int16_t pitch_int;
    int16_t roll_int;
    char yaw_ser[7];
    char pitch_ser[7];
    char roll_ser[7];

    //Serial Setup
    CString PortSpecifier="COM8";
    DCB dcb = {0};
    DWORD byteswritten;
    HANDLE hPort = CreateFile(
        PortSpecifier,
        GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        0,
        NULL
    );

    dcb.BaudRate = CBR_9600; //9600 Baud
    dcb.ByteSize = DATABITS_8; //8 data bits
    dcb.Parity = NOPARITY; //no parity
    dcb.StopBits = ONESTOPBIT; //1 stop

    if (SetCommState(hPort, &dcb) == 0)
    {
        cout << endl << "ERROR: CANNOT CONFIGURE PORT!" << endl << endl;
        CloseHandle(hPort);
    }
}
```

```

// Optional: we can overwrite the previous console to more
// easily see changes in values
HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO bufferInfo;
GetConsoleScreenBufferInfo(h, &bufferInfo);

while(hmd) {

    frameTiming = ovrHmd_BeginFrameTiming(hmd, 0);
    ovrTrackingState ts = ovrHmd_GetTrackingState(hmd,
        frameTiming.ScanoutMidpointSeconds);

    if (ts.StatusFlags & (ovrStatus_OrientationTracked |
        ovrStatus_PositionTracked)) {
        // The cpp compatibility layer is used to convert ovrPosef to Posef (see
        // OVR_Math.h)
        Posef pose = ts.HeadPose.ThePose;
        float ax = ts.RawSensorData.Accelerometer.x;
        float yaw, pitch, roll;
        float yaw_ang, pitch_ang, roll_ang;
        pose.Rotation.GetEulerAngles<Axis_Y, Axis_X, Axis_Z>(&yaw, &pitch,
            &roll);

        // Optional: move cursor back to starting position and print values
        SetConsoleCursorPosition(h, bufferInfo.dwCursorPosition);

        yaw_ang = RadToDegree(yaw);
        pitch_ang = RadToDegree(pitch);
        roll_ang = RadToDegree(roll);

        yaw_int = yaw_ang * 100;
        pitch_int = pitch_ang * 100;
        roll_int = roll_ang * 100;

        cout << endl << "yaw: " << yaw_ang << endl;
        cout << "pitch: " << pitch_ang << endl;
        cout << "roll: " << roll_ang << endl;

        //Convert Data to ASCII strings and place into buffer variables.
        itoa(yaw_int, yaw_ser, 10);
        itoa(pitch_int, pitch_ser, 10);
        itoa(roll_int, roll_ser, 10);

        //Data sent with delimiters in format: Y#P#R#E
        WriteFile(hPort, "Y", 1, &byteswritten, NULL);
        WriteFile(hPort, yaw_ser, 5, &byteswritten, NULL);
        WriteFile(hPort, "P", 1, &byteswritten, NULL);
        WriteFile(hPort, pitch_ser, 5, &byteswritten, NULL);
        WriteFile(hPort, "R", 1, &byteswritten, NULL);
        WriteFile(hPort, roll_ser, 5, &byteswritten, NULL);
        WriteFile(hPort, "E", 1, &byteswritten, NULL);
        //Sleep(50);
    }

    ovrHmd_EndFrameTiming(hmd);
    if (_kbhit()) exit(0);
}
CloseHandle(hPort); //close the handle
}

int main()
{
    Init();
    Output();
    Clear();
    return 0;
}

```

C: VISION PROCESSOR PYTHON 2.7 CODE

```
import numpy as np
import cv2

width = 1920
height = 1080

left_center_col = width/4
right_center_col = 3*left_center_col

capture = cv2.VideoCapture(0) #Search for available webcam and assign to object.
capture.set(5, 30) #Change framerate to 30 frames/s

while(True):

    ret, frame = capture.read()
    ret = capture.set(4, height)
    ret = capture.set(3, width)

    roi = np.copy(frame[0:1080, left_center_col:right_center_col])

    frame[0:roi.shape[0], 0:roi.shape[1]] = roi
    frame[0:roi.shape[0], roi.shape[1]:2*roi.shape[1]] = roi

    cv2.imshow('HMD Output', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

capture.release()
cv2.destroyAllWindows()
```

Class Documentation

absencoder_driver Class Reference

This class enables the encoder values to be read from the GPIO pins.

```
#include <absencoder_driver.h>
```

Public Member Functions

- **absencoder_driver** (emstream *p_serial_port, volatile uint8_t *DDR_input, volatile uint8_t *PIN_input, uint8_t bit0, uint8_t bit1, uint8_t bit2, uint8_t bit3, uint8_t bit4, uint8_t bit5, uint8_t bit6, uint8_t bit7)
This class enables the encoder values to be read from the GPIO pins via the read_value method.
- void **read_value** (void)
The read_value method reads the GPIO pins to determine the value of the encoder. The encoder value is used as an index for the look up table to find the corresponding angle.

Protected Attributes

- emstream * **ptr_to_serial**
 - uint8_t **BIT0**
 - uint8_t **BIT1**
 - uint8_t **BIT2**
 - uint8_t **BIT3**
 - uint8_t **BIT4**
 - uint8_t **BIT5**
 - uint8_t **BIT6**
 - uint8_t **BIT7**
 - volatile uint8_t * **PINn**
 - int16_t **encoder_lut_angle** [256]
 - uint16_t **i**
-

Detailed Description

This class enables the encoder values to be read from the GPIO pins.

This driver is built for an absolute encoder that has 8 bit resolution, which uses 8 GPIO pins that read each bit.

Parameters:

<i>ptr_to_serial</i>	The my_motor_driver class uses this pointer to the serial port to print desired data
<i>DDR_input</i>	initializes the data direction register for input.
<i>PIN_input</i>	initializes the PIN register for fetch input.
<i>bit0</i>	the zeroth bit of the encoder.
<i>bit1</i>	the first bit of the encoder.
<i>bit2</i>	the second bit of the encoder.
<i>bit3</i>	the third bit of the encoder.
<i>bit4</i>	the fourth bit of the encoder.
<i>bit5</i>	the fifth bit of the encoder.
<i>bit6</i>	the sixth bit of the encoder.
<i>bit7</i>	the seventh bit of the encoder.

Definition at line 58 of file absencoder_driver.h.

Constructor & Destructor Documentation

```
absencoder_driver::absencoder_driver (emstream * p_serial_port, volatile uint8_t *  
DDR_input, volatile uint8_t * PIN_input, uint8_t bit0, uint8_t bit1, uint8_t bit2, uint8_t  
bit3, uint8_t bit4, uint8_t bit5, uint8_t bit6, uint8_t bit7)
```

This class enables the encoder values to be read from the GPIO pins via the `read_value` method.

This driver is built for an absolute encoder that has 8 bit resolution, which uses 8 GPIO pins that read each bit.

Parameters:

<i>ptr_to_serial</i>	The my_motor_driver class uses this pointer to the serial port to print desired data
<i>DDR_input</i>	initializes the data direction register for input.
<i>PIN_input</i>	initializes the PIN register for fetch input.
<i>bit0</i>	the zeroth bit of the encoder.
<i>bit1</i>	the first bit of the encoder.
<i>bit2</i>	the second bit of the encoder.
<i>bit3</i>	the third bit of the encoder.
<i>bit4</i>	the fourth bit of the encoder.
<i>bit5</i>	the fifth bit of the encoder.
<i>bit6</i>	the sixth bit of the encoder.
<i>bit7</i>	the seventh bit of the encoder.

Definition at line 46 of file `absencoder_driver.cpp`.

Member Function Documentation

`void absencoder_driver::read_value (void )`

The `read_value` method reads the GPIO pins to determine the value of the encoder. The encoder value is used as an index for the look up table to find the corresponding angle. The method reads from 8 GPIO - one for each of the bits for the 8 bit encoder. The method also converts the bit readings from a greyscale to binary value, then uses this as an index to determine the angular position from the look up.

Definition at line 91 of file `absencoder_driver.cpp`.

References `imu_yaw`.

The documentation for this class was generated from the following files:

- **`absencoder_driver.h`**
- **`absencoder_driver.cpp`**

motor_driver Class Reference

This class should run the motor driver on an AVR processor.

```
#include <motor_driver.h>
```

Public Member Functions

- **motor_driver** (emstream *p_serial_port, volatile uint8_t *DDR_PWM, uint8_t PWMPIN1, uint8_t PWMPIN2, uint8_t PWMPIN3, volatile uint16_t *PWM1, volatile uint16_t *PWM2, volatile uint16_t *PWM3)
This constructor initializes the motor driver.
- void **move** (int8_t dir)
This method uses the previously saved motor configuration and determines the next configuration in the sequence to allow the motor to move. A look up table that estimates the PWM values to build a sinewave is generated in the constructor.
- void **stop** (void)
This method uses the previously saved motor configuration and outputs those same values to the PWM output pins. This freezes the motor to the current state and prevents it from moving; this essentially provides a braking function for the motor.
- void **PID_yaw** (int16_t position_error, int16_t position_y)
This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).
- void **PID_pitch** (int16_t position_error, int16_t position_p)
This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).
- void **PID_roll** (int16_t position_error, int16_t position_r)
This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).

Protected Attributes

- emstream * **ptr_to_serial**
- volatile uint16_t * **pwm1**
- volatile uint16_t * **pwm2**
- volatile uint16_t * **pwm3**
- uint8_t **pwmSin** [48]
- uint8_t **position**
- uint8_t **currentStep1**
- uint8_t **currentStep2**

- `uint8_t currentStep3`
- `uint32_t iState_y`
- `uint32_t iState_p`
- `uint32_t iState_r`
- `int16_t dState_y`
- `int16_t dState_p`
- `int16_t dState_r`

Detailed Description

This class should run the motor driver on an AVR processor.

This class allows the use of 3-phase brushless DC motors to be interfaced to the controller.

Parameters:

<i>ptr_to_serial</i>	The my_motor_driver class uses this pointer to the serial port to print desired data
<i>DDR_PWM</i>	initializes the data direction register for the motor
<i>PWMPIN1</i>	sets the motor pin corresponding to phase 1.
<i>PWMPIN2</i>	sets the motor pin corresponding to phase 2.
<i>PWMPIN3</i>	sets the motor pin corresponding to phase 3.
<i>PWM1</i>	the output PWM control register for PWMPIN1.
<i>PWM2</i>	the output PWM control register for PWMPIN2.
<i>PWM3</i>	the output PWM control register for PWMPIN3.

Definition at line 54 of file motor_driver.h.

Constructor & Destructor Documentation

```
motor_driver::motor_driver (emstream * p_serial_port, volatile uint8_t * DDR_PWM,  
uint8_t PWMPIN1, uint8_t PWMPIN2, uint8_t PWMPIN3, volatile uint16_t * PWM1,  
volatile uint16_t * PWM2, volatile uint16_t * PWM3)
```

This constructor initializes the motor driver.

The motor driver outputs three PWM signals to drive a three phase motor. The **move()** method takes the last saved position and determine the next value in the in the sequence via look up table. This class also includes a **stop()** method, that outputs the last saved PWM settings so that the motors are frozen to its current state. Lastly, this class includes PID control (**PID_yaw()** , **PID_pitch()** , **PID_roll**) for each of the three motors which control the frequency of in the motor task.

Parameters:

<i>ptr_to_serial</i>	The my_motor_driver class uses this pointer to the serial port to print desired data
<i>DDR_PWM</i>	initializes the data direction registers for the motor
<i>PWMPIN1,the</i>	input of H-bridge chip corresponding to the phase 1 output to the motor.
<i>PWMPIN2,the</i>	input of H-bridge chip corresponding to the phase 2 output to the motor.
<i>PWMPIN3,the</i>	input of H-bridge chip corresponding to the phase 3 output to the motor.
<i>PWM1,the</i>	PWM control register for PWMPIN1.
<i>PWM2,the</i>	PWM control register for PWMPIN2.
<i>PWM3,the</i>	PWM control register for PWMPIN3.

Definition at line 51 of file motor_driver.cpp.

Member Function Documentation

`void motor_driver::move (int8_t dir)`

This method uses the previously saved motor configuration and determines the next configuration in the sequence to allow the motor to move. A look up table that estimates the PWM values to build a sinewave is generated in the constructor.

The brake function sets the mode select bits A & B to the same value.

Parameters:

<i>dir</i>	dictates the direction that the motor should move.
------------	--

Definition at line 85 of file motor_driver.cpp.

`void motor_driver::PID_pitch (int16_t position_error, int16_t position_p)`

This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).

This method writes the PID gain for the pitch motor and stores it into a shared variable to be passed to the motor control task.

Parameters:

<i>position_error</i>	The positional error - the absolute value of the difference of the input and desired position readings.
<i>position_p</i>	The current position that is read from the IMU.

Definition at line 201 of file motor_driver.cpp.

`void motor_driver::PID_roll (int16_t position_error, int16_t position_r)`

This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).

This method writes the PID gain for the roll motor and stores it into a shared variable to be passed to the motor control task. This axis suffers the most from instability issues related to vibration. For higher gains at the center point, the motor has a tendency to oscillate.

Parameters:

<i>position_error</i>	The positional error - the absolute value of the difference of the input and desired position readings.
<i>position_r</i>	The current position that is read from the IMU.

Definition at line 247 of file motor_driver.cpp.

```
void motor_driver::PID_yaw (int16_t position_error, int16_t position_y)
```

This method determines the PID gain based on position error, current position, and last position (protected class variable to drive derivative control).

This method writes the PID gain for the yaw motor and stores it into a shared variable to be passed to the motor control task.

Parameters:

<i>position_error</i>	The positional error - the absolute value of the difference of the input and desired position readings.
<i>position_y</i>	The current position that is read from the IMU.

Definition at line 158 of file motor_driver.cpp.

The documentation for this class was generated from the following files:

- **motor_driver.h**
- **motor_driver.cpp**

motor_task Class Reference

This task controls the direction and speed of the motor given the sensor readings of the HMD and IMU. The purpose of this task to allow the IMU to track the HMD coordinates by way of motor control.

```
#include <motor_task.h>
```

Public Member Functions

- **motor_task** (const char *, unsigned portBASE_TYPE, size_t, emstream *)
- void **run** (void)

This method utilizes the positional data collected from the Read HMD and Read IMU tasks and provides headtracking functionality via motor control. As the HMD is moved, the motors move to correct the IMU position by matching the angular position of the HMD. Additionally, the PID gains are adjusted with each run cycle by using the current error and current position.

Detailed Description

This task controls the direction and speed of the motor given the sensor readings of the HMD and IMU. The purpose of this task to allow the IMU to track the HMD coordinates by way of motor control.

The motor driver is run using a driver in files **motor_driver.h** and **motor_driver.cpp** . Definition at line 62 of file motor_task.h.

Constructor & Destructor Documentation

motor_task::motor_task (const char * a_name, unsigned portBASE_TYPE a_priority, size_t a_stack_size, emstream * p_ser_dev)

(frt_task); the parent's constructor.

Parameters:

<i>a_name</i>	A character string which will be the name of this task
<i>a_priority</i>	The priority at which this task will initially run (default: 0)
<i>a_stack_size</i>	The size of this task's stack in bytes (default: configMINIMAL_STACK_SIZE)
<i>p_ser_dev</i>	Pointer to a serial device (port, radio, SD card, etc.) which can be used by this task to communicate (default: NULL)

Definition at line 35 of file motor_task.cpp.

Member Function Documentation

void motor_task::run (void)

This method utilizes the positional data collected from the Read HMD and Read IMU tasks and provides headtracking functionality via motor control. As the HMD is moved, the motors move to correct the IMU position by matching the angular position of the HMD. Additionally, the PID gains are adjusted with each run cycle by using the current error and current position.

This method utilizes the **motor_driver** class to control 3-phase brushless DC motors.

Definition at line 59 of file motor_task.cpp.

References imu_yaw.

The documentation for this class was generated from the following files:

- **motor_task.h**
- **motor_task.cpp**

readhmd_task Class Reference

This task receives data from the serial port, which correspond to HMD coordinates.

```
#include <readhmd_task.h>
```

Public Member Functions

- **readhmd_task** (const char *, unsigned portBASE_TYPE, size_t, emstream *)
- void **run** (void)
This method reads from the serial port one character at a time, and detects the start delimiter, 'Y' to begin the data-read process. When this method is called, it waits for the delimiter to guarantee that data will be collected from the iteration. The data stream is stored into a buffer, then converted from ASCII to numeric values to be stored in the shared variable line to be used by the motor task.
- int **atoi** (char digit)
This function converts numeric ASCII characters to numeric integer values.

Detailed Description

This task receives data from the serial port, which correspond to HMD coordinates.

The absolute encoder driver is run using a driver in files **absencoder_driver.h** and **absencoder_driver.cpp** .

Definition at line 58 of file readhmd_task.h.

Constructor & Destructor Documentation

readhmd_task::readhmd_task (const char * a_name, unsigned portBASE_TYPE

a_priority, size_t a_stack_size, emstream * p_ser_dev)

(frt_task); the parent's constructor.

Parameters:

<i>a_name</i>	A character string which will be the name of this task
<i>a_priority</i>	The priority at which this task will initially run (default: 0)
<i>a_stack_size</i>	The size of this task's stack in bytes (default: configMINIMAL_STACK_SIZE)
<i>p_ser_dev</i>	Pointer to a serial device (port, radio, SD card, etc.) which can be used by this task to communicate (default: NULL)

Definition at line 35 of file readhmd_task.cpp.

Member Function Documentation

int readhmd_task::atoi (char digit)

This function converts numeric ASCII characters to numeric integer values.

Parameters:

<i>digit,a</i>	char-type variable that stores an ASCII character.
----------------	--

Definition at line 226 of file readhmd_task.cpp.

Referenced by run().

The documentation for this class was generated from the following files:

- **readhmd_task.h**
- **readhmd_task.cpp**

readimu_task Class Reference

This task receives data from the serial port, which correspond to IMU coordinates.

```
#include <readimu_task.h>
```

Public Member Functions

- **readimu_task** (const char *, unsigned portBASE_TYPE, size_t, emstream *)
- void **run** (void)
This method reads from the serial port one character at a time, and detects the start delimiter, 'Y' to begin the data-read process. When this method is called, it waits for the delimiter to guarantee that data will be collected from the iteration. The data stream is stored into a buffer, then converted from ASCII to numeric values to be stored in the shared variable line to be used by the motor task. Note that although the magnetometer readings are sent by the IMU and read by the microcontroller, they are not used; the magnetometer provides false readings when placed near motors. Instead, the absolute encoder driver is instantiated to determine Yaw readings.
- int **atoi** (char digit)
This function converts numeric ASCII characters to numeric integer values.

Detailed Description

This task receives data from the serial port, which correspond to IMU coordinates.
Definition at line 56 of file readimu_task.h.

Constructor & Destructor Documentation

readimu_task::readimu_task (const char * a_name, unsigned portBASE_TYPE a_priority, size_t a_stack_size, emstream * p_ser_dev)

(frt_task); the parent's constructor.

Parameters:

<i>a_name</i>	A character string which will be the name of this task
<i>a_priority</i>	The priority at which this task will initially run (default: 0)
<i>a_stack_size</i>	The size of this task's stack in bytes (default: configMINIMAL_STACK_SIZE)
<i>p_ser_dev</i>	Pointer to a serial device (port, radio, SD card, etc.) which can be used by this task to communicate (default: NULL)

Definition at line 37 of file readimu_task.cpp.

Member Function Documentation

int readimu_task::atoi (char digit)

This function converts numeric ASCII characters to numeric integer values.

Parameters:

<i>digit,a</i>	char-type variable that stores an ASCII character.
----------------	--

Definition at line 222 of file readimu_task.cpp.

Referenced by run().

The documentation for this class was generated from the following files:

- **readimu_task.h**
- **readimu_task.cpp**

File Documentation

absencoder_driver.cpp File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include <math.h>
#include <avr/interrupt.h>
#include "rs232int.h"
#include "absencoder_driver.h"
```

Detailed Description

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **absencoder_driver.cpp**.

absencoder_driver.h File Reference

```
#include "emstream.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "shares.h"
```

Classes

- class **absencoder_driver**

This class enables the encoder values to be read from the GPIO pins.

Detailed Description

This file contains an absolute encoder driver header file. This class only includes one method, `read_value`, that reads the inputs to the GPIO pins when at the instant the method is called.

Revisions:

- 01-15-2008 JRR Original (somewhat useful) file
- 10-11-2012 JRR Less original, more useful file with FreeRTOS mutex added
- 10-12-2012 JRR There was a bug in the mutex code, and it has been fixed
- 05-25-2015 Darren Chan - This code was completely re-written for absolute encoder-read functionality.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **absencoder_driver.h**.

motor_driver.cpp File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include <math.h>
#include <avr/interrupt.h>
#include "rs232int.h"
#include "motor_driver.h"
```

Detailed Description

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **motor_driver.cpp**.

motor_driver.h File Reference

```
#include "emstream.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "shares.h"
```

Classes

- class **motor_driver**

This class should run the motor driver on an AVR processor.

Detailed Description

This file contains a motor driver header file. This class provides functionality to three phase motors via a half H-bridge driver chip. The class includes the move, stop, and PID control methods.

Revisions:

- 01-15-2008 JRR Original (somewhat useful) file
- 10-11-2012 JRR Less original, more useful file with FreeRTOS mutex added
- 10-12-2012 JRR There was a bug in the mutex code, and it has been fixed
- 05-12-2015 Darren Chan - Completely re-written to provide functionality to control three phase motors via half H-bridge driver chip.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **motor_driver.h**.

motor_task.cpp File Reference

```
#include "frt_text_queue.h"  
#include "motor_driver.h"  
#include "motor_task.h"  
#include <util/delay.h>
```

Detailed Description

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **motor_task.cpp**.

motor_task.h File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include "FreeRTOS.h"
#include "queue.h"
#include "frt_task.h"
#include "time_stamp.h"
#include "frt_queue.h"
#include "frt_shared_data.h"
#include "rs232int.h"
#include "shares.h"
```

Classes

- class **motor_task**

This task controls the direction and speed of the motor given the sensor readings of the

HMD and IMU. The purpose of this task to allow the IMU to track the HMD coordinates by way of motor control.

Detailed Description

This file contains the header for a task class that controls motor operation based on IMU and HMD sensor readings. The sensors are first read for position coordinates, and error values are computed. The error values and current positions of each motor are passed into their respective PID methods to determine the frequency at which the motors should operate.

Revisions:

- 09-30-2012 JRR Original file was a one-file demonstration with two tasks
- 10-05-2012 JRR Split into multiple files, one for each task
- 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
- 10-27-2012 JRR Altered from data sending task into LED blinking class
- 11-04-2012 JRR Altered again into the multi-task monstrosity
- 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
- 05-25-2015 Darren Chan - Completely rewrote code for motor control.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **motor_task.h**.

readhmd_task.cpp File Reference

```
#include "frt_text_queue.h"  
#include "readhmd_task.h"  
#include <util/delay.h>
```

Detailed Description

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **readhmd_task.cpp**.

readhmd_task.h File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include "FreeRTOS.h"
#include "queue.h"
#include "frt_task.h"
#include "time_stamp.h"
#include "frt_queue.h"
#include "frt_shared_data.h"
#include "rs232int.h"
#include "shares.h"
```

Classes

- class **readhmd_task**

This task receives data from the serial port, which correspond to HMD coordinates.

Detailed Description

This file contains the header for a task class that reads the HMD coordinate values to decode them into numeric form.

Revisions:

- 09-30-2012 JRR Original file was a one-file demonstration with two tasks
- 10-05-2012 JRR Split into multiple files, one for each task
- 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
- 10-27-2012 JRR Altered from data sending task into LED blinking class
- 11-04-2012 JRR Altered again into the multi-task monstrosity
- 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
- 05-12-2015 Darren Chan - Completely re-written to provide functionality to read and decode characters from a serial port.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **readhmd_task.h**.

readimu_task.cpp File Reference

```
#include "frt_text_queue.h"  
#include "readimu_task.h"  
#include "absencoder_driver.h"  
#include <util/delay.h>
```

Detailed Description

This file contains the header for a task class that reads the IMU coordinate values to decode them into numeric form.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **readimu_task.cpp**.

readimu_task.h File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include "FreeRTOS.h"
#include "queue.h"
#include "frt_task.h"
#include "time_stamp.h"
#include "frt_queue.h"
#include "frt_shared_data.h"
#include "rs232int.h"
#include "shares.h"
```

Classes

- class **readimu_task**

This task receives data from the serial port, which correspond to IMU coordinates.

Detailed Description

This file contains the header for a task class that controls the motor PWM, direction and mode of operation. This task controls a motor driver.

Revisions:

- 09-30-2012 JRR Original file was a one-file demonstration with two tasks
- 10-05-2012 JRR Split into multiple files, one for each task
- 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
- 10-27-2012 JRR Altered from data sending task into LED blinking class
- 11-04-2012 JRR Altered again into the multi-task monstrosity
- 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
- 05-12-2015 Darren Chan - Completely re-written to provide functionality to read and decode characters from a serial port.

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **readimu_task.h**.

shares.h File Reference

```
#include "frt_queue.h"
#include "frt_text_queue.h"
#include "frt_shared_data.h"
```

Variables

- frt_text_queue **print_ser_queue**
- frt_queue< uint32_t > **queue_1**
- shared_data< int16_t > **imu_yaw**
- shared_data< int16_t > **imu_pitch**
- shared_data< int16_t > **imu_roll**
- shared_data< int16_t > **hmd_yaw**
- shared_data< int16_t > **hmd_pitch**
- shared_data< int16_t > **hmd_roll**
- shared_data< uint8_t > **inv_gain_yaw**
- shared_data< uint8_t > **inv_gain_pitch**
- shared_data< uint8_t > **inv_gain_roll**

Detailed Description

This file contains extern declarations for queues and other inter-task data communication objects used in a ME405/507/FreeRTOS project.

Revisions:

- 09-30-2012 JRR Original file was a one-file demonstration with two tasks
- 10-05-2012 JRR Split into multiple files, one for each task plus a main one
- 10-29-2012 JRR Reorganized with global queue and shared data references
- 5-15-2015 Darren Chan Added shared variables for communication between Read HMD, Read IMU and Motor Controller Tasks

License: This file is copyright 2012 by JR Ridgely and released under the Lesser GNU Public License, version 2. It intended for educational use only, but its use is not limited thereto.

Definition in file **shares.h**.

E: EMBEDDED C++ CODE (HOG-1)

```
//*****
/*  License:
 *      This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 *      Public License, version 2. It intended for educational use only, but its use
 *      is not limited thereto.*/
/*      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 *      AND      ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 *      IMPLIED      WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 *      ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 *      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 *      TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 *      OR SERVICES; LOSS OF USE, DATA
 *      , OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *      CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 *      OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 *      OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

#include <stdlib.h>                // Prototype declarations for I/O functions
#include <avr/io.h>                // Port I/O for SFR's
#include <avr/wdt.h>               // Watchdog timer header
#include <string.h>                // Functions for C string handling

#include "FreeRTOS.h"             // Primary header for FreeRTOS
#include "task.h"                 // Header for FreeRTOS task functions
#include "queue.h"                // FreeRTOS inter-task communication queues
#include "croutine.h"             // Header for co-routines and such

#include "rs232int.h"             // ME405/507 library for serial comm.
#include "time_stamp.h"           // Class to implement a microsecond timer
#include "frt_task.h"             // Header of wrapper for FreeRTOS tasks
#include "frt_text_queue.h"       // Wrapper for FreeRTOS character queues
#include "frt_queue.h"            // Header of wrapper for FreeRTOS queues
#include "frt_shared_data.h"      // Header for thread-safe shared data
#include "shares.h"               // Global ('extern') queue declarations
#include "motor_task.h"
#include "readimu_task.h"
#include "readhmd_task.h"

#include <util/delay.h>           // Delay

// Declare the queues which are used by tasks to communicate with each other here.
// Each queue must also be declared 'extern' in a header file which will be read
// by every task that needs to use that queue. The format for all queues except
// the serial text printing queue is 'frt_queue<type> name (size)', where 'type'
// is the type of data in the queue and 'size' is the number of items (not neces-
// sarily bytes) which the queue can hold

/** This is a print queue, descended from \c emstream so that things can be printed
 * into the queue using the "<<" operator and they'll come out the other end as a
 * stream of characters. It's used by tasks that send things to the user interface
 * task to be printed.
 */

frt_text_queue print_ser_queue (32, NULL, 10);

/** This queue sends data from the source task to the sink task (both of these tasks
 * have been removed in this revision).
 */

frt_queue<uint32_t> queue_1 (20);

/** This shared data item allows communication between the motor task and the task user.
```



```

*   In this way, we can make a user interface which can update the motor operation in
*   realtime as the tasks are called pseudo simultaneously in the RTOS.
*/
shared_data<int16_t> imu_yaw; // Yaw data read from the IMU via serial.
shared_data<int16_t> imu_pitch; // Pitch data read from the IMU via serial.
shared_data<int16_t> imu_roll; // Roll data read from the IMU via serial.

shared_data<int16_t> hmd_yaw; // Yaw data read from the HMD via serial.
shared_data<int16_t> hmd_pitch; // Pitch data read from the HMD via serial.
shared_data<int16_t> hmd_roll; // Roll data read from the HMD via serial.

shared_data<uint8_t> inv_gain_yaw; // Gain variables for PID controller.
shared_data<uint8_t> inv_gain_pitch;
shared_data<uint8_t> inv_gain_roll;

//=====
/** \brief The main function sets up the RTOS. Some test tasks are created. Then the
 * scheduler is started up; the scheduler runs until power is turned off or there's a
 * reset.
 * @return This is a real-time microcontroller program which doesn't return. Ever.
 */

int main (void)
{
    MCUSR = 0;
    wdt_disable();

    _delay_ms(5000);

    //Configure the serial ports for device communication - one for the PC, the
    //other for the IMU.

    hmd_yaw.put(0);
    hmd_roll.put(0);
    hmd_pitch.put(0);

    rs232_ser_port_pc (57600, 0);
    rs232_ser_port_imu (57600, 1);

    //Force baudrates of ports 0 and 1 to 567500

    UBRR0 = 16;
    UBRR1 = 16;

    //The task that reads IMU positional data from the serial port.
    new readimu_task ("ReadIMUTask", task_priority (1), 280, &ser_port_imu);

    //The task that read HMD positional data from the other serial port.
    new readhmd_task ("ReadHMDTask", task_priority (1), 280, &ser_port_pc);

    //The motor controller task.
    new motor_task ("MotorTask", task_priority (1), 280, &ser_port_imu);

    vTaskStartScheduler ();
}

```

```

//*****
/** \file shares.h
 * This file contains extern declarations for queues and other inter-task data
 * communication objects used in a ME405/507/FreeRTOS project.
 *
 * Revisions:
 * \li 09-30-2012 JRR Original file was a one-file demonstration with two tasks
 * \li 10-05-2012 JRR Split into multiple files, one for each task plus a main one
 * \li 10-29-2012 JRR Reorganized with global queue and shared data references
 * \li 5-15-2015 Darren Chan Added shared variables for communication between Read HMD,
 * Read IMU and Motor Controller Tasks
 *
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

// This define prevents this .h file from being included multiple times in a .cpp file
#ifndef _SHARES_H_
#define _SHARES_H_
#include "frt_queue.h"
#include "frt_text_queue.h"
#include "frt_shared_data.h"

//-----
// Externs: In this section, we declare variables and functions that are used in all
// (or at least two) of the files in the data acquisition project. Each of these items
// will also be declared exactly once, without the keyword 'extern', in one .cpp file
// as well as being declared extern here.

// This queue allows tasks to send characters to the user interface task for display.
//extern frt_text_queue print_ser_queue;

/* This shared data item allows a value to be posted by the source task and read by
 * the sink task.
 */
extern frt_text_queue print_ser_queue;

/* This queue sends data from the source task to the sink task.
 */
extern frt_queue<uint32_t> queue_1;

extern shared_data<int16_t> imu_yaw; // IMU Yaw data
extern shared_data<int16_t> imu_pitch; // IMU pitch data
extern shared_data<int16_t> imu_roll; // IMU roll data

extern shared_data<int16_t> hmd_yaw; // IMU Yaw data
extern shared_data<int16_t> hmd_pitch; // IMU pitch data
extern shared_data<int16_t> hmd_roll; // IMU roll data

extern shared_data<uint8_t> inv_gain_yaw; //PID Gain for the Yaw Axis
extern shared_data<uint8_t> inv_gain_pitch; //PID Gain for Pitch Axis
extern shared_data<uint8_t> inv_gain_roll; //PID Gain for Roll Axis

#endif // _SHARES_H_

```

```

//=====
/** \file absencoder_driver.h
 * This file contains an absolute encoder driver header file. This class only includes
 * one method, read_value, that reads the inputs to the GPIO pins when at the instant
 * the method is called.
 *
 * Revisions:
 * \li 01-15-2008 JRR Original (somewhat useful) file
 * \li 10-11-2012 JRR Less original, more useful file with FreeRTOS mutex added
 * \li 10-12-2012 JRR There was a bug in the mutex code, and it has been fixed
 * \li 05-25-2015 Darren Chan - This code was completely re-written for absolute
 * encoder-read functionality.
 *
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//=====

// This define prevents this .H file from being included multiple times in a .CPP file
#ifndef _absencoder_driver_H_
#define _absencoder_driver_H_

#include "emstream.h" // Header for serial ports and devices
#include "FreeRTOS.h" // Header for the FreeRTOS RTOS
#include "task.h" // Header for FreeRTOS task functions
#include "queue.h" // Header for FreeRTOS queues
#include "semphr.h" // Header for FreeRTOS semaphores
#include "shares.h"

//-----
/** \brief This class enables the encoder values to be read from the GPIO pins.
 * \details This driver is built for an absolute encoder that has 8 bit resolution, which
 * uses 8 GPIO pins that read each bit.
 * @param ptr_to_serial The my_motor_driver class uses this pointer to the serial port to
 * print desired data
 * @param DDR_input initializes the data direction register for input.
 * @param PIN_input initializes the PIN register for fetch input.
 * @param bit0 the zeroth bit of the encoder.
 * @param bit1 the first bit of the encoder.
 * @param bit2 the second bit of the encoder.
 * @param bit3 the third bit of the encoder.
 * @param bit4 the fourth bit of the encoder.
 * @param bit5 the fifth bit of the encoder.
 * @param bit6 the sixth bit of the encoder.
 * @param bit7 the seventh bit of the encoder.
 */

class absencoder_driver
{
protected:

    emstream* ptr_to_serial;

    //Numeric bit values.

    uint8_t BIT0, BIT1, BIT2, BIT3, BIT4, BIT5, BIT6, BIT7;

    //Pointer to the PIN input.

```

```

volatile uint8_t* PINn;

//Instantiate the LUT for the encoder.

int16_t encoder_lut_angle[256];
uint16_t i;

public:

absencoder_driver (emstream* p_serial_port, volatile uint8_t* DDR_input,
volatile uint8_t* PIN_input, uint8_t bit0, uint8_t bit1, uint8_t bit2, uint8_t
bit3, uint8_t bit4, uint8_t bit5, uint8_t bit6, uint8_t bit7);

//The read_value method reads the GPIO pins to determine the encoder
//position.

void read_value(void);

};

#endif // _AVR__absencoder_driver_H_

```

```

//*****
/** \file absencoder_driver.cpp

* License:
* This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
* Public License, version 2. It intended for educational use only, but its use
* is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
* TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

#include <stdlib.h> // Include standard library header files
#include <avr/io.h>
#include <math.h>
#include <avr/interrupt.h>

#include "rs232int.h" // Include header for serial port class
#include "absencoder_driver.h" // Include header for the A/D class

//-----
/** \brief This class enables the encoder values to be read from the GPIO pins via
* the \c read_value method.
* \details This driver is built for an absolute encoder that has 8 bit resolution, which
* uses 8 GPIO pins that read each bit.
* @param ptr_to_serial The my_motor_driver class uses this pointer to the serial port to
print desired data
* @param DDR_input initializes the data direction register for input.
* @param PIN_input initializes the PIN register for fetch input.
* @param bit0 the zeroth bit of the encoder.
* @param bit1 the first bit of the encoder.
* @param bit2 the second bit of the encoder.
* @param bit3 the third bit of the encoder.
* @param bit4 the fourth bit of the encoder.
* @param bit5 the fifth bit of the encoder.
* @param bit6 the sixth bit of the encoder.
* @param bit7 the seventh bit of the encoder.
*/

absencoder_driver::absencoder_driver (emstream* p_serial_port, volatile uint8_t* DDR_input,
volatile uint8_t* PIN_input, uint8_t bit0, uint8_t bit1, uint8_t bit2, uint8_t bit3, uint8_t
bit4, uint8_t bit5, uint8_t bit6, uint8_t bit7)
{
    ptr_to_serial = p_serial_port;

    //Set pointer variables to the pins.

    BIT0 = bit0;
    BIT1 = bit1;
    BIT2 = bit2;
    BIT3 = bit3;
    BIT4 = bit4;
    BIT5 = bit5;
    BIT6 = bit6;
    BIT7 = bit7;

    //Set the input register to read.

    PINn = PIN_input;

```

```

        //Sets the data direction register for input for the pins that will be used to read
the encoder.

        *DDR_input &= ~((1 << BIT0) & (1 << BIT1) & (1 << BIT2) & (1 << BIT3) & (1 << BIT4)
& (1 << BIT5) & (1 << BIT6) & (1 << BIT7));

        //Generate look up values for encoder value to angle conversion.

        encoder_lut_angle[129] = 0;
        encoder_lut_angle[0] = 17780;
        for(i = 130; i <= 255; i++){
            encoder_lut_angle[i] = 140*(i-129);
        }

        for(i = 128; i > 0; i--){
            encoder_lut_angle[i] = -140*(129-i);
        }

    }

//-----
---
/** \brief The read_value method reads the GPIO pins to determine the value of the encoder.
The
* encoder value is used as an index for the look up table to find the corresponding angle.
* \details The method reads from 8 GPIO - one for each of the bits for the 8 bit encoder.
* The method also converts the bit readings from a greyscale to binary value, then uses this
* as an index to determine the angular position from the look up.
*/

void absencoder_driver::read_value(void)
{
    uint8_t encoder_value = 0;
    uint16_t bit0_val, bit1_val, bit2_val, bit3_val, bit4_val, bit5_val, bit6_val,
bit7_val = 0;

    if(*PINn & (1 << BIT0))
    {
        bit0_val = 0;
    }
    else
    {
        bit0_val = 1;
    }
    if(*PINn & (1 << BIT1))
    {
        bit1_val = 0;
    }
    else
    {
        bit1_val = 1;
    }
    if(*PINn & (1 << BIT2))
    {
        bit2_val = 0;
    }
    else
    {
        bit2_val = 1;
    }
    if(*PINn & (1 << BIT3))
    {
        bit3_val = 0;
    }
    else
    {
        bit3_val = 1;
    }
}

```

```

        if(*PINn & (1 << BIT4))
        {
            bit4_val = 0;
        }
        else
        {
            bit4_val = 1;
        }

        if(*PINn & (1 << BIT5))
        {
            bit5_val = 0;
        }
        else
        {
            bit5_val = 1;
        }

        if(*PINn & (1 << BIT6))
        {
            bit6_val = 0;
        }
        else
        {
            bit6_val = 1;
        }

        if(*PINn & (1 << BIT7))
        {
            bit7_val = 0;
        }
        else
        {
            bit7_val = 1;
        }

        //Conversion from grey code to binary.

        bit6_val ^= bit7_val;
        bit5_val ^= bit6_val;
        bit4_val ^= bit5_val;
        bit3_val ^= bit4_val;
        bit2_val ^= bit3_val;
        bit1_val ^= bit2_val;
        bit0_val ^= bit1_val;

        //The encoder value serves as an index for the angle look up table.

        encoder_value = (bit7_val<<7) | (bit6_val<<6) | (bit5_val<<5) | (bit4_val<<4) |
        (bit3_val<<3) | (bit2_val<<2) | (bit1_val<<1) | (bit0_val<<0);

        imu_yaw.put(encoder_lut_angle[encoder_value]);
    }

```

```

//=====
/** \file motor_driver.h
 * This file contains a motor driver header file. This class provides functionality to
 * three phase motors via a half H-bridge driver chip. The class includes the move, stop,
 * and PID control methods.
 *
 * Revisions:
 * \li 01-15-2008 JRR Original (somewhat useful) file
 * \li 10-11-2012 JRR Less original, more useful file with FreeRTOS mutex added
 * \li 10-12-2012 JRR There was a bug in the mutex code, and it has been fixed
 * \li 05-12-2015 Darren Chan - Completely re-written to provide functionality to
 * control three phase motors via half H-bridge driver chip.
 *
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//=====

// This define prevents this .H file from being included multiple times in a .CPP file
#ifndef _motor_driver_H_
#define _motor_driver_H_

#include "emstream.h" // Header for serial ports and devices
#include "FreeRTOS.h" // Header for the FreeRTOS RTOS
#include "task.h" // Header for FreeRTOS task functions
#include "queue.h" // Header for FreeRTOS queues
#include "semphr.h" // Header for FreeRTOS semaphores

#include "shares.h"
//-----
/** \brief This class should run the motor driver on an AVR processor.
 * \details This class allows the use of 3-phase brushless DC motors to be interfaced to the
 * controller.
 * @param ptr_to_serial The my_motor_driver class uses this pointer to the serial port to
 * print desired data
 * @param DDR_PWM initializes the data direction register for the motor
 * @param PWMPIN1 sets the motor pin corresponding to phase 1.
 * @param PWMPIN2 sets the motor pin corresponding to phase 2.
 * @param PWMPIN3 sets the motor pin corresponding to phase 3.
 * @param PWM1 the output PWM control register for PWMPIN1.
 * @param PWM2 the output PWM control register for PWMPIN2.
 * @param PWM3 the output PWM control register for PWMPIN3.
 */

class motor_driver
{
protected:

    emstream* ptr_to_serial;

    volatile uint16_t* pwm1;
    volatile uint16_t* pwm2;
    volatile uint16_t* pwm3;

    //Memory to store the sinewave look up table.
    uint8_t pwmSin [48];

    //Position and phase variables to determine the next motor phase sequence.
    uint8_t position;

```



```

uint8_t currentStep1,currentStep2,currentStep3;

//PID state variables
uint32_t iState_y;
uint32_t iState_p;
uint32_t iState_r;
int16_t dState_y;
int16_t dState_p;
int16_t dState_r;

public:
    // The constructor sets up the motor driver. The "= NULL" part is a
    // default parameter, meaning that if that parameter isn't given on the line
    // where this constructor is called, the compiler will just fill in "NULL".
    // In this case that has the effect of turning off diagnostic printouts.
    // This follows for the "= 0" part.
    motor_driver (emstream* p_serial_port, volatile uint8_t* DDR_PWM, uint8_t
PWMPIN1, uint8_t PWMPIN2, uint8_t PWMPIN3, volatile uint16_t* PWM1, volatile uint16_t* PWM2,
volatile uint16_t* PWM3);

    void move(int8_t dir);

    void stop(void);

    void PID_yaw(int16_t position_error, int16_t position_y);

    void PID_pitch(int16_t position_error, int16_t position_p);

    void PID_roll(int16_t position_error, int16_t position_r);

};

#endif // _AVR_motor_driver_H_

```

```

//*****
/** \file motor_driver.cpp

* License:
* This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
* Public License, version 2. It intended for educational use only, but its use
* is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
* TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

#include <stdlib.h> // Include standard library header files
#include <avr/io.h>
#include <math.h>
#include <avr/interrupt.h>

#include "rs232int.h" // Include header for serial port class
#include "motor_driver.h" // Include header for the A/D class

//-----
/** \brief This constructor initializes the motor driver.
* \details The motor driver outputs three PWM signals to drive a three phase motor. The
* \c move() method takes the last saved position and determine the next value in the in
* the sequence via look up table. This class also includes a \c stop() method, that outputs
* the last saved PWM settings so that the motors are frozen to its current state. Lastly,
* this class includes PID control (\c PID_yaw(), \c PID_pitch(), \c PID_roll) for each of
* the three motors which control the frequency
* of in the motor task.
* @param ptr_to_serial The my_motor_driver class uses this pointer to the serial port to
print desired data
* @param DDR_PWM initializes the data direction registers for the motor
* @param PWMPIN1, the input of H-bridge chip corresponding to the phase 1 output to
* the motor.
* @param PWMPIN2, the input of H-bridge chip corresponding to the phase 2 output to
* the motor.
* @param PWMPIN3, the input of H-bridge chip corresponding to the phase 3 output to
* the motor.
* @param PWM1, the PWM control register for PWMPIN1.
* @param PWM2, the PWM control register for PWMPIN2.
* @param PWM3, the PWM control register for PWMPIN3.
*/

//Initialize the motor driver.
motor_driver::motor_driver (emstream* p_serial_port, volatile uint8_t* DDR_PWM, uint8_t
PWMPIN1, uint8_t PWMPIN2, uint8_t PWMPIN3, volatile uint16_t* PWM1, volatile uint16_t* PWM2,
volatile uint16_t* PWM3)
{
    ptr_to_serial = p_serial_port;
    position = 0;

    //Pass the pointers to protected class variables.
    pwm1 = PWM1;
    pwm2 = PWM2;
    pwm3 = PWM3;

    *DDR_PWM |= (1 << PWMPIN1) | (1 << PWMPIN2) | (1 << PWMPIN3); //Set the pwm pins for
output (3 per motor).

    //Initialize the Sinewave Look Up.

```

```

        pwmSin[0] = 127;pwmSin[1] = 110;pwmSin[2] = 94;pwmSin[3] = 78;pwmSin[4] = 64;pwmSin[5]
= 50;pwmSin[6] = 37;pwmSin[7] = 26;pwmSin[8] = 17;
        pwmSin[9] = 10;pwmSin[10] = 4;pwmSin[11] = 1;pwmSin[12] = 0;pwmSin[13] = 1;pwmSin[14]
= 4;pwmSin[15] = 10;pwmSin[16] = 17;pwmSin[17] = 26;
        pwmSin[18] = 37;pwmSin[19] = 50;pwmSin[20] = 64;pwmSin[21] = 78;pwmSin[22] =
94;pwmSin[23] = 110;pwmSin[24] = 127;pwmSin[25] = 144;
        pwmSin[26] = 160;pwmSin[27] = 176;pwmSin[28] = 191;pwmSin[29] = 204;pwmSin[30] =
217;pwmSin[31] = 228;pwmSin[32] = 237;pwmSin[33] = 244;
        pwmSin[34] = 250;pwmSin[35] = 253;pwmSin[36] = 254;pwmSin[37] = 253;pwmSin[38] =
250;pwmSin[39] = 244;pwmSin[40] = 237;pwmSin[41] = 228;
        pwmSin[42] = 217;pwmSin[43] = 204;pwmSin[44] = 191;pwmSin[45] = 176;pwmSin[46] =
160;pwmSin[47] = 144;

        //Initialize the integrator state of the PI controller to 0.
        iState_y = 0;
        iState_p = 0;
        iState_r = 0;
}

//-----
/** \brief This method uses the previously saved motor configuration and determines the
 * next configuration in the sequence to allow the motor to move. A look up table that
 * estimates the PWM values to build a sinewave is generated in the constructor.
 * \details The brake function sets the mode select bits A & B to the same value.
 * @param dir dictates the direction that the motor should move.
 */

void motor_driver::move(int8_t dir)
{
    //Move the motor clockwise.
    if (dir > 0){
        if (position >= 47){
            position = 0;
        }
        else{
            position++;
        }
        if (position >= 32){
            currentStep2 = position - 32;
        }
        else{
            currentStep2 = position + 16;
        }
        if (position >= 16){
            currentStep3 = position - 16;
        }
        else{
            currentStep3 = position + 32;
        }
    }

    //Move the motor counter-clockwise.
    else if (dir < 0){
        if (position <= 0){
            position = 47;
        }
        else{
            position--;
        }
        if (position < 32){
            currentStep2 = position + 16;
        }
        else{
            currentStep2 = position - 32;
        }
        if (position < 16){
            currentStep3 = position + 32;
        }
        else{
            currentStep3 = position - 16;
        }
    }
}

```

```

        }
    }
    currentStep1 = position;
    *pwm1 = pwmSin[currentStep1];
    *pwm2 = pwmSin[currentStep2];
    *pwm3 = pwmSin[currentStep3];
}

/** \brief This method uses the previously saved motor configuration and outputs those
 * same values to the PWM output pins. This freezes the motor to the current state and
 * prevents it from moving; this essentially provides a braking function for the motor.
 */

void motor_driver::stop(void)
{
    *pwm1 = pwmSin[currentStep1];
    *pwm2 = pwmSin[currentStep2];
    *pwm3 = pwmSin[currentStep3];
}

/** \brief This method determines the PID gain based on position error, current position,
 * and last position (protected class variable to drive derivative control).
 * \details This method writes the PID gain for the yaw motor and stores it into a shared
 * variable to be passed to the motor control task.
 * @param position_error The positional error - the absolute value of the difference
 * of the input and desired position readings.
 * @param position_y The current position that is read from the IMU.
 */

void motor_driver::PID_yaw(int16_t position_error, int16_t position_y)
{
    //Proportional error variables
    double pGain = 0.004;
    double pTerm = pGain*abs(position_error);

    //Integral error variables
    double iGain = 0.000000002;
    uint32_t iMax = 45000000;
    uint8_t iMin = 0;
    double iTerm = 0.0;

    //Integral gain term.
    iState_y += position_error;

    if( iState_y > iMax )
    {
        iState_y = iMax;
    }
    else if( iState_y < iMin )
    {
        iState_y = iMin;
    }

    iTerm = iGain * iState_y;

    //Derivative error variables
    double dGain = 0.001;
    double dTerm = dGain * (position_y - dState_y);
    dState_y = position_y; //Update derivative state

    inv_gain_yaw.put(abs(round(pTerm + dTerm + iTerm)));
}

/** \brief This method determines the PID gain based on position error, current position,
 * and last position (protected class variable to drive derivative control).
 * \details This method writes the PID gain for the pitch motor and stores it into a shared
 * variable to be passed to the motor control task.
 * @param position_error The positional error - the absolute value of the difference
 * of the input and desired position readings.

```

```

* @param position_p The current position that is read from the IMU.
*/

void motor_driver::PID_pitch(int16_t position_error, int16_t position_p)
{
    //Proportional error variables
    double pGain = 0.03;
    double pTerm = pGain*abs(position_error);

    //Integral error variables
    double iGain = 0.00000004;
    uint32_t iMax = 45000000;
    uint8_t iMin = 0;
    double iTerm;

    //Integral gain term.
    iState_p += position_error;

    if( iState_p > iMax )
    {
        iState_p = iMax;
    }
    else if( iState_p < iMin )
    {
        iState_p = iMin;
    }

    iTerm = iGain * iState_p;

    //Derivative error variables
    double dGain = 0.00001;
    double dTerm = dGain * (position_p - dState_p);
    dState_p = position_p; //Update derivative state

    inv_gain_pitch.put(abs(round(pTerm + dTerm + iTerm)));
}

/** \brief This method determines the PID gain based on position error, current position,
 * and last position (protected class variable to drive derivative control).
 * \details This method writes the PID gain for the roll motor and stores it into a shared
 * variable to be passed to the motor control task. This axis suffers the most from instability
 * issues related to vibration. For higher gains at the center point, the motor has a tendency
 * to oscillate.
 * @param position_error The positional error - the absolute value of the difference
 * of the input and desired position readings.
 * @param position_r The current position that is read from the IMU.
 */

void motor_driver::PID_roll(int16_t position_error, int16_t position_r)
{
    //Proportional error variables.
    double pGain = 0.004;
    double pTerm = pGain*abs(position_error);

    //Integral error variables.
    double iGain = 0.00000004;
    uint32_t iMax = 4500000;
    uint8_t iMin = 0;
    double iTerm;

    //Integral gain term.
    iState_r += position_error;

    if( iState_r > iMax )
    {
        iState_r = iMax;
    }
    else if( iState_r < iMin )
    {

```

```

        iState_r = iMin;
    }

    iTerm = iGain * iState_r;

    //Derivative gain.
    double dGain = 0.00001;

    //Derivative error variables
    double dTerm = dGain * (position_r - dState_r);
    dState_r = position_r; //Update derivative state

    inv_gain_roll.put(abs(round(pTerm + dTerm + iTerm)));
}

```

```

//*****
/** \file motor_task.h
 * This file contains the header for a task class that controls motor operation based
 * on IMU and HMD sensor readings. The sensors are first read for position coordinates,
 * and error values are computed. The error values and current positions of each motor
 * are passed into their respective PID methods to determine the frequency at which the
 * motors should operate.
 * Revisions:
 * \li 09-30-2012 JRR Original file was a one-file demonstration with two tasks
 * \li 10-05-2012 JRR Split into multiple files, one for each task
 * \li 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
 * \li 10-27-2012 JRR Altered from data sending task into LED blinking class
 * \li 11-04-2012 JRR Altered again into the multi-task monstrosity
 * \li 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
 * \li 05-25-2015 Darren Chan - Completely rewrote code for motor control.
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

// This define prevents this .h file from being included multiple times in a .cpp file
#ifndef _MOTOR_TASK_H_
#define _MOTOR_TASK_H_

#include <stdlib.h> // Prototype declarations for I/O functions
#include <avr/io.h> // Header for special function registers

#include "FreeRTOS.h" // Primary header for FreeRTOS
#include "queue.h" // FreeRTOS inter-task communication queues

#include "frt_task.h" // ME405/507 base task class
#include "time_stamp.h" // Class to implement a microsecond timer
#include "frt_queue.h" // Header of wrapper for FreeRTOS queues
#include "frt_shared_data.h" // Header for thread-safe shared data
#include "rs232int.h" // ME405/507 library for serial comm.

#include "shares.h" // Global ('extern') queue declarations

//-----
/** \brief This task controls the direction and speed of the motor given the sensor readings
 * of the HMD and IMU. The purpose of this task to allow the IMU to track the HMD coordinates
 * by way of motor control.
 * \details The motor driver is run using a driver in files \c motor_driver.h and
 * \c motor_driver.cpp.
 */
class motor_task : public frt_task
{
private:
    // No private variables or methods for this class.
protected:
    // No protected variables or methods for this class.
public:
    // This constructor creates a task for controlling motors.
    motor_task(const char*, unsigned portBASE_TYPE, size_t, emstream*);
    // This method is called by the RTOS once to run the task loop.
    void run(void);
};

#endif

```

```

//*****
/** \file motor_task.cpp
 *
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****
#include "ftr_text_queue.h"           // Header for text queue class
#include "motor_driver.h"
#include "motor_task.h"               // Header for this task
#include <util/delay.h>

//-----
/**
 * (\c ftr_task ); the parent's constructor.
 * @param a_name A character string which will be the name of this task
 * @param a_priority The priority at which this task will initially run (default: 0)
 * @param a_stack_size The size of this task's stack in bytes
 *                      (default: configMINIMAL_STACK_SIZE)
 * @param p_ser_dev Pointer to a serial device (port, radio, SD card, etc.) which can
 *                  be used by this task to communicate (default: NULL)
 */

motor_task::motor_task (const char* a_name,
                        unsigned portBASE_TYPE
a_priority,
                        size_t a_stack_size,
                        emstream* p_ser_dev
                        )
    : ftr_task (a_name, a_priority, a_stack_size, p_ser_dev)
{
    p_serial = p_ser_dev;

    // Initialize the runs counter
    runs = 0;
}

//-----
/** \brief This method utilizes the positional data collected from the Read HMD and Read
 * IMU tasks and provides headtracking functionality via motor control. As the HMD is
 * moved, the motors move to correct the IMU position by matching the angular position of
 * the HMD. Additionally, the PID gains are adjusted with each run cycle by using the
 * current error and current position.
 *
 * \details This method utilizes the \c motor_driver class to control 3-phase brushless DC
 * motors.
 */

void motor_task::run (void)
{
    portTickType previousTicks = xTaskGetTickCount ();

    //Enable and configure timers.
    //Set for FASTPWM mode, with no timing prescaling. Lowering the frequency
    //generates an annoying high pitch noise. Trial and testing revealed that
    //the H-bridge driver can run at this high frequency with no performance loss.

```



```

TCCR1A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);
TCCR1B |= (1 << WGM12) | (1 << CS10);
TCCR3A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);
TCCR3B |= (1 << WGM12) | /*(1 << CS11)*/ (1 << CS10);
TCCR4A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1) | (1 << COM1C1);
TCCR4B |= (1 << WGM12) | /*(1 << CS11)*/ (1 << CS10);;

//Create motor objects

motor_driver* yaw_motor = new motor_driver (p_serial, &DDRB, PB5, PB6, PB7, &OCR1A,
&OCR1B, &OCR1C);
motor_driver* pitch_motor = new motor_driver (p_serial, &DDRE, PE3, PE4, PE5, &OCR3A,
&OCR3B, &OCR3C);
motor_driver* roll_motor = new motor_driver (p_serial, &DDRH, PH3, PH4, PH5, &OCR4A,
&OCR4B, &OCR4C);

//Initialize all the sensor readings to zero.

int16_t hmd_yaw_temp = 0;
int16_t hmd_pitch_temp = 0;
int16_t hmd_roll_temp = 0;
int16_t imu_yaw_temp = 0;
int16_t imu_pitch_temp = 0;
int16_t imu_roll_temp = 0;
uint8_t counter_yaw = 0;
uint8_t counter_roll = 0;
uint8_t counter_pitch = 0;
int16_t yaw_error = 0;
int16_t pitch_error = 0;
int16_t roll_error = 0;
uint8_t yaw_motor_freq = 0;
uint8_t pitch_motor_freq = 0;
uint8_t roll_motor_freq = 0;

bool init = true;

for (;;)
{
    //Read and store to stack, the positional coordinates of the IMU stored in the
    //shared variables.

    imu_yaw_temp = imu_yaw.get();
    imu_pitch_temp = imu_pitch.get();
    imu_roll_temp = imu_roll.get();

    if(init){
        hmd_yaw_temp = 0;
        hmd_pitch_temp = 0;
        hmd_roll_temp = 0;

        if(abs(hmd_yaw_temp - imu_yaw_temp) < 400 && abs(hmd_pitch_temp -
            imu_pitch_temp) < 100 && abs(hmd_roll_temp - imu_roll_temp) <
            200){
            init = false;
        }
    }

    //Read and store to stack, positional coordinates of the HMD stored in the shared
variables.

    else{

        hmd_yaw_temp = -1 * hmd_yaw.get(); //Note: The HMD yaw axis is inverted,
            //so this must be multiplied by -1.
        hmd_pitch_temp = hmd_pitch.get();
        hmd_roll_temp = -1 * hmd_roll.get(); //Note: The HMD roll axis is
            //inverted, so this must be multiplied by -1.
    }
}

```

```

//Calculate the positional error.
yaw_error = hmd_yaw_temp - imu_yaw_temp;
pitch_error = hmd_pitch_temp - imu_pitch_temp;
roll_error = hmd_roll_temp - imu_roll_temp;

//Stop limits for the HMD. This prevents the motors from moving past their usable
limits (and hitting the platform).

if(hmd_yaw_temp < -10000){
    hmd_yaw_temp = -9000;
}
else if(hmd_yaw_temp > 10000){
    hmd_yaw_temp = 9000;
}

if(hmd_roll_temp < -7000){
    hmd_roll_temp = -6000;
}
else if(hmd_roll_temp > 7000){
    hmd_roll_temp = 6000;
}

if(hmd_pitch_temp < -7000){
    hmd_pitch_temp = -6000;
}
else if(hmd_pitch_temp > 7000){
    hmd_pitch_temp = 6000;
}

//Call PID method for each motor to have the gains stored into the shared
variables.

yaw_motor -> PID_yaw(yaw_error, imu_yaw_temp);
pitch_motor -> PID_pitch(pitch_error, imu_pitch_temp);
roll_motor -> PID_roll(roll_error, imu_roll_temp);

//The motors are frequency controlled, where this task can only modify the
period based on the counter. The counter
//variable is shortened or lengthened depending on the gain.

yaw_motor_freq = round(60/inv_gain_yaw.get());
pitch_motor_freq = round(60/inv_gain_pitch.get());
roll_motor_freq = round(60/inv_gain_roll.get());

//Limits the highest speed.

if(yaw_motor_freq < 1){
    yaw_motor_freq = 0;
}
if(pitch_motor_freq < 1){
    pitch_motor_freq = 0;
}
if(roll_motor_freq < 1){
    roll_motor_freq = 0;
}

//Control for Yaw motor movement.

if(abs(yaw_error) < 400){
    yaw_motor -> stop();
    counter_yaw = 0;
}
else if(counter_yaw > yaw_motor_freq){
    if(hmd_yaw_temp < imu_yaw_temp){
        yaw_motor -> move(1);
    }
}

```

```

        else if(hmd_yaw_temp > imu_yaw_temp){
            yaw_motor -> move(-1);
        }
        counter_yaw = 0;
    }
    //Control for Pitch motor movement.

    if(abs(pitch_error) < 100){
        pitch_motor -> stop();
        counter_pitch = 0;
    }
    else if(counter_pitch > pitch_motor_freq){
        if(hmd_pitch_temp < imu_pitch_temp){
            pitch_motor -> move(1);
        }
        else if(hmd_pitch_temp > imu_pitch_temp){
            pitch_motor -> move(-1);
        }
        counter_pitch = 0;
    }

    //Control for Roll motor movement.

    if(abs(roll_error) < 200){
        roll_motor -> stop();
        counter_roll = 0;
    }
    else if(counter_roll > roll_motor_freq){
        if(hmd_roll_temp < imu_roll_temp){
            roll_motor -> move(1);
        }
        else if(hmd_roll_temp > imu_roll_temp){
            roll_motor -> move(-1);
        }
        counter_roll = 0;
    }

    /*p_serial << imu_yaw.get() << endl;

    counter_yaw++;
    counter_pitch++;
    counter_roll++;

    runs++;

    delay_from_to (previousTicks, configMS_TO_TICKS (0.1));
}
}

```

```

//*****
/** \file readhmd_task.h
 *   This file contains the header for a task class that reads the HMD coordinate
 *   values to decode them into numeric form.
 *
 * Revisions:
 *   \li 09-30-2012 JRR Original file was a one-file demonstration with two tasks
 *   \li 10-05-2012 JRR Split into multiple files, one for each task
 *   \li 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
 *   \li 10-27-2012 JRR Altered from data sending task into LED blinking class
 *   \li 11-04-2012 JRR Altered again into the multi-task monstrosity
 *   \li 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
 *   \li 05-12-2015 Darren Chan - Completely re-written to provide functionality to
 *   read and decode characters from a serial port.
 *
 * License:
 *   This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 *   Public License, version 2. It intended for educational use only, but its use
 *   is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

// This define prevents this .h file from being included multiple times in a .cpp file
#ifndef _READHMD_TASK_H_
#define _READHMD_TASK_H_

#include <stdlib.h> // Prototype declarations for I/O functions
#include <avr/io.h> // Header for special function registers
#include "FreeRTOS.h" // Primary header for FreeRTOS
#include "queue.h" // FreeRTOS inter-task communication queues
#include "frt_task.h" // ME405/507 base task class
#include "time_stamp.h" // Class to implement a microsecond timer
#include "frt_queue.h" // Header of wrapper for FreeRTOS queues
#include "frt_shared_data.h" // Header for thread-safe shared data
#include "rs232int.h" // ME405/507 library for serial comm.
#include "shares.h" // Global ('extern') queue declarations

//-----
/** \brief This task receives data from the serial port, which correspond to HMD coordinates.
 *   \details The absolute encoder driver is run using a driver in files \c absencoder_driver.h
 *   and
 *   \c absencoder_driver.cpp.
 *   */

class readhmd_task : public frt_task
{
private:
    // No private variables or methods for this class
protected:
    // No protected variables or methods for this class
public:
    readhmd_task (const char*, unsigned portBASE_TYPE, size_t, emstream*);

    // This method is called by the RTOS once to run the task loop for ever and ever.
    void run (void);

    // This method converts char-type digits to numeric digits.
    int atoi(char digit);
};

#endif

```

```

//*****
/** \file readhmd_task.cpp
 *
 * License:
 * This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 * Public License, version 2. It intended for educational use only, but its use
 * is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****
#include "frt_text_queue.h" // Header for text queue class
#include "readhmd_task.h"
#include <util/delay.h>

//-----
/**
 * (\c frt_task ); the parent's constructor.
 * @param a_name A character string which will be the name of this task
 * @param a_priority The priority at which this task will initially run (default: 0)
 * @param a_stack_size The size of this task's stack in bytes
 * (default: configMINIMAL_STACK_SIZE)
 * @param p_ser_dev Pointer to a serial device (port, radio, SD card, etc.) which can
 * be used by this task to communicate (default: NULL)
 */

readhmd_task::readhmd_task (const char* a_name,
                           unsigned portBASE_TYPE
a_priority,
                           size_t a_stack_size,
                           emstream* p_ser_dev
                           )
: frt_task (a_name, a_priority, a_stack_size, p_ser_dev)
{
    p_serial = p_ser_dev;
    runs = 0;
}

//-----
/** \brief This method reads from the serial port one character at a time, and detects
 * the start delimiter, 'Y' to begin the data-read process. When this method is called,
 * it waits for the delimiter to guarantee that data will be collected from the iteration.
 * The data stream is stored into a buffer, then converted from ASCII to numeric values
 * to be stored in the shared variable line to be used by the motor task.
 */

void readhmd_task::run (void)
{
    portTickType previousTicks = xTaskGetTickCount ();

    int16_t r_byte = 0;
    bool y_flag = false;
    bool p_flag = false;
    bool r_flag = false;
    bool y_neg = false;
    bool p_neg = false;
    bool r_neg = false;
    uint8_t y_count = 0;
    uint8_t p_count = 0;
    uint8_t r_count = 0;

```

```

bool start_seq = false;
bool end_seq = false;
int16_t yaw, pitch, roll;
char y_buf[5], p_buf[5], r_buf[5];
    y_buf[0] = 0; y_buf[1] = 0; y_buf[2] = 0; y_buf[3] = 0; y_buf[4] = 0;
    p_buf[0] = 0; p_buf[1] = 0; p_buf[2] = 0; p_buf[3] = 0; p_buf[4] = 0;
    r_buf[0] = 0; r_buf[1] = 0; r_buf[2] = 0; r_buf[3] = 0; r_buf[4] = 0;

for (;;)
{
    while(!end_seq){
        if (p_serial->check_for_char())
        {
            r_byte = p_serial->getchar(); // Read the character
            if(r_byte > 0){
                if (r_byte == 'Y' && !start_seq) {
                    y_flag = true;
                    p_flag = false;
                    r_flag = false;
                    start_seq = true;
                }
                else if (r_byte == 'P' && start_seq) {
                    p_flag = true;
                    y_flag = false;
                    r_flag = false;
                }
                else if (r_byte == 'R' && start_seq) {
                    r_flag = true;
                    y_flag = false;
                    p_flag = false;
                }
                else if (r_byte == 'E' && start_seq) {
                    end_seq = true;
                }

                if (y_flag && start_seq) {
                    if (r_byte == '-') {
                        y_neg = true;
                    }
                    else if (r_byte != 'Y' && r_byte != ' ') {
                        y_buf[y_count] = r_byte;
                        y_count++;
                    }
                }
                else if (p_flag && start_seq) {
                    if (r_byte == '-') {
                        p_neg = true;
                    }
                    else if (r_byte != 'P' && r_byte != ' ') {
                        p_buf[p_count] = r_byte;
                        p_count++;
                    }
                }
                else if (r_flag && start_seq) {
                    if (r_byte == '-') {
                        r_neg = true;
                    }
                    else if (r_byte != 'R' && r_byte != ' ') {
                        r_buf[r_count] = r_byte;
                        r_count++;
                    }
                }
            }
        }
    }
    if (end_seq)
    {

```

```

//YAW CONVERSION
if (y_count == 5) { //5 Digits
    yaw = 100*(100 * atoi(y_buf[0]) + 10 * atoi(y_buf[1]) +
        atoi(y_buf[2]) + 0.1 * atoi(y_buf[3]) + 0.01 * atoi(y_buf[4]));
}
else if (y_count == 4) { //4 Digits
    yaw = 100*(10 * atoi(y_buf[0]) + atoi(y_buf[1]) + 0.1 *
        atoi(y_buf[2]) + 0.01 * atoi(y_buf[3]));
}
else if (y_count == 3) { //3 Digits
    yaw = 100*(atoi(y_buf[0]) + 0.1 * atoi(y_buf[1]) + 0.01 *
        atoi(y_buf[2]));
}
else if (y_count == 2) { //2 Digits
    yaw = 100*(0.1 * atoi(y_buf[0]) + 0.01 * atoi(y_buf[1]));
}
if (y_neg) {
    yaw = -1 * yaw;
}

//PITCH CONVERSION

if (p_count == 5) { //5 Digits
    pitch = 100*(100 * atoi(p_buf[0]) + 10 * atoi(p_buf[1]) +
        atoi(p_buf[2]) + 0.1 * atoi(p_buf[3]) + 0.01 * atoi(p_buf[4]));
}
else if (p_count == 4) { //4 Digits
    pitch = 100*(10 * atoi(p_buf[0]) + atoi(p_buf[1]) + 0.1 *
        atoi(p_buf[2]) + 0.01 * atoi(p_buf[3]));
}
else if (p_count == 3) { //3 Digits
    pitch = 100*(atoi(p_buf[0]) + 0.1 * atoi(p_buf[1]) + 0.01 *
        atoi(p_buf[2]));
}
else if (p_count == 2) { //2 Digits
    pitch = 100*(0.1 * atoi(p_buf[0]) + 0.01 * atoi(p_buf[1]));
}
if (p_neg) {
    pitch = -1 * pitch;
}

//ROLL CONVERSION

if (r_count == 5) { //5 Digits
    roll = 100*(10 * atoi(r_buf[0]) + 1 * atoi(r_buf[1]) +
        atoi(r_buf[2]) + 0.01 * atoi(r_buf[3]) + 0.001 *
        atoi(r_buf[4]));
}
else if (r_count == 4) { //4 Digits
    roll = 100*(atoi(r_buf[0]) + atoi(r_buf[1]) + 0.01 *
        atoi(r_buf[2]) + 0.001 * atoi(r_buf[3]));
}
else if (r_count == 3) { //3 Digits
    roll = 100*(0.1 * atoi(r_buf[0]) + 0.01 * atoi(r_buf[1]) + 0.001
        * atoi(r_buf[2]));
}
else if (r_count == 2) { //2 Digits
    roll = 100*(0.01 * atoi(r_buf[0]) + 0.001 * atoi(r_buf[1]));
}
if (r_neg) {
    roll = -1 * roll;
}

//Place the converted sensor readings into the shared variable
//corresponding with the axis.

hmd_yaw.put(yaw);
hmd_pitch.put(pitch);
hmd_roll.put(roll);

```

```

        //Reset all values.

        y_flag = false;
        p_flag = false;
        r_flag = false;
        y_neg = false;
        p_neg = false;
        r_neg = false;
        y_count = 0;
        p_count = 0;
        r_count = 0;
        start_seq = false;
        end_seq = false;
        y_buf[0] = 0; y_buf[1] = 0; y_buf[2] = 0; y_buf[3] = 0; y_buf[4] = 0;
        p_buf[0] = 0; p_buf[1] = 0; p_buf[2] = 0; p_buf[3] = 0; p_buf[4] = 0;
        r_buf[0] = 0; r_buf[1] = 0; r_buf[2] = 0; r_buf[3] = 0; r_buf[4] = 0;
    }

    runs++;

    delay_from_to (previousTicks, configMS_TO_TICKS (4));
}

}

//-----
/** \brief This function converts numeric ASCII characters to numeric integer values.
 * @param digit, a char-type variable that stores an ASCII character.
 */

int readhmd_task::atoi(char digit){
    if (digit == '1') {
        return 1;
    }
    else if (digit == '2') {
        return 2;
    }
    else if (digit == '3') {
        return 3;
    }
    else if (digit == '4') {
        return 4;
    }
    else if (digit == '5') {
        return 5;
    }
    else if (digit == '6') {
        return 6;
    }
    else if (digit == '7') {
        return 7;
    }
    else if (digit == '8') {
        return 8;
    }
    else if (digit == '9') {
        return 9;
    }
    else{
        return 0;
    }
}

```



```

//*****
/** \file readimu_task.h
 *   This file contains the header for a task class that controls the motor PWM,
 *   direction and mode of operation. This task controls a motor driver.
 *
 * Revisions:
 *   \li 09-30-2012 JRR Original file was a one-file demonstration with two tasks
 *   \li 10-05-2012 JRR Split into multiple files, one for each task
 *   \li 10-25-2012 JRR Changed to a more fully C++ version with class task_sender
 *   \li 10-27-2012 JRR Altered from data sending task into LED blinking class
 *   \li 11-04-2012 JRR Altered again into the multi-task monstrosity
 *   \li 12-13-2012 JRR Yet again transmogrified; now it controls LED brightness
 *   \li 05-12-2015 Darren Chan - Completely re-written to provide functionality to
 *       read and decode characters from a serial port.
 *
 * License:
 *   This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
 *   Public License, version 2. It intended for educational use only, but its use
 *   is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
 * TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
//*****

// This define prevents this .h file from being included multiple times in a .cpp file
#ifndef _READIMU_TASK_H_
#define _READIMU_TASK_H_

#include <stdlib.h> // Prototype declarations for I/O functions
#include <avr/io.h> // Header for special function registers

#include "FreeRTOS.h" // Primary header for FreeRTOS
#include "queue.h" // FreeRTOS inter-task communication queues

#include "frt_task.h" // ME405/507 base task class
#include "time_stamp.h" // Class to implement a microsecond timer
#include "frt_queue.h" // Header of wrapper for FreeRTOS queues
#include "frt_shared_data.h" // Header for thread-safe shared data

#include "rs232int.h" // ME405/507 library for serial comm.

#include "shares.h" // Global ('extern') queue declarations

//-----
/** \brief This task receives data from the serial port, which correspond to IMU coordinates.
 */

class readimu_task : public frt_task
{
private:
    // No private variables or methods for this class
protected:
    // No protected variables or methods for this class
public:
    readimu_task (const char*, unsigned portBASE_TYPE, size_t, emstream*);

    // This method is called by the RTOS once to run the task loop for ever and ever.
    void run (void);
    // This method converts char-type digits to numeric digits.
    int atoi(char digit);

```

```

};

#endif
/*****
** \file readimu_task.cpp
* This file contains the header for a task class that reads the IMU coordinate
* values to decode them into numeric form.
*
* License:
* This file is copyright 2012 by JR Ridgely and released under the Lesser GNU
* Public License, version 2. It intended for educational use only, but its use
* is not limited thereto. */
/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUEN-
* TIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */
*****/
#include "frt_text_queue.h" // Header for text queue class
#include "readimu_task.h"
#include "absencoder_driver.h"
#include <util/delay.h>

//-----
/**
* (\c frt_task ); the parent's constructor.
* @param a_name A character string which will be the name of this task
* @param a_priority The priority at which this task will initially run (default: 0)
* @param a_stack_size The size of this task's stack in bytes
* (default: configMINIMAL_STACK_SIZE)
* @param p_ser_dev Pointer to a serial device (port, radio, SD card, etc.) which can
* be used by this task to communicate (default: NULL)
*/

readimu_task::readimu_task (const char* a_name,
                                unsigned portBASE_TYPE
a_priority,
                                size_t a_stack_size,
                                emstream* p_ser_dev
                                )
: frt_task (a_name, a_priority, a_stack_size, p_ser_dev)
{
    p_serial = p_ser_dev;

    runs = 0;
}

//-----
/** \brief This method reads from the serial port one character at a time, and detects
* the start delimiter, 'Y' to begin the data-read process. When this method is called,
* it waits for the delimiter to guarantee that data will be collected from the iteration.
* The data stream is stored into a buffer, then converted from ASCII to numeric values
* to be stored in the shared variable line to be used by the motor task.
* Note that although the magnetometer readings are sent by the IMU and read by the
* microcontroller, they are not used; the magnetometer provides false readings when placed
* near motors. Instead, the absolute encoder driver is instantiated to determine Yaw
readings.
*/
void readimu_task::run (void)
{
    portTickType previousTicks = xTaskGetTickCount ();

    //Initialize the variables for the read sequence.

```

```

char r_byte = 0;
bool y_flag = false;
bool p_flag = false;
bool r_flag = false;
bool y_neg = false;
bool p_neg = false;
bool r_neg = false;
uint8_t y_count = 0;
uint8_t p_count = 0;
uint8_t r_count = 0;
bool start_seq = false;
bool end_seq = false;
int16_t yaw, pitch, roll;
char y_buf[5], p_buf[5], r_buf[5];
    y_buf[0] = 0; y_buf[1] = 0; y_buf[2] = 0; y_buf[3] = 0; y_buf[4] = 0;
    p_buf[0] = 0; p_buf[1] = 0; p_buf[2] = 0; p_buf[3] = 0; p_buf[4] = 0;
    r_buf[0] = 0; r_buf[1] = 0; r_buf[2] = 0; r_buf[3] = 0; r_buf[4] = 0;

//Instantiate the absolute encoder driver to collect Yaw data.

absencoder_driver* absencoder = new absencoder_driver (p_serial, &DDRF, &PINF, PF0,
PF1, PF2, PF3, PF4, PF5, PF6, PF7);

for (;;)
{
    while(!end_seq){
        if (p_serial->check_for_char())
        {
            r_byte = p_serial->getchar(); // Read the character
            if (r_byte == 'Y' && !start_seq) {
                y_flag = true;
                p_flag = false;
                r_flag = false;
                start_seq = true;
            }
            else if (r_byte == 'P' && start_seq) {
                p_flag = true;
                y_flag = false;
                r_flag = false;
            }
            else if (r_byte == 'R' && start_seq) {
                r_flag = true;
                y_flag = false;
                p_flag = false;
            }
            else if (r_byte == 'E' && start_seq) {
                end_seq = true;
            }

            if (y_flag && start_seq) {
                if (r_byte == '-') {
                    y_neg = true;
                }
                else if (r_byte != 'Y' && r_byte != '.') {
                    y_buf[y_count] = r_byte;
                    y_count++;
                }
            }
            else if (p_flag && start_seq) {
                if (r_byte == '-') {
                    p_neg = true;
                }
                else if (r_byte != 'P' && r_byte != '.') {
                    p_buf[p_count] = r_byte;
                    p_count++;
                }
            }
            else if (r_flag && start_seq) {
                if (r_byte == '-') {

```

```

        r_neg = true;
    }
    else if (r_byte != 'R' && r_byte != '.') {
        r_buf[r_count] = r_byte;
        r_count++;
    }
}
}
if (end_seq)
{
    //YAW CONVERSION
    if (y_count == 5) { //5 Digits
        yaw = 100*(100 * atoi(y_buf[0]) + 10 * atoi(y_buf[1]) +
            atoi(y_buf[2]) + 0.1 * atoi(y_buf[3]) + 0.01 * atoi(y_buf[4]));
    }
    else if (y_count == 4) { //4 Digits
        yaw = 100*(10 * atoi(y_buf[0]) + atoi(y_buf[1]) + 0.1 *
            atoi(y_buf[2]) + 0.01 * atoi(y_buf[3]));
    }
    else if (y_count == 3) { //3 Digits
        yaw = 100*(atoi(y_buf[0]) + 0.1 * atoi(y_buf[1]) + 0.01 *
            atoi(y_buf[2]));
    }
    if (y_neg) {
        yaw = -1 * yaw;
    }

    //PITCH CONVERSION

    if (p_count == 5) { //5 Digits
        pitch = 100*(100 * atoi(p_buf[0]) + 10 * atoi(p_buf[1]) +
            atoi(p_buf[2]) + 0.1 * atoi(p_buf[3]) + 0.01 * atoi(p_buf[4]));
    }
    else if (p_count == 4) { //4 Digits
        pitch = 100*(10 * atoi(p_buf[0]) + atoi(p_buf[1]) + 0.1 *
            atoi(p_buf[2]) + 0.01 * atoi(p_buf[3]));
    }
    else if (p_count == 3) { //3 Digits
        pitch = 100*(atoi(p_buf[0]) + 0.1 * atoi(p_buf[1]) + 0.01 *
            atoi(p_buf[2]));
    }
    if (p_neg) {
        pitch = -1 * pitch;
    }

    //ROLL CONVERSION

    if (r_count == 5) { //5 Digits
        roll = 100*(10 * atoi(r_buf[0]) + 1 * atoi(r_buf[1]) +
            atoi(r_buf[2]) + 0.01 * atoi(r_buf[3]) + 0.001 *
            atoi(r_buf[4]));
    }
    else if (r_count == 4) { //4 Digits
        roll = 100*(1 * atoi(r_buf[0]) + atoi(r_buf[1]) + 0.01 *
            atoi(r_buf[2]) + 0.001 * atoi(r_buf[3]));
    }
    else if (r_count == 3) { //3 Digits
        roll = 100*(0.1 * atoi(r_buf[0]) + 0.01 * atoi(r_buf[1]) + 0.001
            * atoi(r_buf[2]));
    }
    if (r_neg) {
        roll = -1 * roll;
    }

    absencoder -> read_value();
    imu_pitch.put(pitch);
    imu_roll.put(roll);
}

```

```

        y_flag = false;
        p_flag = false;
        r_flag = false;
        y_neg = false;
        p_neg = false;
        r_neg = false;
        y_count = 0;
        p_count = 0;
        r_count = 0;
        start_seq = false;
        end_seq = false;
        y_buf[0] = 0; y_buf[1] = 0; y_buf[2] = 0; y_buf[3] = 0; y_buf[4] = 0;
        p_buf[0] = 0; p_buf[1] = 0; p_buf[2] = 0; p_buf[3] = 0; p_buf[4] = 0;
        r_buf[0] = 0; r_buf[1] = 0; r_buf[2] = 0; r_buf[3] = 0; r_buf[4] = 0;
    }

    runs++;

    delay_from_to (previousTicks, configMS_TO_TICKS (4));
}

//-----
/** \brief This function converts numeric ASCII characters to numeric integer values.
 * @param digit, a char-type variable that stores an ASCII character.
 */

int readimu_task::atoi(char digit){
    if (digit == '1') {
        return 1;
    }
    else if (digit == '2') {
        return 2;
    }
    else if (digit == '3') {
        return 3;
    }
    else if (digit == '4') {
        return 4;
    }
    else if (digit == '5') {
        return 5;
    }
    else if (digit == '6') {
        return 6;
    }
    else if (digit == '7') {
        return 7;
    }
    else if (digit == '8') {
        return 8;
    }
    else if (digit == '9') {
        return 9;
    }
    else{
        return 0;
    }
}

```