2014

Design and Control of a Two-Wheeled Robotic Walker

Airton R. da Silva Jr.
University of Massachusetts Amherst

DESIGN AND CONTROL OF A TWO-WHEELED ROBOTIC WALKER

A Thesis Presented

by

AIRTON R. DA SILVA JR.

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

September 2014

Mechanical and Industrial Engineering

DESIGN AND CONTROL OF A TWO-WHEELED ROBOTIC WALKER

A Thesis Presented

by

AIRTON R. DA SILVA JR.

Approved as to style and content by:

_____
Frank C. Sup IV, Chair

_____
Roderic A. Grupen, Member

_____
Yossi Chait, Member

_____
Donald Fisher, Department Head
Mechanical and Industrial Engineering

ACKNOWLEDGMENTS

I would like to express my greatest gratitude to my advisor, Prof. Frank Sup, for all his guidance, encouragement, and support throughout my graduate studies. I am deeply thankful for the opportunity to work in his laboratory and for his commitment to education and research work. I would also like to thank the members of my thesis committee, Prof. Roderic Grupen and Prof. Yossi Chait, for all their invaluable guidance and insight on the research project. Special thanks to Prof. Grupen for his contribution to prototype construction and the opportunity to cooperate with lab members from the Laboratory for Perceptual Robotics (LPR).

I would also like to thank all lab members (Ryder, Cummings, Kadrolkar, LaPrè, Zhang, and Nie) from the Mechatronics and Robotics Research Laboratory (MRRL) for their assistance, insight, and support during various stages of my thesis project. I would like to specifically thank Matthew Ryder for all his witty and enthralling tutoring on various electrical engineering topics pertinent to my thesis project.

I would like to give a special thanks to undergraduate student, Natalie Zucker, for her outstanding design and construction work on the instrumented handlebar used in the prototype. Another very special thanks to Dr. Cynthia Jacelon from the Nursing Department for her insight on geriatric care and rehabilitation. I would also like thank the kind staff and nurses of the Jewish Geriatric Services in Longmeadow, MA, for granting me a tour of their facilities and allowing me to collect critical observational data pertinent to my research. Another thanks to Brittney Muir from Purdue for sharing her knowledge on the gait and balance characteristics of elderly people.

I would also like to thank Dirk Ruiken from the LPR for his guidance and support during firmware development. I would also like to thank Rick Winn and Miles Eastman from the MIE department's machine shop for their assistance and guidance during prototype construction. Finally, I would like to offer my greatest gratitude to my family and friends, without whom it would not have been possible for me to complete my master's thesis.

ABSTRACT

DESIGN AND CONTROL OF A TWO-WHEELED ROBOTIC WALKER

SEPTEMBER 2014

AIRTON R. DA SILVA JR.

M.S.M.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Frank C. Sup IV

This thesis presents the design, construction, and control of a two-wheeled inverted pendulum (TWIP) robotic walker prototype for assisting mobility-impaired users with balance and fall prevention. A conceptual model of the robotic walker is developed and used to illustrate the purpose of this study. A linearized mathematical model of the two-wheeled system is derived using Newtonian mechanics. A control strategy consisting of a decoupled LQR controller and three state variable controllers is developed to stabilize the platform and regulate its behavior with robust disturbance rejection performance. Simulation results reveal that the LQR controller is capable of stabilizing the platform and rejecting external disturbances while the state variable controllers simultaneously regulate the system's position with smooth and minimum jerk control.

A prototype for the two-wheeled system is fabricated and assembled followed by the implementation and tuning of the control algorithms responsible for stabilizing the prototype and regulating its position with optimal performance. Several experiments are conducted, confirming the ability of the decoupled LQR controller to robustly balance the platform while the state variable controllers regulate the platform's position with smooth and minimum jerk control.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Retirement has reached a critical point in the United States with approximately 10,000 baby boomers reaching the retirement age of 65 every day for the next 16 years [1]. In time, many of these older adults will start to experience age-associated functional and cognitive impairments that may lead to dependence on mobility aids and personal care assistants in order accomplish basic and instrumental activities of daily living (ADLs). However, with the rising cost of health care and shortages of trained professionals, many industries are turning to intelligent robots in pursuit of an effective solution to some of these challenges.

1.1. Research Motivation

The motivation of this research is based on the need to alleviate the demand for trained professionals by providing older adults and mobility-impaired users with prolonged access to the essential tasks of daily living. As of 2010, the cost of the three key federal entitlement programs for older Americans is roughly 9.4 percent of the gross domestic product (GDP) but this expenditure is projected to increase dramatically to a whopping 28 percent of the GDP by 2050 [2]. These expenditures along with the rapid increase in the aging population are placing severe financial stress on entitlement programs and threatening the nation's economic stability. In an effort to introduce a viable solution to the shortage of qualified personal care assistants and economic challenges associated with the dramatic increase in the older adult population, an intelligent robotic walker with multiple functionalities is proposed.

This research has also been motivated by Dr. Roderic Grupen's work in dexterous mobile manipulators and on distributed service robots in residential older adult care. Dr. Grupen is a professor in the Computer Science Department at the University of Massachusetts Amherst, where he also serves as director of the Laboratory for Perceptual Robotics (LPR).

1.2. Overall Research Purpose and Vision

The purpose of this research is to develop an intelligent robotic vehicle capable of assisting older adults as well as mobility-impaired users with balance and stability. To accomplish this goal, the intermediate conceptual model depicted in Fig. 1-1 was devised and formulated. The proposed concept model combines the superior maneuverability of a two-wheeled mobile platform with the structural robustness of a four-wheeled system. Accordingly, two modes of operation are possible: two-wheeled and four-wheeled. The ability of the vehicle to transition between these two modes is accomplished by the addition of two symmetrically operated arms.

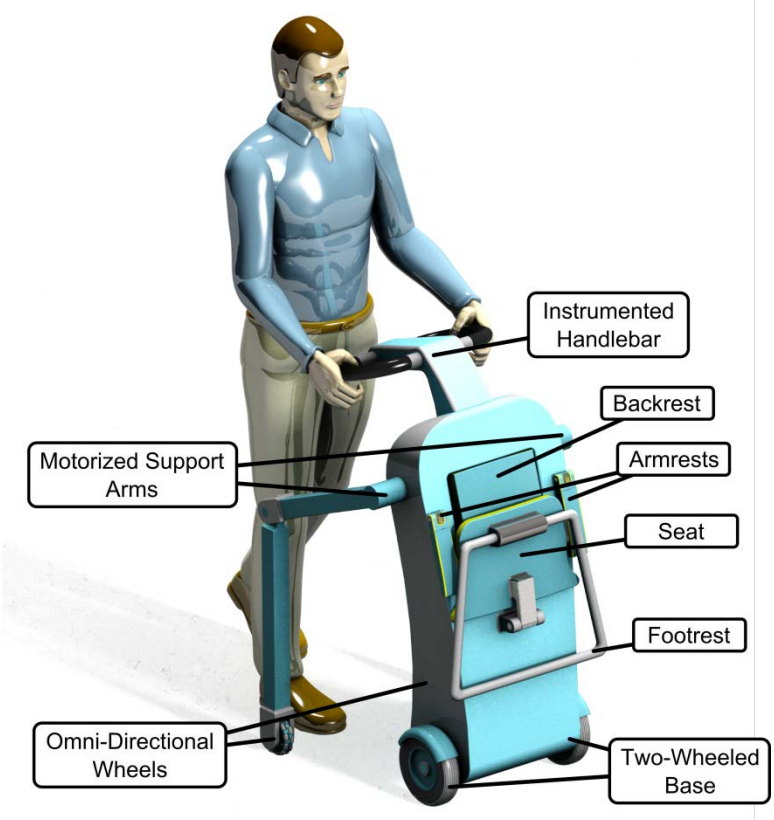

Figure 1-1. Intermediate conceptual model of the TWIP robotic walker.

The actively powered arms are an essential component to making the two-wheeled inverted pendulum (TWIP) robotic walker succeed as a superior substitute for a standard passive walker. With the integration of two arms, the robotic walker can be used to accommodate a wider range of users with variable assistance needs and help users transfer between sitting and standing positions. A few additional functionalities that become accessible for a TWIP system with two arms include wheelchair mode, advanced fall prevention capabilities, and a four-point base support for added stability while walking.

The vehicle drives in reaction to user input forces observed by force sensors placed at the handlebar and utilizes an active impedance scheme to regulate gait and

speed based on a set of safety parameters. The active impedance control scheme is shared in both modes of operation but the two-wheeled mode consists of three additional controllers: two decoupled state-space controllers and a pitch controller. One of the decoupled state-space controllers is dedicated to upright stabilization of the vehicle while the other controls yaw rotation. Meanwhile, the pitch controller adjusts the pitch angle of the robotic walker towards the user based on measured force applied at the handlebars. This leaning action creates a supporting moment against the user which can serve to increase the stability of the device and mitigate external disturbance forces.

Meanwhile, by developing a fall prevention control scheme and using sensors to monitor the user's gait, vital signs, and biomechanical performance, the risk of falls can be significantly reduced. However, the development of an effective fall prevention control scheme will require intelligent decision-making skills to ensure that the device does not react to false positives. The integration of sensors into the vehicle could also enable autonomy and obstacle avoidance intelligence. However, these functions are just general possibilities and not all are in the current focus of this research. The specific scope of this research will be discussed in the following section.

1.3. Scope of Research and Project Overview

As a first step towards the overall vision, the contribution of this study is the development, prototype construction, and evaluation of a prototype two-wheeled walker without robotic arms. Prior to concept design, an extensive review of background and prior work was performed on personal mobility aids and smart walkers as well as modeling and control approaches for two-wheeled inverted pendulums. Following

literature review, design constraints and system requirements were established and utilized to create a solid model of the TWIP robotic walker using Pro/ENGINEER.

A simplified free-body diagram (FBD) of the concept model was created and used to obtain the kinematic equations of the vehicle based on the principles of Newtonian mechanics. The kinematic equations were combined, linearized, and rewritten to obtain the coupled state-space representation. In order to eliminate the effects of loop interaction and impose the desired dynamics on the system, a decoupling control scheme was implemented. The resulting decoupled state-space representation was then used to design a robustly tuned linear quadratic regulator (LQR) controller for upright stabilization of the TWIP system in MATLAB/Simulink. Improved disturbance rejection was achieved through the integration of a pitch controller. To control the vehicle's linear displacement and heading angle, two additional state variable controllers were developed and simulated.

The prototype demonstrated in Fig. 1-2 was fabricated and all assembly components were identified, including hardware and electronics. The base assembly is an aluminum frame consisting of two wheels, arranged in a differential drive configuration, that are coupled to two direct drive brushless DC motors, each controlled by a Puck™ (B3880) motor-drive-electronics module from Barrett Technology, Inc. An instrumented aluminum handlebar design with two torque load cells was secured to the top of the base assembly.

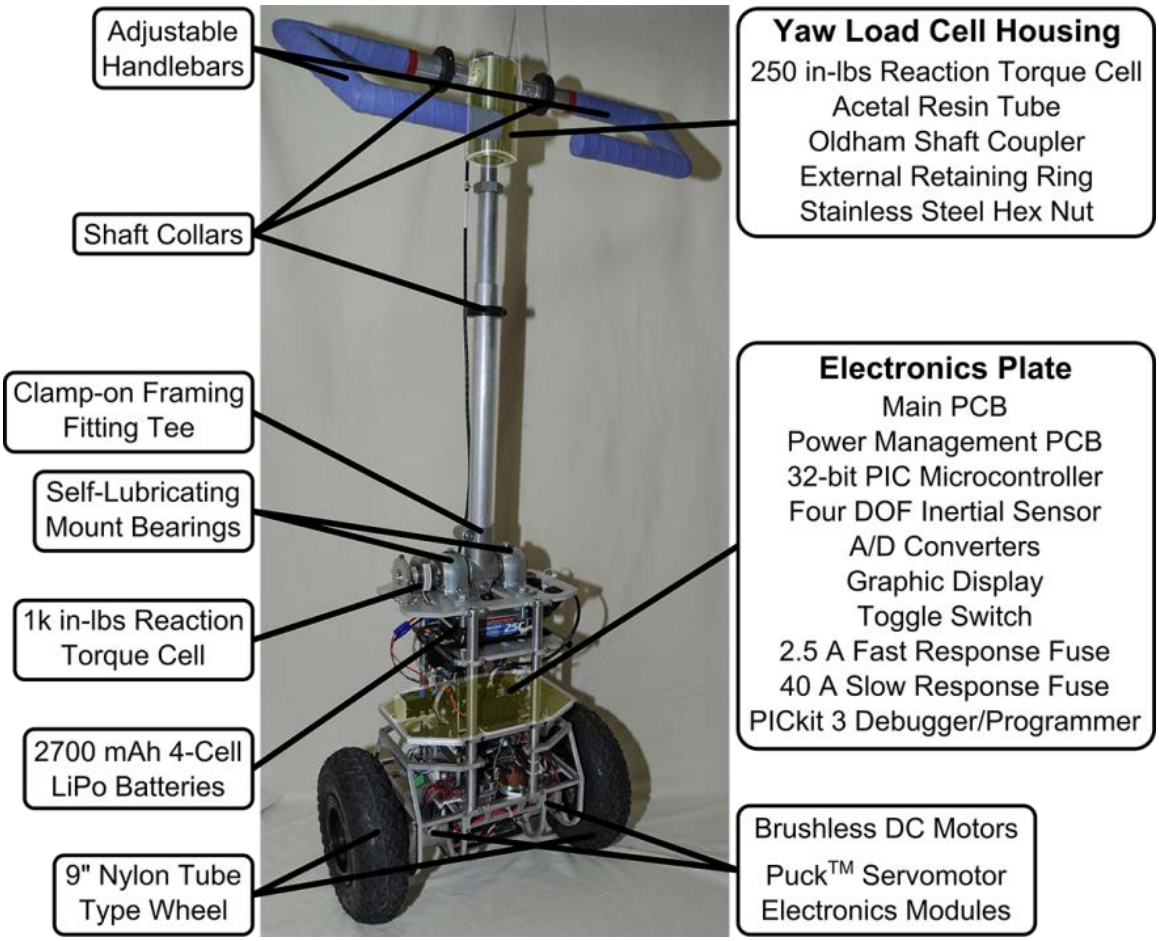

Figure 1-2. Completed prototype assembly of the two-wheeled system.

The Puck motor drivers communicate with a 32-bit peripheral interface controller (PIC) through a controller area network (CAN) bus. The vehicle's pitch rate and tri-axial acceleration data are measured and processed by a four degrees of freedom inertial measurement unit (IMU) which interfaces with the PIC using serial peripheral interface (SPI) protocol. Meanwhile, data collected from the load cells are processed through an external analog-to-digital converter (ADC) and transmitted via SPI to the PIC.

With the prototype assembled and all electronics integrated, the software developed to drive all on-board electronics and control the platform was implemented

and tuned. Several experiments were conducted in order to test and evaluate the performance of the decoupled LQR and state variable controllers responsible for stabilizing the platform and regulating its position with smooth and minimum jerk control.

1.4. Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 reviews background and prior work related to personal mobility aids, smart walkers, physical modeling techniques and control methodologies for TWIP systems. Chapter 3 describes the design process and requirements that had to be fulfilled by the developed conceptual model. Chapter 4 derives the equations of motion describing system dynamics, combines and rewrites these equations into state-space form, and conducts an open-loop analysis. Chapter 5 identifies, develops, and simulates the control strategy that will be utilized to dictate the desired behavior of the vehicle.

Chapter 6 discusses hardware selection and architecture as well as the design process of two custom printed circuit boards (PCB) used to power all on-board electronics and implement the software designed to control the two-wheeled system. Chapter 7 details the development of the firmware and software responsible for driving the hardware and controlling the two-wheeled system. Chapter 8 covers the development and construction process of a prototype for the TWIP robotic walker. Chapter 9 evaluates the overall performance of the platform by conducting and analyzing a series of experiments. Chapter 10 concludes this thesis with a summary of prototype performance and recommendations for future work.

CHAPTER 2

BACKGROUND AND PRIOR WORK

This chapter provides background and prior work information on four topics that are relevant to this research: personal mobility aids, smart walkers, physical modeling and control methodologies for TWIP systems.

2.1. Personal Mobility Aids and Smart Walkers

Personal mobility aids come in all forms from simple walking canes to fully motorized scooters. For immobile users, the solution is usually a wheelchair. However, if the user has residual mobility capacities, then the solution may be found within mobility devices that provide either wearable augmentation—orthoses and prostheses—or external assistance from canes, crutches, or walkers.

Among the external augmentative mobility devices, there have been various studies conducted on smart walkers that can promote cognitive and physical functioning through sensorial assistance [3]–[5], active physical support [6], [7], and fall prevention [8]. However, most of these studies were performed on either a three-wheeled or four-wheeled walker with very little research on two-wheeled walking support systems. Murata Manufacturing has demonstrated a research prototype called the KeePace™. The self-balancing inverted pendulum, as shown in Fig. 2-1, is intended to be used as a walker. The main challenges in designing a two-wheeled walker that can surpass conventional ones are device safety and load capacity. Our research proposes a potential solution to these challenges and dedicates itself to devising and developing such mobility

aid; but first, a proper mathematical description and control scheme for the two-wheeled system is required.

Figure 2-1. KeePace™ Murata Manufacturing.

2.2. Two-Wheeled Inverted Pendulum Physical Modeling

Over the past few decades, research on the TWIP system has gained extensive momentum and become increasingly popular with researchers around the world. As a result, a substantial amount of literature on the TWIP is readily available along with a variety of modeling approaches and control methodologies. Some of the approaches that researches have taken to derive the mathematical description of the TWIP model include Newtonian mechanics [9]–[18], Lagrangian formulation [19]–[25], and Kane's method [26]–[28]. For this research, the system of equations representing the dynamics of the two-wheeled system will be derived using Newtonian mechanics.

2.3. Control Methodology

The solution to stabilizing an inherently unstable system such as the TWIP can be very complex and diverse. As there is no single control scheme that will perform efficiently for every scenario, there have been distributed efforts in controlling the TWIP system using zero-moment point (ZMP) [9], pole placement [10], [28], linear quadratic regulator (LQR) [13], [14], [16], [19], [20], [26], proportional-integral-derivative (PID) [11], [18], [21], [29], [30], and fuzzy control [17], [22]. Although the aforementioned control methodologies were all applied to linearized systems, there has also been a wide range of research dedicated to controlling nonlinear TWIP systems [12], [15], [31], [32]. In Lin *et al.*'s (2009) work, the authors developed an effective adaptive control scheme based on the widely adopted nonlinear control methodology, sliding mode control (SMC) [12]. Another control scheme that can efficiently manage nonlinear systems and perform extremely well under disturbance forces is H_∞ (H-infinity) optimal control [31], [33].

Meanwhile, Lin *et al*. (2011) took a further step into the field of robust control by implementing an adaptive robust control scheme to balance a two-wheeled human transportation vehicle (HTV) with system uncertainties and parameter variations [34]. In their background research, the authors discovered that the performances of a partial feedback linearization scheme and an adaptive linearized control method may be severely impaired by parameter uncertainty and input signal variations, respectively. Motivated to circumvent these limitations, the authors decided to develop two adaptive robust regulators based on the Lyapunov stability theory. The proposed control scheme consists of two decoupled adaptive robust proportional-derivative (PD) controllers that are

independently responsible for controlling the upright stabilization and yaw motion of the vehicle. Simulation and experimental results demonstrated that the proposed regulators were capable of achieving satisfactory riding performance for riders of varying weight.

The implementation of a model reference adaptive controller (MRAC) can be found in Yang *et al*. (2012)'s published study [35]. In their work, an optimized adaptive control scheme leveraging on LQR and neural network (NN) was formulated to control a WIP system in the presence of plant uncertainties and time-varying external disturbances. First, LQR optimization was employed to derive a reference model with minimized motion tracking error and transient acceleration for best driving comfort. A MRAC algorithm was then implemented to make the controller dynamics match the reference model dynamics while a NN-based adaptive generator of implicit control trajectory (AGICT) was used to approximate the tilt angle responsible for implicitly manipulating the forward velocity. Despite the presence of unknown system parameters and external disturbance, the proposed adaptive control scheme was able to guarantee exact tracking of the tilt and yaw of the vehicle while the NN-based trajectory planner maintained desired forward velocity trajectories.

Most controllers are designed to operate within a set of operating conditions. Consequently, designing a single controller that can optimally control a system with changing dynamics is often difficult or even impossible [36]. The control strategy developed for the robotic walker aims to incorporate multiple controllers in order to optimize task-specific functions operating under varying operating conditions. Not every controller will be operating at the same time, as a result, it is necessary to investigate the

effects of transitioning between controllers. The importance of implementing a smooth state transition controller is emphasized Kalra *et al*.'s work [14].

In light of the ability of LQR controller to stabilize a two-wheeled system with optimal overshoot and settling performance, this research will be implementing LQR control. To eliminate the effects of loop interaction and impose the desired dynamics on the system, the decoupling scheme discussed in Grasser *et al.*'s research [10] will also be employed. Furthermore, to regulate the two-wheeled system's linear displacement, pitch angle, and heading angle, three state variable controllers will be formulated and implemented.

CHAPTER 3

CONCEPT DESIGN

Due to the multidisciplinary nature of this research, a wide variety of insights were collected from professors and graduate students working in the Computer Science, Kinesiology, and Nursing Departments in addition to the Mechanical Engineering Department. The names and contributions of these individuals have been acknowledged at the beginning of this thesis. In combination with literature reviews and contributions from each department, a novel robotic platform was designed. This chapter describes the mechanical design process and the steps that were taken to develop a conceptual model for the TWIP walker.

3.1. Mechanical Design Process

Prior to drafting concept models for a new type of intelligent robotic walker, it was necessary to define the essential characteristics concerning the structure and functionalities of such device. To assist with the identification of these characteristics, the group of individuals that would be using the robotic walker were identified and their physical and cognitive needs were explored. The next step in the mechanical design process was to conduct an extensive background research on existing robotic walkers that were on the market or under development. Once enough information about the users and existing robotic walkers had been acquired, a list of design features covering the requirements for a robotic walker concept model was identified and described.

3.1.1. Customer Identification and Requirements

The group of individuals that will be using the robotic walker are older adults and mobility-impaired users. The next step is to understand the physical and cognitive needs of these group of individuals and how they can be addressed by specific design features. Considering how there already exists a rich quantity of mobility support systems that are either available on the market or currently being researched, all essential customer requirements can be identified by reviewing existing surveys and literature.

As part of identifying customer requirements, one important step is to consider some of the problems that older adults have with existing walkers. In 1995, Mann *et al.* conducted an analysis on the problems that older adults with disabilities encountered while using a walker [37]. The sample used by these researchers consisted of 333 older adults diagnosed with some sort of difficulty with ambulation. Of the 333 subjects in the analysis, 69 used a walker. It was reported that 42 out of the 69 walker users had some complaint or dissatisfaction regarding the walker they owned. Some of the reasons for dissatisfaction with walkers reported by these older adults are summarized in Table 3-1.

Table 3-1. Summary of reasons for dissatisfaction with walkers among a sample of older adults as reported in Mann et al.'s research work [37].

Difficulties handling the device
Unable to use the device outdoors
Difficulties adjusting to the device
Too clumsy and bulky
Too heavy to lift
Poor structural stability
Lack of knowledge about how to use device
Pain caused by the lack of an ergonomic grip
Poor fit with environment or person's need
Denial of need
Lack of emotional security
Feelings of embarrassment

Using the list of problems that older adults encountered while using a walker, several customer requirements for the robotic walker design can be identified. According to the reasons for dissatisfaction among walker users presented in Table 3-1, a walker should be easy to operate, easy to learn, agile and slim, lightweight, structurally robust, comfortable to use, adaptable to the environment and person's needs, secure, and aesthetically pleasing. Although aesthetics has been regarded by some researchers as an unimportant feature [38], many researchers have emphasized its importance [39], [40]. Based on research findings, some of the basic features that older adults expect from a walker are listed in Table 3-2.

Table 3-2. List of features that older adults expect from a walker.

Easy to operate
Easy to learn
Agile and slim
Lightweight
Structurally robust and stable
Comfortable to use
Adaptable to the environment and person's needs
Secure
Aesthetically pleasing

3.1.2. Existing Robotic Walkers

Another critical step in the mechanical design process is the investigation and analysis of existing robotic walker technologies. Researching existing products provides the opportunity to learn about the robotic walker technologies that already exists as well as the opportunity to improve on what already exists. A compilation of some existing robotic walker prototypes and products along with the key features of each device are listed in Table 3-3.

Table 3-3. Robotic walker prototypes and products.

Robotic Walker	Key Features
i-Walker [5]	Monitorization, navigation support, cognitive support, shared autonomy, handlebars and gait sensors, active walking support
Walking Helper II [41]	Three wheeled omnidirectional mobile base, sit-to-stand support system, force/torque sensors, front laser range system
SIMBIOSIS Walker [42]	Predictive human-machine cooperation, human-machine interface, braking control, leg sensors, instrumented forearm support
JARoW [43]	Three omnidirectional wheels, lower limb location tracking, natural user interface, obstacle avoidance, feedback motion control
MARC Smart Walker [44]	Sonar, infrared sensors, wheel encoders, obstacle avoidance, warning system, safety braking and steering control, path following
RT Walker [45]	Obstacle avoidance, gravity compensation, path following, user intent estimation, variable motion characteristics, fall prevention
Guido [46]	Map-based navigation, spring-loaded handlebars with sensors, obstacle avoidance, path-planning algorithm, autonomous
PAMM Smart Walker [47]	Health monitoring, medicine and other scheduling, omnidirectional mobility, user intent estimation, guidance and obstacle avoidance

3.1.3. Design Features

In this step, the goal is to utilize the knowledge of customer requirements and existing robotic walker technologies to generate a list design features that will serve as a benchmark for developing the concept model of a novel robotic walker. One of the most important features desired for the concept model is fall prevention. Preventing falls can be accomplished in several different ways but one of the most effective methods is to reduce the risk factors that contribute to falls [48]. Risk factors for falls can be classified into two major categories: intrinsic factors and extrinsic factors. Intrinsic risk factors pertains to age-related physiologic changes, diseases, and psychological problems while extrinsic risk factors pertains to environmental hazards such poor home design or ground obstacles. A feature that could reduce the risk factors that contribute to falls is an active impedance scheme that regulates that speed of the user based on the user's vital signs and biomechanical performance.

Another desired feature for the robotic walker is the intelligence to visually recognize environmental obstacles that could threaten the user's stability and to eliminate the fall risk by warning the user or helping the user circumnavigate around the obstacle. The ability of the device to assist the user transfer between sitting and standing positions is also another desirable feature. Furthermore, to address fatigue mitigation, another design feature would be integration of a padded seat, backrest, and armrests. Lastly, physical therapy support is another feature that could be enabled by integrating sensors into the device to monitor the user's vital signs and biomechanical performance. The list of desired features that should be included in the design of a novel robotic walker, which also agrees with the findings in Korba et al.'s research, is summarized in Table 3-4 [49].

Table 3-4. Desired features for the ideal intelligent walker as defined in Korba *et al*. [49].

Desired Feature	**Description**
Intelligence	Possesses the ability to autonomously navigate through an environment with obstacle avoidance intelligence.
Cooperative Behavior	Contains a controller that automatically adjusts the walker's speed to accommodate the user based on monitored user gait data.
Lifting Mechanism	Capable of helping users transfer between sitting and standing positions while providing just the right amount of support the user needs.
Physical Therapy Support	Able to assist in walking therapy sessions by monitoring and recording user's vital signs and biomechanical performance.
Active Impedance Scheme	Executes carefully incremented exercises by setting limits on maximum heart rate, maximum speed, and distance traveled based on data recorded from user's vital signs and biomechanical performance.
Voice User Interface	Allows the user to command and communicate with the device through speech recognition.
Fatigue Mitigation	Provides a seat for users who become fatigue.
Fall Prevention Mechanism	The integration of a support mechanism to prevent the user from falling is considered to be the most important feature.

3.2. Solid Modeling

Using the design criteria developed in the previous section, several designs were drafted and modeled until an intermediate concept model was selected. An intermediate conceptual model of the platform can be seen in Fig. 3-1. The proposed concept model combines the superior maneuverability of a two-wheeled mobile platform with the structural robustness of a four-wheeled system. With the proposed mechanical configuration, two modes of operation can be achieved: two-wheeled and four-wheeled. The transition between the two-wheeled and four-wheeled modes is accomplished through the integration of two motorized mechanical arms.

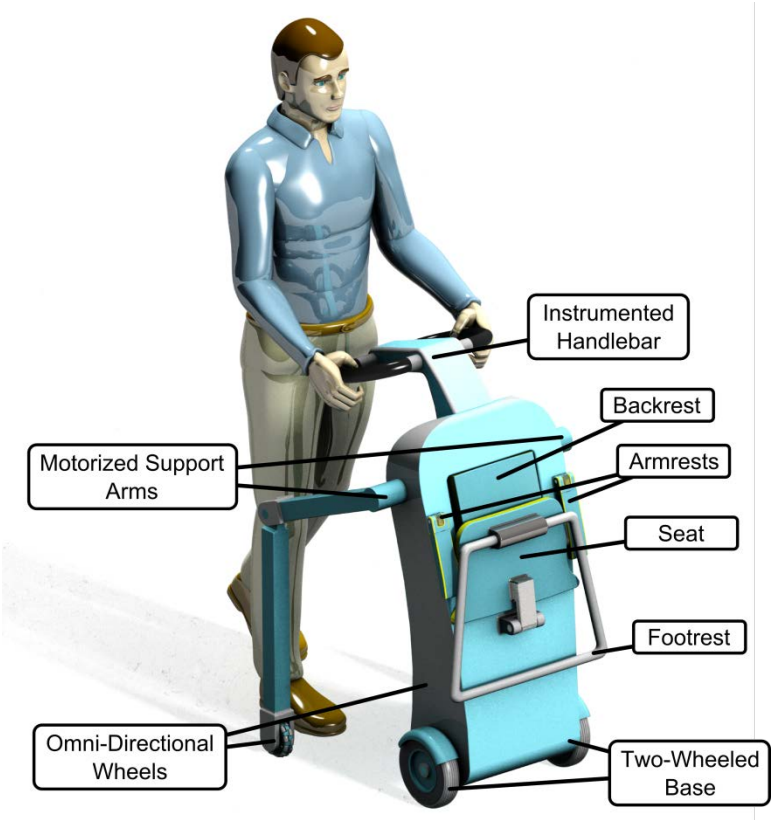

Figure 3-1. Intermediate conceptual model of the TWIP robotic walker.

The integration of two actively powered support arms enables the robotic walker to accommodate a wider range of users with variable assistance needs as well as to assist users transfer between sitting and standing positions. A few more features enabled by the implementation of two robotic arms include wheelchair mode, advanced fall prevention capabilities, and a four-point base support for added stability during walking.

The task of balancing the platform in its two-wheeled mode is accomplished using a decoupled LQR control scheme. Improved disturbance rejection in the two-wheeled mode is achieved using a pitch controller, which adjusts the pitch angle of the robotic walker towards the user based on the measured force applied at the handlebar. The tilting action imposed by this controller causes the platform to shift its center of mass towards the user, thereby creating a supporting moment against the user that can serve to increase the stability of the device and mitigate external disturbance forces. To drive the platform, two torque load cells integrated into the handlebar are utilized to measure the user's input forces and determine whether the user desires to travel along the platform's sagittal plane or turn about its yaw axis. The torque measurements collected from these load cells also allow the implementation of two state variable controllers capable of actively regulating the user's gait and speed by controlling the platform's linear and yaw velocities. The concept model presented in Fig. 3-1 and its design features is currently the long term vision. In this thesis, a TWIP with no robotic arms was developed.

CHAPTER 4

SYSTEM MODELING

Prior to developing a controller for the system, a mathematical description must be created to facilitate the process. Since the two robotic arms are not included in this initial prototype, arm dynamics will be ignored. Several assumptions that do not significantly compromise the accuracy of simulated results were made. First, it is assumed that wheels are always in contact with the ground and do not slip. Additionally, a linearized version of the nonlinear system can be utilized as long as the simulated pitch angle of the pendulum remains within ±20 degrees of the operating point [50]. Lastly, since the system will be linearized by small angle approximation of θ_P, the moment of inertia of the chassis about the z-axis J_{P_z}, which varies as the pendulum swings back and forth, is assumed to be constant.

The plant model developed in this chapter describes the physical behavior of the robotic walker over a specific operating range. If the system is forced out of this operating range then the accuracy of the plant model is compromised, making the system more prone to destabilization. Therefore, it should be noted that the plant model developed in this chapter assume a linear time-invariant (LTI) system. In this chapter, the dynamic equations for the robotic walker's two-wheeled model are derived and presented in state-space form followed by an open-loop analysis illustrating the stability of the system without any controllers.

4.1. Nomenclature

For reference, all principal variables and symbols used in this section have been listed below:

m_P	Mass of the pendulum
m_W	Mass of the wheel
D_1	Distance between left and right wheels along the y-axis
l_{g1}	Distance from point *O* to the center of mass, *CoM*, of the pendulum
r	Radius of the wheel
g	Gravitational acceleration
x	Displacement of the vehicle along the x-axis
ϕ_P	Linearized tilt angle of the pendulum
ψ	Heading angle of the vehicle
x_{LW}, x_{RW}	Displacements of the left and right wheels along the x-axis
θ_{LW}, θ_{RW}	Rotational angles of the left and right wheels
T_{d_ϕ}	Disturbance torque acting about the handlebar's y-axis or pitch axis
T_{d_ψ}	Disturbance torque acting about the handlebar's z-axis or yaw axis
T_{LW}, T_{RW}	Torques supplied by the actuators to the left and right wheels
J_W	Moment of inertia of the wheel about the y-axis
J_{P_y}	Combined moment of inertia of the pendulum and chassis about the y-axis
J_{P_z}	Combined moment of inertia of the pendulum and chassis about the z-axis

4.2. Dynamic Analysis and Equations of Motion

The physical representation of system dynamics has been simplified to a TWIP with a handlebar and disturbance torques acting about handlebar's pitch axis, T_{d_ϕ}, and yaw axis, T_{d_ψ}, as shown in Fig. 4-1. Linear displacement of the vehicle is denoted as x, angular rotation about the y-axis (pitch) as θ_P, and angular rotation about the z-axis (yaw) as ψ. The vehicle is actuated by two brushless DC motors that drive the left and right wheels along the x-axis (x_{LW}, x_{RW}) by supplying torques T_{LW} and T_{RW} to the left and right wheels, respectively.

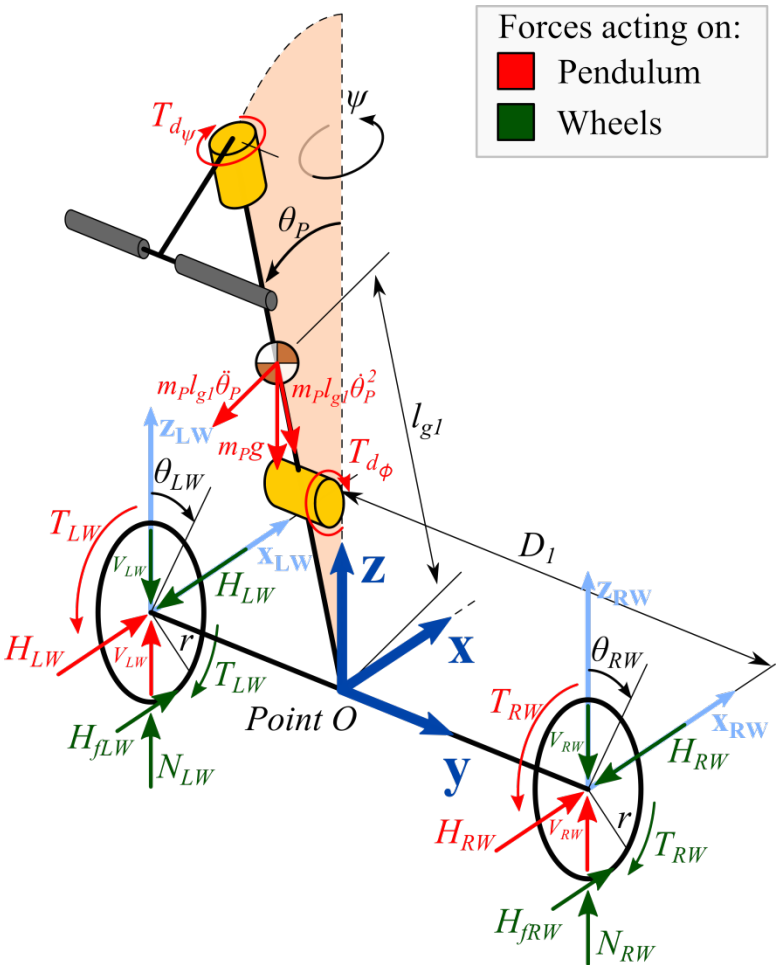

Figure 4-1. Diagram of forces and moments acting on the chassis and wheels of the TWIP system.

Using Newtonian mechanics, the equations of motion for the left wheel, which are completely analogous to the right wheel, that will be used to derive the state space representation of the system are as follows:

$$m_W \ddot{x}_{LW} = -H_{LW} + H_{fLW} \tag{4.1}$$

$$J_W \ddot{\theta}_{LW} = T_{LW} - H_{fLW} r \tag{4.2}$$

Combining and rearranging (4.1) and (4.2) leads to the following equation describing the dynamics of the left wheel:

$$m_W \ddot{x}_{LW} = \frac{1}{r} T_{LW} - \frac{J_W}{r} \ddot{\theta}_{LW} - H_{LW} \tag{4.3}$$

The dynamic equation for the left wheel (4.3) and the equivalent equation for the right wheel are then combined to yield:

$$m_W \left(\ddot{x}_{LW} + \ddot{x}_{RW} \right) + \frac{J_W}{r} \left(\ddot{\theta}_{LW} + \ddot{\theta}_{RW} \right) = \frac{1}{r} \left(T_{LW} + T_{RW} \right) - \left(H_{LW} + H_{RW} \right) \tag{4.4}$$

To simplify equation (4.4), two relationships are defined. The first relationship correlates the distance traveled by the left and right wheels (x_{LW}, x_{RW}) to the vehicle's linear displacement along the x-axis (x). The second relationship correlates the angular displacement covered by the left and right wheels (θ_{LW}, θ_{RW}) to the vehicle's linear displacement along the x-axis (x). These relationships are defined as:

$$x = \frac{x_{LW} + x_{RW}}{2} \tag{4.5}$$

$$\theta_{LW} + \theta_{RW} = \frac{x_{LW} + x_{RW}}{r} \tag{4.6}$$

Upon combining the two relationships above with (4.4) and rearranging the result, the following equation ensues:

$$2 \left(m_W + \frac{J_W}{r^2} \right) \ddot{x} = \frac{1}{r} \left(T_{LW} + T_{RW} \right) - \left(H_{LW} + H_{RW} \right) \tag{4.7}$$

The dynamics of the pendulum, which includes the chassis of the vehicle, can be described using the following equations:

$$m_P \ddot{x} = H_W - m_P l_{g1} \ddot{\theta}_P \cos\theta_P + m_P l_{g1} \dot{\theta}_P^2 \sin\theta_P \tag{4.8}$$

$$m_P \ddot{x} \cos\theta_P = H_W \cos\theta_P + V_W \sin\theta_P - m_P g \sin\theta_P - m_P l_{g1} \ddot{\theta}_P \tag{4.9}$$

$$J_{P_y} \ddot{\theta}_P = -H_W l_{g1} \cos\theta_P - V_W l_{g1} \sin\theta_P - (T_{LW} + T_{RW}) + T_{d_\phi} \tag{4.10}$$

$$J_{P_z} \ddot{\psi} = \frac{D_1}{2}(H_{LW} - H_{RW}) + T_{d_\psi} \tag{4.11}$$

where

$$H_W = H_{LW} + H_{RW} \tag{4.12}$$

$$V_W = V_{LW} + V_{RW} \tag{4.13}$$

and H_{LW}, H_{RW}, H_{fLW}, H_{fRW}, V_{LW}, and V_{RW} correspond to the reaction forces acting between the chassis, wheels, and ground.

By combining equations (4.7) through (4.13) and linearizing the resulting nonlinear system of equations at the upright equilibrium point using small angle approximation ($\theta_P = \pi + \phi_P$), the linearized equations are obtained as:

$$\ddot{x} = \frac{m_P^2 g l_{g1}^2}{\alpha}\phi_P + \frac{J_\theta - m_P l_{g1} r}{r\alpha}(T_{LW} + T_{RW}) + \frac{m_P l_{g1}}{\alpha} T_{d_\phi} \tag{4.14}$$

$$\ddot{\phi}_P = \frac{m_P g l_{g1} \beta}{\alpha}\phi_P + \frac{m_P l_{g1} - r\beta}{r\alpha}(T_{LW} + T_{RW}) + \frac{\beta}{\alpha} T_{d_\phi} \tag{4.15}$$

$$\ddot{\psi} = \frac{D_1}{2rJ_\psi}(T_{LW} - T_{RW}) + \frac{1}{J_\psi} T_{d_\psi} \tag{4.16}$$

where

$$J_\psi = \frac{D_1^2}{2}\left(m_W + \frac{J_W}{r^2}\right) + J_{P_z} \tag{4.17}$$

$$J_\theta = J_{P_y} + m_P l_{g1}^2 \tag{4.18}$$

$$\beta = 2m_W + \frac{2J_W}{r^2} + m_P \tag{4.19}$$

$$\alpha = J_\theta \beta - m_P^2 l_{g1}^2 \tag{4.20}$$

4.3. State-Space Representation

The general state-space representation of a continuous LTI system can be written in the following form:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) \end{aligned} \tag{4.21}$$

where $\mathbf{x}(t)$ is the state vector, $\mathbf{y}(t)$ is the output vector, $\mathbf{u}(t)$ is the input vector, **A** is the state matrix, **B** is the input matrix, and **C** is the output matrix. Rewriting equations (4.14) through (4.16) into state-space form yields the following coupled state-space representation:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\phi}_P \\ \ddot{\phi}_P \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_{23} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & A_{43} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi_P \\ \dot{\phi}_P \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ B_{21} & B_{22} & B_{23} & 0 \\ 0 & 0 & 0 & 0 \\ B_{41} & B_{42} & B_{43} & 0 \\ 0 & 0 & 0 & 0 \\ B_{61} & B_{62} & 0 & B_{64} \end{bmatrix} \begin{bmatrix} T_{LW} \\ T_{RW} \\ T_{d_\phi} \\ T_{d_\psi} \end{bmatrix} \tag{4.22}$$

where

$$A_{23} = \frac{m_P^2 g l_{g1}^2}{\alpha} \quad A_{43} = \frac{m_P g l_{g1} \beta}{\alpha} \tag{4.23}$$

$$B_{21} = \frac{J_\theta - m_P l_{g1} r}{r\alpha} \quad B_{22} = \frac{J_\theta - m_P l_{g1} r}{r\alpha} \quad B_{23} = \frac{m_P l_{g1}}{\alpha} \tag{4.24}$$

$$B_{41} = \frac{m_P l_{g1} - r\beta}{r\alpha} \quad B_{42} = \frac{m_P l_{g1} - r\beta}{r\alpha} \quad B_{43} = \frac{\beta}{\alpha} \tag{4.25}$$

$$B_{61} = \frac{D_1}{2rJ_\psi} \quad B_{62} = -\frac{D_1}{2rJ_\psi} \quad B_{64} = \frac{1}{J_\psi} \tag{4.26}$$

Meanwhile, since the initial controller implemented to control the robotic walker is a full-state feedback controller, LQR, all states must be measured and, as a result, the output vector, **C**, is defined as a 6*x*6 identity matrix.

In order to eliminate the effects of loop interaction and impose the desired dynamics on the system, the decoupling control scheme developed in [10] was implemented. The control scheme essentially decouples the dynamics of the two-wheeled system into two separate system of equations. One set of equations represents the linear displacement and rotational dynamics of the system about the y-axis (pitch axis) while the other set represents the rotational dynamics of the system about the z-axis (yaw axis). The decoupling controller can be described as follows:

$$\begin{bmatrix} T_\phi \\ T_\psi \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} T_{LW} \\ T_{RW} \end{bmatrix} \tag{4.27}$$

where $G_{11} = 0.5$, $G_{12} = 0.5$, $G_{21} = 0.5$, and $G_{22} = -0.5$.

To implement the decoupling control scheme, the decoupling controller defined in (4.27) was applied to (4.22). The result is two decoupled state-space equations representing the rotational dynamics of the system about the y-axis and rotational dynamics of the system about the z-axis, respectively:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\phi}_P \\ \ddot{\phi}_P \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & A_{23} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & A_{43} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi_P \\ \dot{\phi}_P \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ B_2 & B_{23} & 0 \\ 0 & 0 & 0 \\ B_4 & B_{43} & 0 \end{bmatrix} \begin{bmatrix} T_\phi \\ T_{d_\phi} \\ T_{d_\psi} \end{bmatrix} \tag{4.28}$$

$$\begin{bmatrix} \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ B_6 & 0 & B_{64} \end{bmatrix} \begin{bmatrix} T_{\psi} \\ T_{d_{\phi}} \\ T_{d_{\psi}} \end{bmatrix} \tag{4.29}$$

where $B_2 = B_{21} = B_{22}$, $B_4 = B_{41} = B_{42}$, and $B_6 = B_{61} = -B_{62}$.

Now that a state-space representation of the vehicle has been formulated, a robustly tuned LQR controller can be implemented and its performance evaluated. However, before discussing control system design, the following section will cover the conversion process from state-space to transfer function representation and conduct an open-loop analysis of the system.

4.4. Open-Loop Analysis

In single-input single-output (SISO) systems, the mathematical representation of the relation between the input and output of the LTI plant can be described by a single transfer function. Multiple-input multiple-output (MIMO) systems, on the other hand, requires one transfer function for every input to output combination. The TWIP robotic walker presented in this report consists of two inputs and six outputs. This means that 12 transfer functions are necessary to completely represent the relationships between each input and output in the system.

The conversion from state-space to transfer function can be accomplished by taking the Laplace transform of the general state-space, Eq. (4.21), and solving for the ratio of $Y(s)$ to $U(s)$. The result is a transfer matrix defined by:

$$H(s) = \frac{Y(s)}{U(s)} = \mathbf{C}\left(s\mathbf{I} - \mathbf{A}\right)^{-1}\mathbf{B} \tag{4.30}$$

Using Eq. (4.30), the transfer matrix for the TWIP robotic walker was calculated to be:

$$H(s)=\begin{bmatrix} \dfrac{0.9992s^2+1.331\times10^{-14}s-28.41}{s^4-1.776\times10^{-15}s^3-47.49s^2} & \dfrac{0.9992s^2+1.331\times10^{-14}s-28.41}{s^4-1.776\times10^{-15}s^3-47.49s^2} \\ \dfrac{0.9992s^2-28.41}{s^3-1.776\times10^{-15}s^2-47.49s} & \dfrac{0.9992s^2-28.41}{s^3-1.776\times10^{-15}s^2-47.49s} \\ \dfrac{1.588}{s^2-1.776\times10^{-15}s-47.49} & \dfrac{1.588}{s^2-1.776\times10^{-15}s-47.49} \\ \dfrac{1.588s+1.411\times10^{-15}}{s^2-1.776\times10^{-15}s-47.49} & \dfrac{1.588s+1.411\times10^{-15}}{s^2-1.776\times10^{-15}s-47.49} \\ \dfrac{6.006}{s^2} & -\dfrac{6.006}{s^2} \\ \dfrac{6.006}{s} & -\dfrac{6.006}{s} \end{bmatrix} \quad (4.31)$$

Using the transfer matrix defined above, Eq. (4.31), the open loop response of the TWIP plant to a unit step input has been simulated as observed in Fig. 4-2. By inspecting Fig. 4-2, it can be seen that all 12 transfer functions are diverging. This means that these transfer functions must contain at least one or more unstable poles. Based on these open loop responses, it is quite evident that the system is highly unstable. Thus, the implementation of a controller is critical.

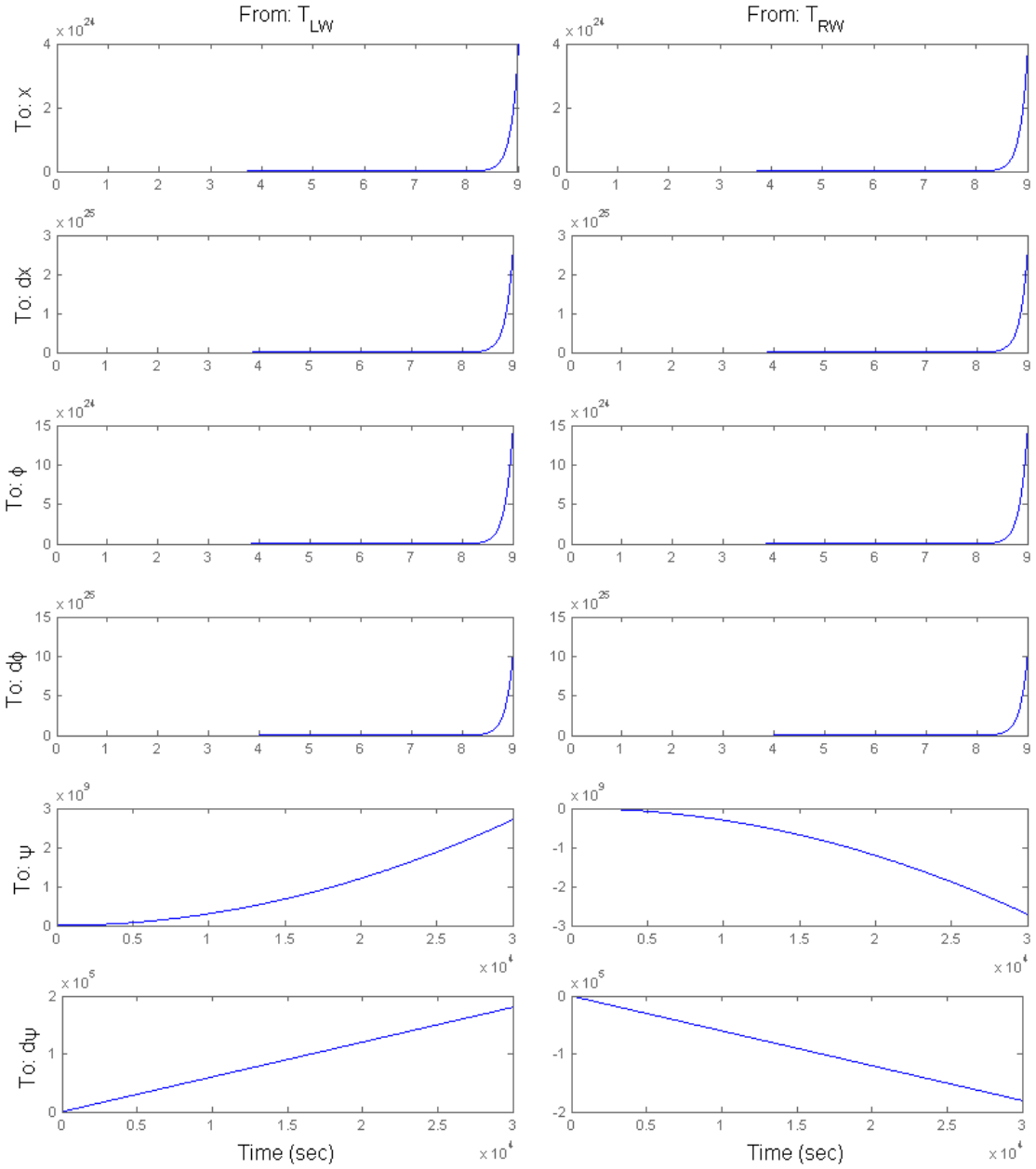

Figure 4-2. Open loop unit step response of the transfer matrix defined in Eq. (4.31). Each column represents one of the two input variables (T_{LW}, T_{RW}) while each row represents one of the six output variables (x, $\dot{x}$, ϕ, $\dot{\phi}$, ψ, $\dot{\psi}$). Each subplot represents the relationship between an input and an output.

CHAPTER 5

CONTROL SYSTEM DESIGN

There exist a wide variety of control methods available to the designer of a control system and, in most cases, more than a single control scheme will be able to satisfy the design requirements of the designer. With so many control methods to select from, the simplest approach usually involves identifying the physical behavior and operating conditions of the system and using the specified design requirements to narrow down the potential control methods. Generally, if the dynamics of the plant model are uniform, free of noise and disturbances then non-adaptive control may be the preferred method. On the other hand, if the parameters or operating conditions of the system are constantly changing and the response of the system is being significantly corrupted by noise and external disturbances then an adaptive control method might play a better role.

For this project, the plant model is a two-wheeled inverted pendulum with external disturbance forces applied at the handlebars. The dynamics of the inverted pendulum is highly nonlinear and inherently unstable. Since the pitch angle of the vehicle will be restricted to ±20 degrees from its upright equilibrium point, the mathematical representation of the system can be considerably simplified without causing a significant penalty to model accuracy by linearizing the equations of motion [50]. Note that ±20 degrees is typically a stable operating range for a balancing inverted pendulum and a valid range for small angle approximation. Using the linearized model of the system, a non-adaptive control scheme such as linear quadratic regulator (LQR) control becomes quite capable of robustly stabilizing the vehicle [13], [14], [16], [19], [20], [26].

This chapter is focused on the development of a control strategy for dictating the behavior of a beneficial self-balancing robotic walker that is suitable for older adults and mobility-impaired users. Note that the simulations conducted in this chapter are intended to provide a rough estimate of the two-wheeled prototype response rather than an exact representation. Section 5.1 will provide an overview of the control approach. Following control approach overview, the design process for developing the LQR and state variable controllers responsible stabilizing the two-wheeled system and regulating its position will be presented. Meanwhile, the focus of Section 5.3 will be the discussion of simulation results generated by each controller. To close the chapter, the last section will evaluate and discuss the performance of the proposed control scheme.

5.1. Control Approach Overview

The use of a TWIP platform configuration creates delicate control problems that must be carefully addressed. Unlike three-wheeled or four-wheeled walkers, a two-wheeled walker is an inherently unstable system with highly nonlinear dynamics that must be actively balanced to remain upright. In addition to developing a controller to stabilize the system, it is also imperative that the device behaves in a predictable and controllable manner that can promote user stability without inducing risks or creating harm.

The task of balancing the two-wheeled system was accomplished by two decoupled state-space controllers that were designed using the LQR method. Improved disturbance rejection was achieved through the implementation of a pitch controller. To

regulate the vehicle's linear displacement and heading angle with smooth and minimum jerk control, two additional state variable controllers were developed and tested.

5.2. Control Strategy

The control block diagram, illustrated in Fig. 5-1, consists of multiple controllers that have been designed to implement two control schemes: decoupled LQR control and state variable control. The primary control scheme, decoupled LQR control, is designed to optimally and robustly balance the two-wheeled system in the upright position. Meanwhile, state variable control is the secondary control scheme which is comprised of three individual controllers that are responsible for regulating the two-wheeled system's linear displacement, pitch angle, and heading angle. To better explain how the control scheme operates, a brief overview of the events occurring during each control scheme will be provided in the following paragraphs.

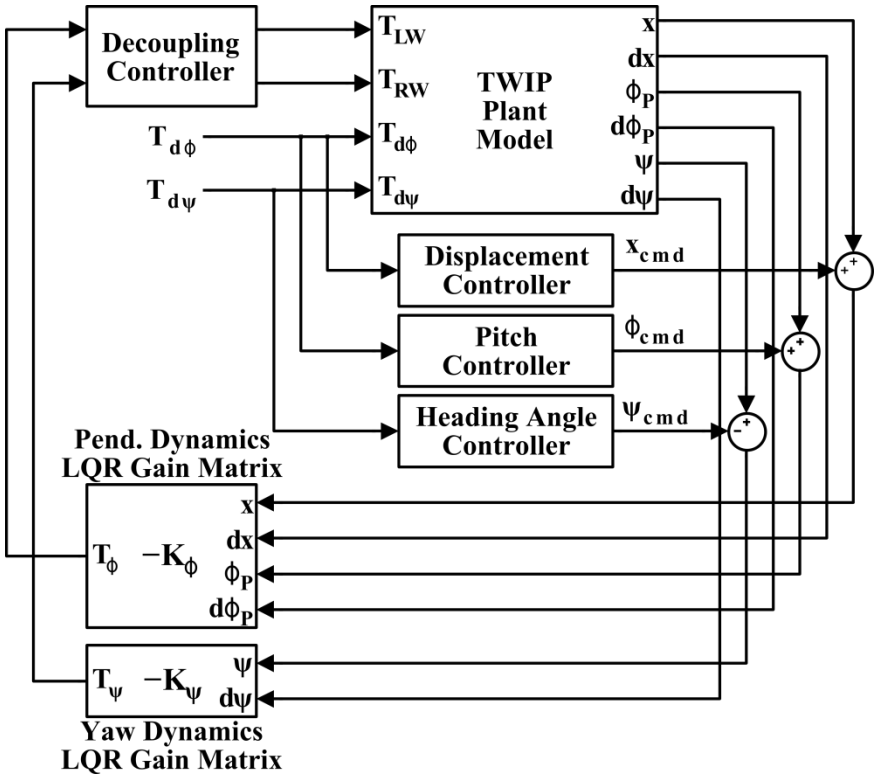

Figure 5-1. Control block diagram for the two-wheeled system.

In the decoupled LQR control scheme, the first event occurs at the TWIP plant model, represented by the coupled state-space representation described in Eq. (4.22), where six state variables are calculated and fed back to the decoupled LQR controllers. Four of these state variables are fed to the LQR gain matrix regulating pendulum dynamics while the other two state variables are fed to the LQR gain matrix regulating yaw dynamics. These LQR gain matrices produce two output signals corresponding to the torques about pitch and yaw axes that will fulfill the LQR objectives to optimally stabilize the system and return all state variables to zero. The decoupling controller defined in the previous chapter is then implemented to transform the LQR output signals (T_ϕ, T_ψ) into torque commands for the left and right motors (T_{LW}, T_{RW}). The motor torque

commands are fed into the TWIP plant model, a new set of state variables are produced, and the decoupled LQR control scheme repeats.

In the state variable control scheme, three individual controllers are used to directly influence the linear displacement, pitch angle, and heading angle outputs of the TWIP plant model. The displacement controller calculates linear displacement commands based on the disturbance torque acting at the handlebar about its pitch axis. To calculate pitch angle commands, the pitch controller also relies on measurements of disturbance torque acting at the handlebar about its pitch axis. The heading angle controller, on the other hand, uses the disturbance torque acting at the handlebar about its yaw axis to calculate heading angle commands. The following subsections will provide further detail about the design of the LQR and state variable controllers.

5.2.1. Linear Quadratic Regulator (LQR) Controller

The task of the LQR controller is to calculate the optimal state feedback gain matrix **K** that will drive the steady-state error to zero based on a set of constraints defined by the quadratic cost function:

$$J = \frac{1}{2}\int_0^\infty \left[\mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) + \mathbf{u}^T(t)\mathbf{R}\mathbf{u}(t) \right] dt \tag{5.1}$$

where **Q** and **R** are symmetric positive-definite matrices that are used to set the relative weights of state deviation and input usage, respectively. Once the relative importance of the control effort (i.e. applied motor torques T_{LW} and T_{RW}) and steady-state error are specified, the value of this cost function is minimized using the following feedback control law:

$$\mathbf{u} = -\mathbf{K}\mathbf{x} \tag{5.2}$$

where the state feedback gain matrix **K** is computed using the following identity:

$$\mathbf{K} = \mathbf{R}^{-1}\mathbf{B}^{T}\mathbf{P} \tag{5.3}$$

Upon combining Eqs. (5.1), (5.2), and (5.3) and solving the infinite integral, the result is a matrix quadratic equation, known as the Algebraic Riccati Equation (ARE):

$$\mathbf{A}^{T}\mathbf{P} + \mathbf{P}\mathbf{A} + \mathbf{Q} - \mathbf{P}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^{T}\mathbf{P} = 0 \tag{5.4}$$

Provided that (**A**, **B**) is controllable and ($\mathbf{Q}^{0.5}$, **A**) is observable, matrix **P** in Eq. (5.4) can be solved and then substituted into Eq. (5.3) in order to calculate the state feedback gain matrix **K** that will impose the desired closed loop behavior on the system.

Since the disturbance forces acting on the system are uncertainties that cannot be controlled or predicted by the LQR method, all disturbance forces must be omitted from the state-space equations before calculating the gain matrix. To minimize the cost function and drive the state variables to zero as time goes to infinity, the values for the state **Q** and **R** weighting matrices were tuned based on the desire to minimize pitch displacement and linear displacement. While tuning the weighting matrices, the primary objective was to minimize overshoot, settling time, and steady-state error for ϕ_P and $\dot{\phi}_P$. Therefore, the gains for ϕ_P and $\dot{\phi}_P$ were kept relatively high in comparison to the other state variables. For simplicity, only the diagonal entries of the weighting matrices were tuned since the performance of the controller is affected the most when the state and input variables of the system are penalized individually.

5.2.2. Displacement Controller

The displacement controller is responsible for regulating the linear displacement of the two-wheeled system with smooth and minimum jerk control using measurements

of disturbance torque acting at the handlebar about the pitch axis (T_{d_ϕ}). To achieve smooth and minimum jerk control, the displacement controller employs a PI controller as well as a minimum jerk trajectory algorithm. A Simulink block diagram of the displacement controller can be seen in Fig. 5-2. For a script of the *MinJerkTraj* embedded function used in the Simulink diagram, refer to Appendix A.

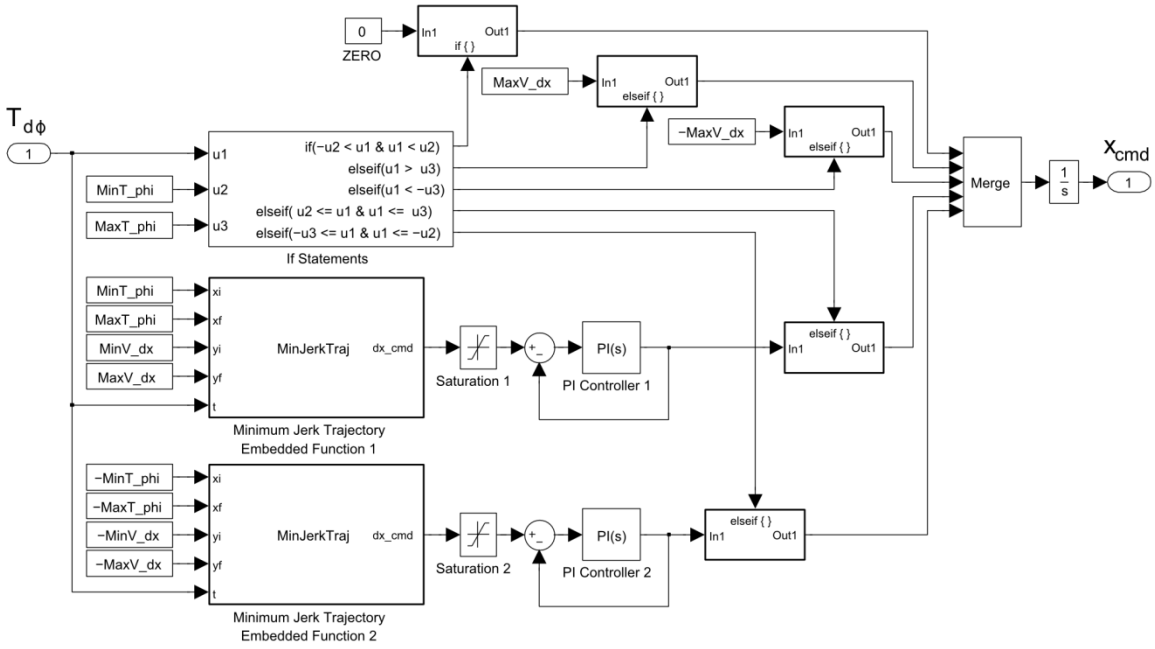

Figure 5-2. Simulink block diagram of the displacement controller.

Once the disturbance torque (T_{d_ϕ}) enters the displacement controller, an IF Statement block is used to determine the rate of change of the vehicle's linear displacement. If the measured disturbance torque (T_{d_ϕ}) is close to zero then the controller will set the linear velocity to zero. If the measured disturbance torque (T_{d_ϕ}) is greater than or less than a specific torque value then the controller will set the linear velocity to positive maximum velocity or negative maximum velocity, respectively. However, if the

measured disturbance torque (T_{d_ϕ}) falls within a certain boundary then the controller will use a minimum jerk trajectory algorithm to calculate a linear velocity profile that smoothly saturates to a minimum and maximum.

To achieve minimum jerk control, the following modified version of Hogan's minimum jerk trajectory formula [51] was implemented:

$$y(t) = y_i + \left(y_f - y_i\right)\left[10\left(\frac{t - x_i}{x_f - x_i}\right)^3 - 15\left(\frac{t - x_i}{x_f - x_i}\right)^4 + 6\left(\frac{t - x_i}{x_f - x_i}\right)^5\right] \tag{5.5}$$

where $y(t)$ is the trajectory path at input variable t, x_i is the initial x-position, x_f is final x-position, y_i is the initial y-position, y_f is the final y-position, and t is the input variable. For the displacement controller, the adjustable parameters (x_i, x_f, y_i y_f) correspond to torque and linear velocity conditions (MinT_phi, MaxT_phi, MinV_dx, MaxV_dx), respectively, specified by the user.

A problem with using the modified minimum jerk trajectory algorithm is that if the measured disturbance torque (T_{d_ϕ}) is noisy or unsteady, the calculated trajectory path will also be noisy and unsteady. To address this problem, a PI controller with a proportional gain of 0.1 and integral gain of 10 was added at the output of the minimum jerk trajectory algorithm. Finally, once the IF Statement block outputs a linear velocity, the integral of this value is calculated and the vehicle's linear displacement command x_{cmd} is obtained. The general waveform for the linear velocity produced by IF statements inside the displacement controller is illustrated in Fig. 5-3. The waveform below essentially depicts the minimum and maximum linear velocities that the displacement

controller is allowed to drive the platform based on the specified torque and linear velocity conditions: (x_i, x_f, y_i y_f) = (MinT_phi, MaxT_phi, MinV_dx, MaxV_dx).

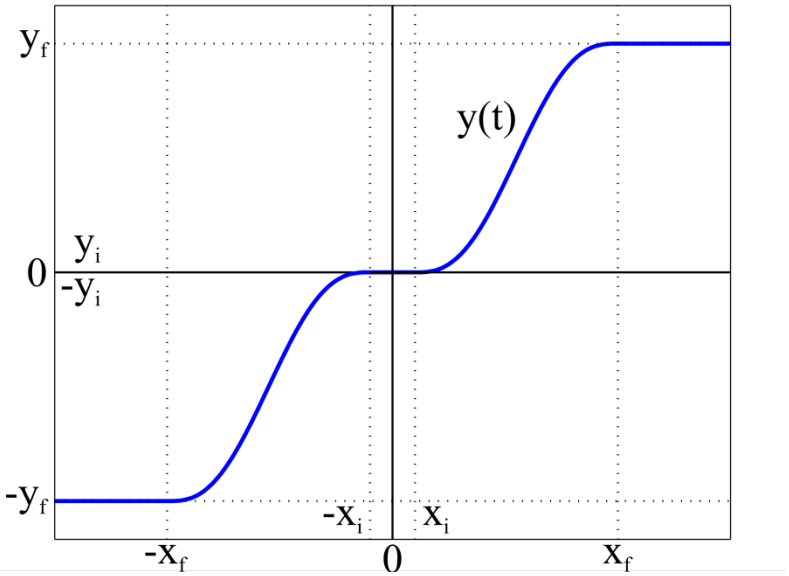

Figure 5-3. General waveform produced by the state variable controllers prior to integrating velocity with respect to time.

5.2.3. Pitch Controller

The LQR controller was designed to robustly stabilize the system at its equilibrium point but LQR control by itself is insufficient to achieve the desired behavior from the vehicle. Another step towards achieving the desired system behavior is the implementation of a pitch controller. The purpose of this controller is to mitigate the effect of external disturbance forces acting on the handlebars of the system. Disturbance mitigation is accomplished by adjusting the pitch of the pendulum in order to offset the vehicle's center of mass and generate a counter torque. A secondary purpose for adjusting the pitch of the vehicle is to create a larger distance between the user's feet and the base of the device, allowing the user to take larger steps without hitting the device.

As it can be seen from Fig. 5-4, the pitch controller has been designed in the same way as the displacement controller aside from some minor differences. For a script of the *MinJerkTraj* embedded function used in the Simulink diagram, refer to Appendix A. Unlike the displacement controller, the response of the pitch controller is defined by the torque and pitch angle conditions: (x_i, x_f, y_i y_f) = (MinT_phi, MaxT_phi, MinA_phi, MaxA_phi). Another difference between the two controllers is that the pitch controller does not need to integrate velocity to obtain the pitch angle command (ϕ_{cmd}). Without the need to integrate velocity, the response of the pitch controller can essentially be defined by the waveform displayed in Fig. 5-3.

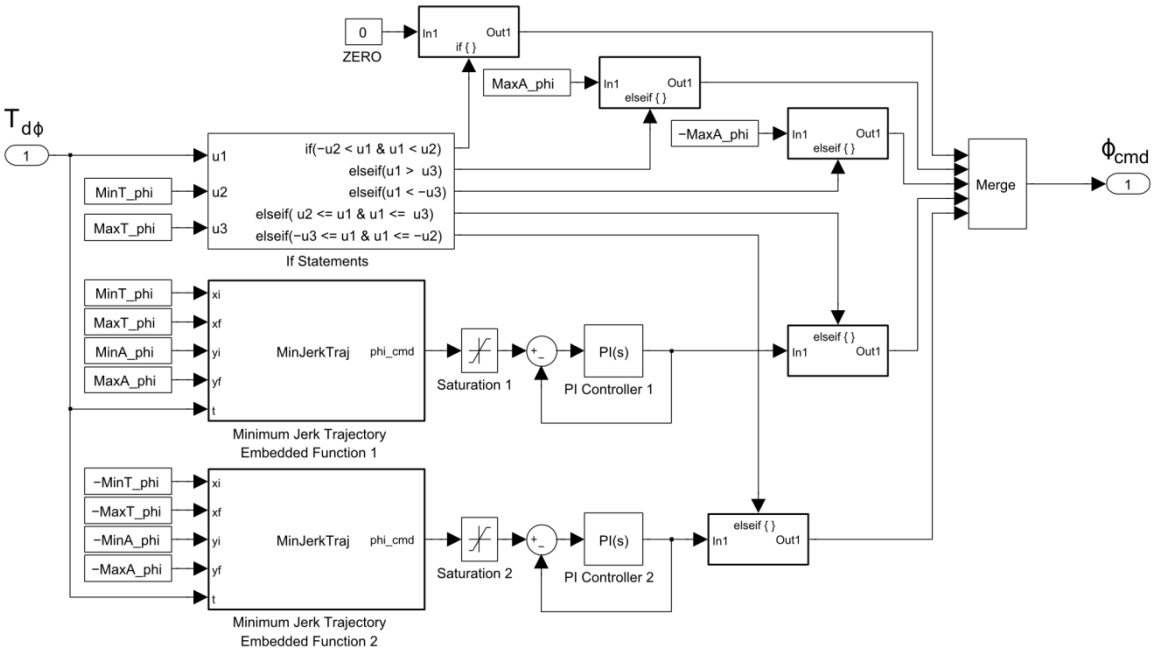

Figure 5-4. Simulink block diagram of the pitch controller.

5.2.4. Heading Angle Controller

The heading angle controller is responsible for regulating the heading angle of the two-wheeled system with smooth and minimum jerk control using measurements of

disturbance torque acting at the handlebar about the yaw axis (T_{d_ψ}). A Simulink block diagram of the heading angle controller can be seen in Fig. 5-5. For a script of the *MinJerkTraj* embedded function used in the Simulink diagram, refer to Appendix A. Upon comparing Fig. 5-5 to Fig. 5-2, it can be seen that the heading angle controller has been designed to function in the exact same way as the displacement controller with the exception of different adjustable parameters, input and output signals. Unlike the displacement controller, the response of the heading angle controller is defined by the torque and heading velocity conditions: (x_i, x_f, y_i y_f) = (MinT_psi, MaxT_psi, MinV_dpsi, MaxV_dpsi). The other difference between the two controllers is that the heading angle controller uses measurements of disturbance torque acting at the handlebar about the yaw axis (T_{d_ψ}) as the input signal and heading angle commands (ψ_{cmd}) as the output signal.

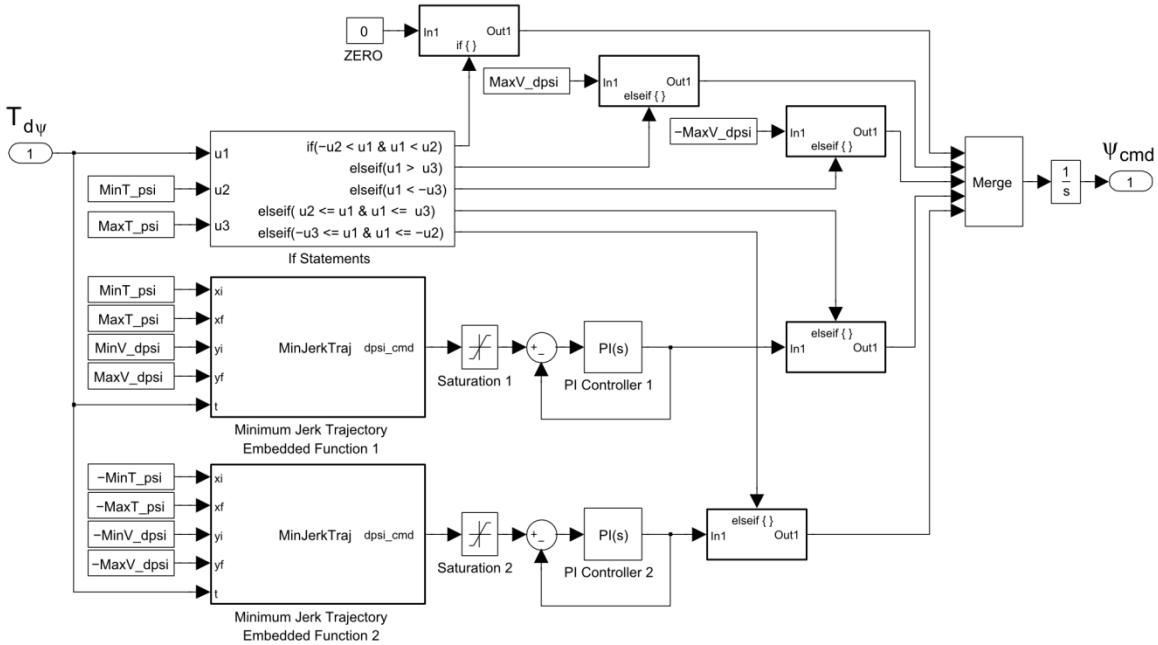

Figure 5-5. Simulink block diagram of the heading angle controller.

5.2.5. Decoupling Controller

The decoupling controller is responsible for transforming the output signals from the decoupled LQR controllers (T_ϕ, T_ψ) into torque commands for the left and right wheel motors (T_{LW}, T_{RW}). In other words, the decoupling controller is what allows the pendulum and yaw dynamics of the two-wheeled system to be independently controlled by two separate LQR controllers. The Simulink block diagram of the decoupling controller can be seen in Fig. 5-6. For the system of equations describing the decoupling controller, refer back to Eq. (4.27) in the previous chapter. To reiterate, the gains used for the decoupling controller were: $G_{11} = 0.5$, $G_{12} = 0.5$, $G_{21} = 0.5$, and $G_{22} = -0.5$.

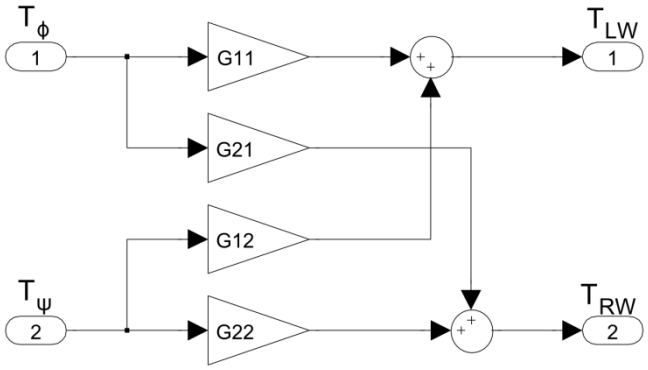

Figure 5-6. Simulink block diagram of the decoupling controller.

5.3. Simulation Results and Analysis

In order to evaluate the performance of the tuned LQR and state variable controllers, the response of each controller was simulated in MATLAB/Simulink and the results were analyzed. The TWIP plant model parameters used to simulate the response of the system are listed in Table 5-1. Although initial simulations were performed before prototype construction, the parameters in Table 5-1 as well as the simulation results

demonstrated in this section have been updated to better reflect the actual characteristics of the constructed prototype.

Table 5-1. Two-wheeled system parameters and description.

Variable	Description	Value	Unit
m_P	Combined mass of the pendulum and chassis	11.445	kg
m_W	Mass of the wheel	0.90	kg
D_1	Distance between left and right wheels along the y-axis	0.346	m
l_{g1}	Distance from point *O* to the center of mass, *CoM*, of the pendulum	0.30	m
r	Radius of the wheel	0.114	m
g	Gravitational acceleration	9.81	m/s^2
J_W	Moment of inertia of the wheel about the y-axis	0.00573	$kg \cdot m^2$
J_{P_y}	Combined moment of inertia of the pend. and chassis about the y-axis	1.0418	$kg \cdot m^2$
J_{P_z}	Combined moment of inertia of the pend. and chassis about the z-axis	0.2241	$kg \cdot m^2$

The combined mass of the pendulum and chassis (m_P) was calculated by measuring the total mass of the assembled system and subtracting this value from the measured mass of the wheels (m_W). The distance between left and right wheels along the y-axis (D_1) as we as the radius of the wheel (r) were collected from actual measurements of the prototype. The distance from point *O* to the center of mass, *CoM*, of the pendulum (l_{g1}) was measured by placing the prototype on a pivot and locating the prototype's point of equilibrium along the pendulum. To calculate the moment of inertia of the wheel about the y-axis (J_W), the geometry of the wheel was first simplified to a cylindrical shell with open ends and negligible shell thickness whose moment of inertia can be described by the following equation:

$$J_W = m_W R^2 \tag{5.6}$$

where J_W is the wheel's moment of inertia about the y-axis, m_W is the mass of the wheel, and R is the distance from the center of the wheel where all of its mass is distributed. Next, it was assumed that the distance from the center of the wheel where all of its mass is distributed (R) is equal to 70% of the actual radius of the wheel; resulting in R = 0.0855 m. Upon substituting m_W = 0.90 kg and R = 0.0798 m into equation (5.4), the moment of inertia of the wheel about the y-axis (J_W) was estimated to be 0.00573 kg·m^2.

Meanwhile, the combined moment of inertia of the pendulum and chassis about the y-axis (J_{P_y}) was estimated based on the mass properties of the two-wheeled model calculated in SolidWorks. Similarly, the combined moment of inertia of the pendulum and chassis about the z-axis (J_{P_z}) was also estimated using SolidWorks calculations of the solid model's mass properties. The material properties assigned to each component in the solid model varied between 6061 aluminum alloy, cast alloy steel, and polycarbonate/acrylonitrile butadiene styrene (PC-ABS). The mass of the electronics were assumed to be negligible compared to the moment of inertia contributed by the aluminum and steel components. To ensure reasonable estimations for the two-wheeled system's moments of inertia, the results were compared to the estimated properties of similar two-wheeled platforms in other research works [13], [15], [52].

5.3.1. Decoupled LQR Control

Since this section will only be analyzing the performance of the decoupled LQR control scheme, the state variable controllers have been disabled. Considering how the dynamics of the two-wheeled system has been decoupled, two separate LQR controllers

need to be designed and tuned. To control pendulum and yaw dynamics with optimal LQR control, the weighting matrices ($\mathbf{Q}_\phi$, $\mathbf{Q}_\psi$) and ($\mathbf{R}_\phi$, $\mathbf{R}_\psi$) were tuned until simulation results were displaying desired system performance. The robustly tuned weighting matrices for the decoupled LQR controllers are:

$$\mathbf{Q}_\phi = \begin{bmatrix} 10000 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_\phi = 1 \tag{5.7}$$

$$\mathbf{Q}_\psi = \begin{bmatrix} 1000 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{R}_\psi = 1 \tag{5.8}$$

The LQR gains corresponding to the weighting matrices above are as follows:

$$\mathbf{K}_\phi = \begin{bmatrix} -100.0000 & -60.7486 & 370.5861 & 80.6936 \end{bmatrix} \tag{5.9}$$

$$\mathbf{K}_\psi = \begin{bmatrix} 31.6228 & 3.6995 \end{bmatrix} \tag{5.10}$$

Additional information about how the LQR gains were calculated can be found in the MATLAB file listed in Appendix B.

The LQR gain matrices used to perform system simulations are listed in equations (5.9) and (5.10). To test the ability of the decoupled LQR control scheme to stabilize the two-wheeled system and reject external disturbance forces, the TWIP plant model was subjected to a 5 Nm ramp disturbance torque acting at the handlebar about the pitch axis (T_{d_ϕ}). The ramp profile of the input disturbance torque as well as the resulting system response is shown in Fig. 5-7.

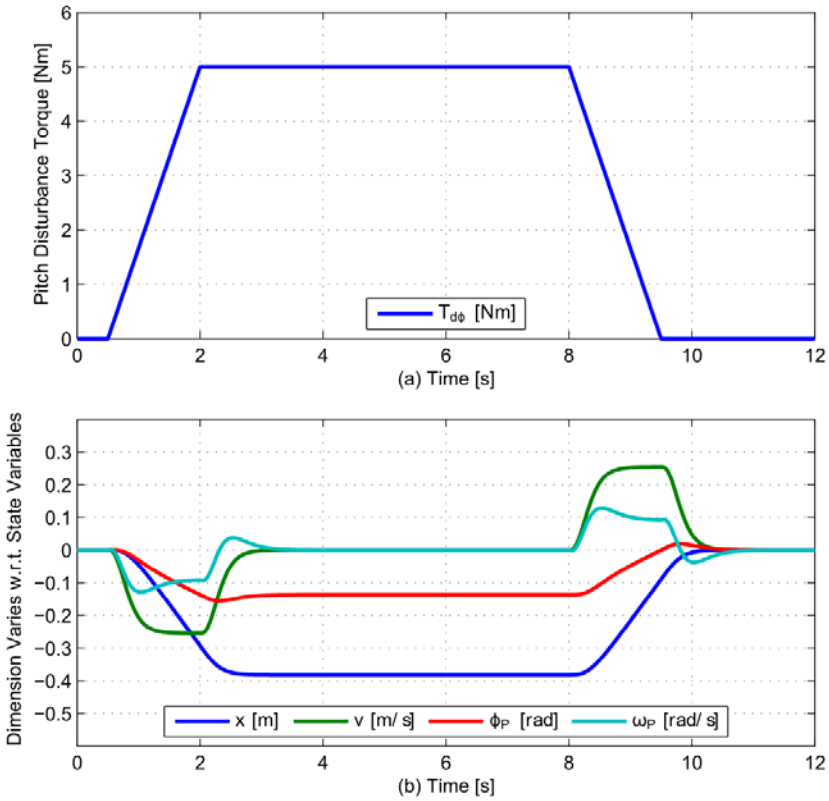

Figure 5-7. Simulation of (a) 5 Nm ramp disturbance torque acting at the handlebar about the pitch axis and the resulting (b) system response with decoupled LQR control.

In response to the disturbance torque profile depicted in Fig. 5-7a, the linear displacement and velocity of the two-wheeled system gradually changes but quickly reaches steady state with very low overshoot whenever the input torque stop changing. Despite the 5 Nm ramp disturbance torque acting at the handlebar about the pitch axis, the vehicle displaced a maximum linear distance of about 0.38 m which is an acceptable performance.

5.3.2. Displacement Control

This subsection will test the individual performance of the displacement controller as well as its ability to change the vehicle's linear displacement while the decoupled LQR control scheme is balancing the two-wheeled system. The torque and linear velocity

conditions (x_i, x_f, y_i y_f) = (MinT_phi, MaxT_phi, MinV_dx, MaxV_dx) were selected based on rough expectations of how the displacement controller should behave. Therefore, the lower and upper limits for the input disturbance torque (MinT_phi, MaxT_phi) were set to 0.05 Nm and 3 Nm, respectively, while the minimum and maximum linear velocities (MinV_dx, MaxV_dx) were set to 0 m/s and 0.8 m/s, respectively. The ramp profile of the input disturbance torque as well as the resulting responses for the displacement controller and TWIP system can be seen in Fig. 5-8.

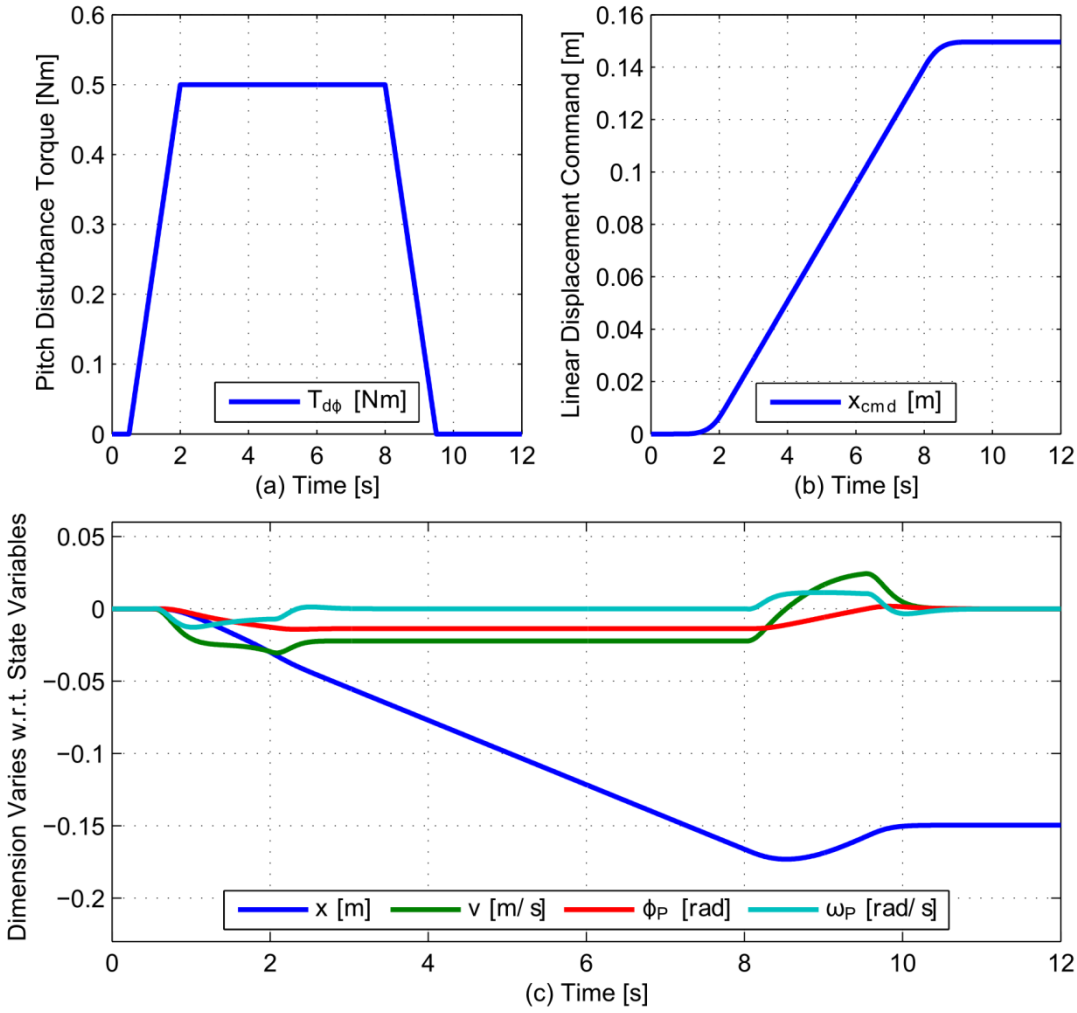

Figure 5-8. Simulation of (a) 0.5 Nm ramp disturbance torque acting at the handlebar about the pitch axis as well as the resulting (b) displacement controller response and (c) system response with decoupled LQR and displacement control.

In response to the pitch disturbance torque waveform depicted in Fig. 5-8a, the displacement controller generates a linear displacement command curve that is smooth and steady, as shown in Fig. 5-8b. The final linear displacement command calculated by the displacement controller is about 0.15 m. From Fig. 5-8c, it can be observed that the displacement controller has been successful in regulating the linear displacement of the two-wheeled system to 0.15 m with smooth and minimum jerk control.

5.3.3. Pitch Control

To test the ability of the pitch controller to regulate the pitch angle of the two-wheeled system and increase disturbance rejection, the TWIP plant model was subjected to a 0.5 Nm ramp disturbance torque acting at the handlebar about the pitch axis (T_{d_ϕ}). The torque and pitch angle conditions (x_i, x_f, y_i y_f) = (MinT_phi, MaxT_phi, MinA_phi, MaxA_phi) were selected based on rough expectations of how the pitch controller should behave. Therefore, the lower and upper limits for the input disturbance torque (MinT_phi, MaxT_phi) were set to 0.05 Nm and 4 Nm, respectively, while the minimum and maximum pitch angles (MinA_phi, MaxA_phi) were set to 0 rad and 0.2618 rad, respectively. The ramp profile of the input disturbance torque as well as the resulting responses for the pitch controller and TWIP system can be seen in Fig. 5-9.

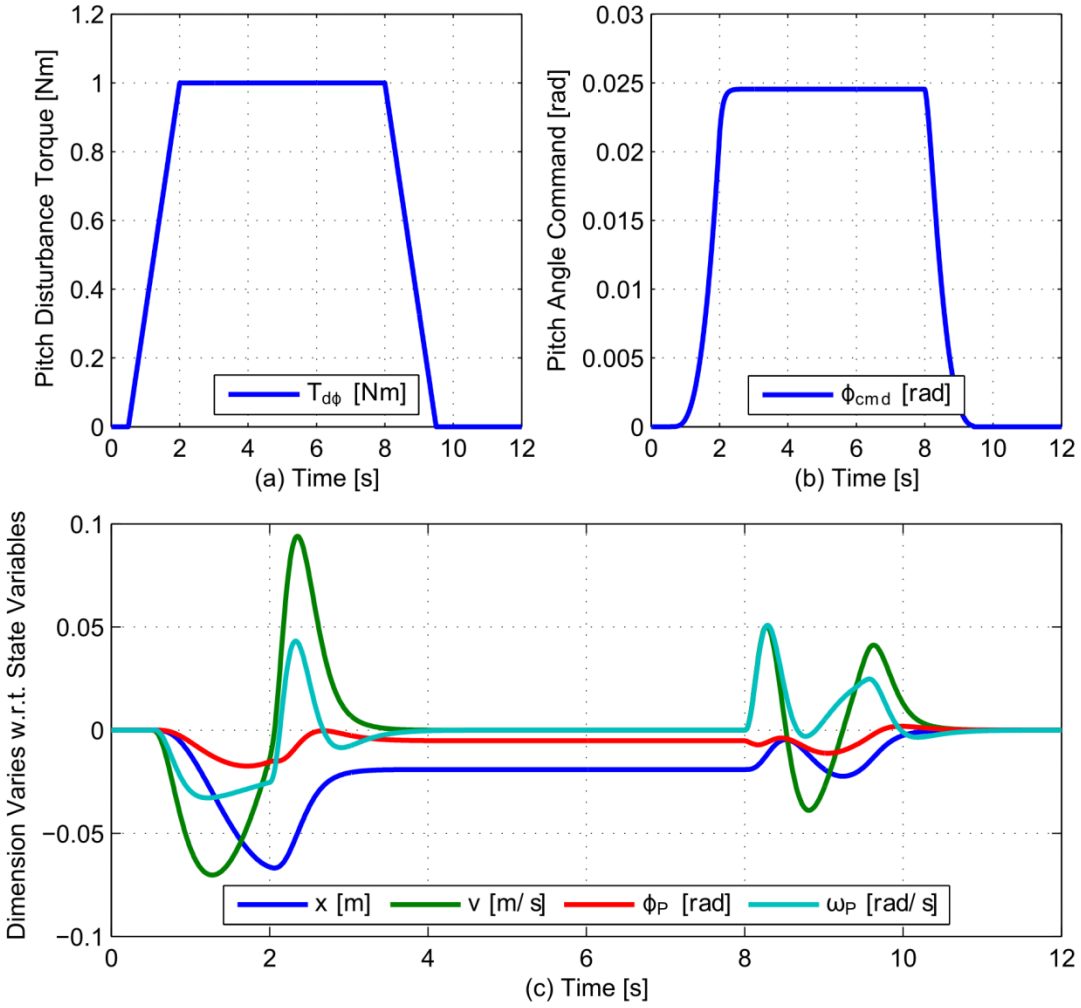

Figure 5-9. Simulation of (a) 1 Nm ramp disturbance torque acting at the handlebar about the pitch axis as well as the resulting (b) pitch controller response and (c) system response with decoupled LQR and pitch control.

As expected from the PI controller and minimum jerk trajectory algorithm, the pitch controller responds to the 1 Nm ramp disturbance torque waveform with smooth and minimum jerk control, as shown in Fig. 5-9b. The effect of the pitch controller on the response of the two-wheeled system, as observed in Fig. 5-9c, is reduced linear displacement which can be translated into improved disturbance rejection at the cost of slightly more effort from the motors. For a better illustration of pitch controller performance, a comparison between the response of the two-wheeled system with and

without pitch control can be seen in Fig. 5-10. Without pitch control, the vehicle's linear displacement reaches a steady state at 0.110 m from the equilibrium point but with pitch control, the total distance traveled by the vehicle is reduced to 0.0191 m. The reduction in linear displacement due to pitch control will vary with input disturbance torque but the objective to improve disturbance rejection by adjusting the vehicle's pitch angle has been accomplished.

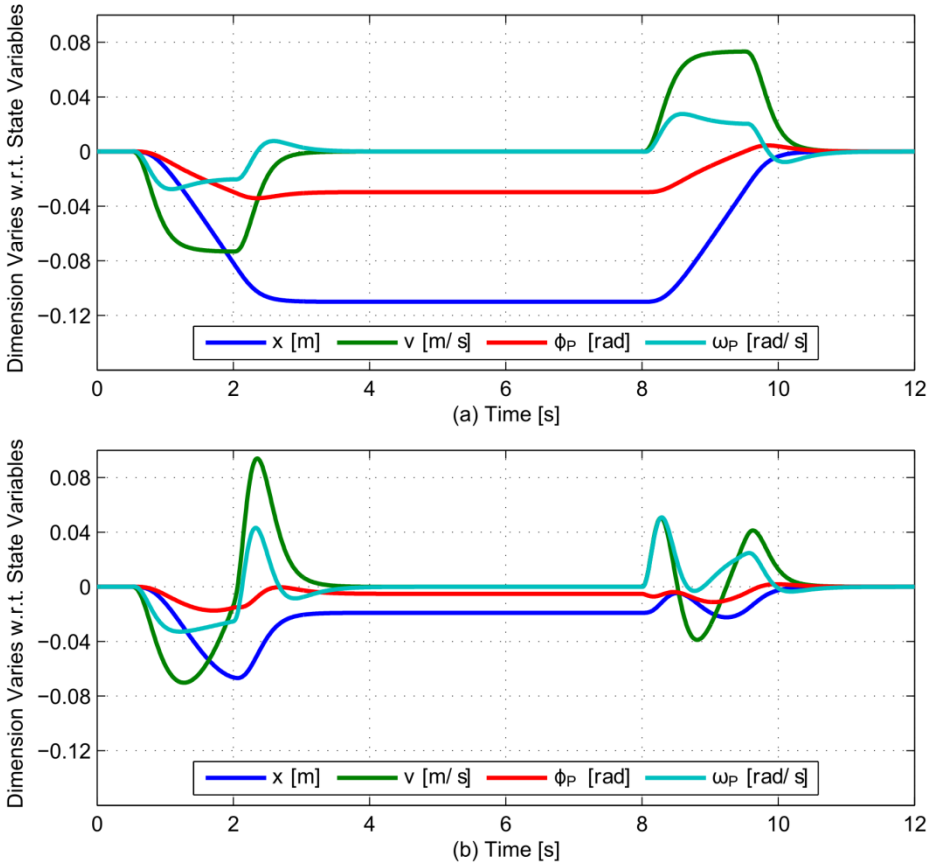

Figure 5-10. System response to a 1 Nm ramp disturbance torque acting at the handlebar about the pitch axis (a) without and (b) with pitch control.

5.3.4. Heading Angle Control

The performance of the heading angle controller was tested in the same way as the previous state variable controllers except that instead of applying a disturbance torque

at the handlebar about the pitch axis, a disturbance torque was applied about the yaw axis. The TWIP plant model was subjected to a 1 Nm ramp disturbance torque acting at the handlebar about the yaw axis (T_{d_ψ}) and the response of the heading angle controller as well as the two-wheeled system were simulated and analyzed. The torque and heading velocity conditions (x_i, x_f, y_i y_f) = (MinT_psi, MaxT_psi, MinV_dpsi, MaxV_dpsi) were also selected based on rough expectations of how the heading angle controller should behave. Therefore, the lower and upper limits for the input disturbance torque (MinT_psi, MaxT_psi) were set to 0.05 Nm and 1 Nm, respectively, while the minimum and maximum yaw velocities (MinV_dpsi, MaxV_dpsi) were set to 0 rad/s and 1 rad/s, respectively. The ramp profile of the input disturbance torque as well as the resulting responses for the heading angle controller and TWIP system can be seen in Fig. 5-11.

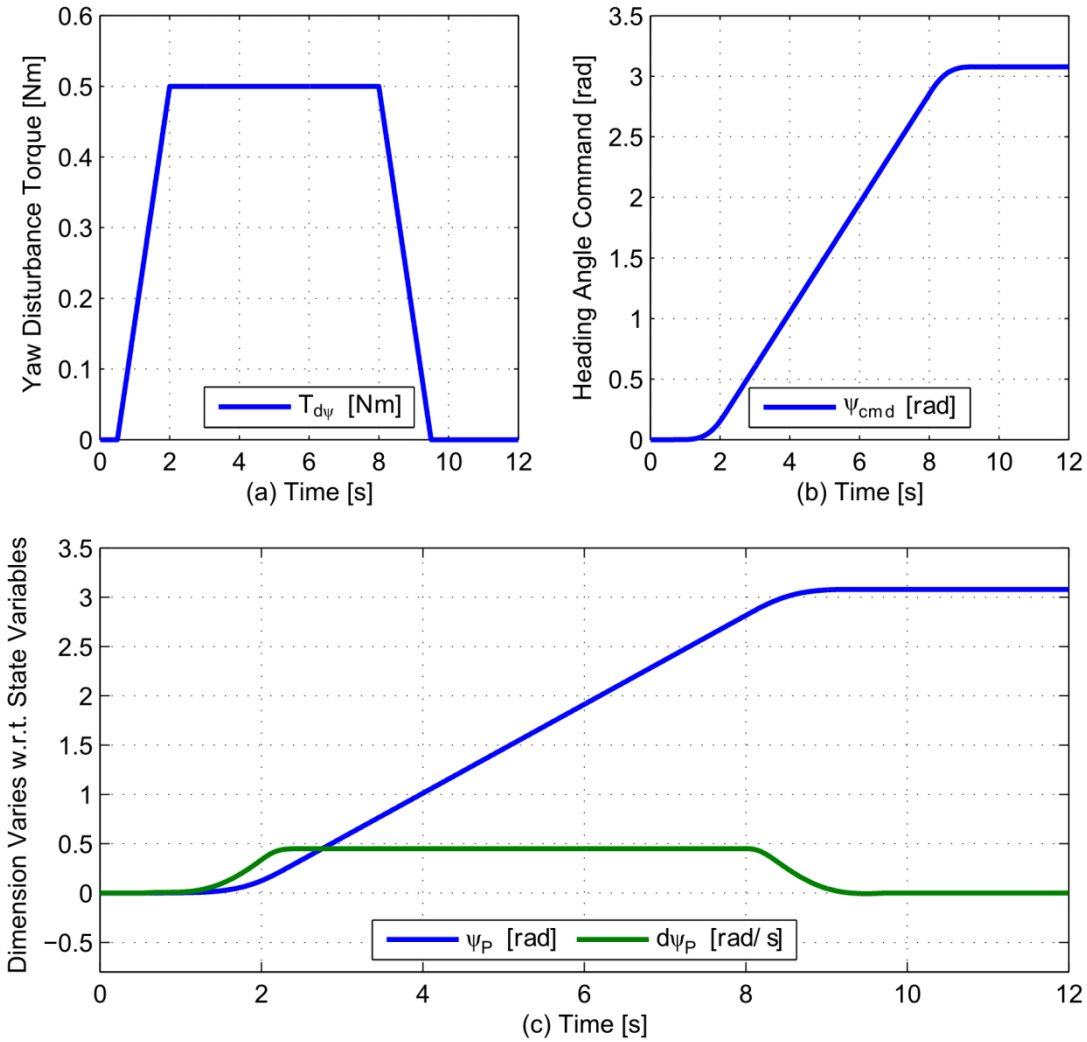

Figure 5-11. Simulation of (a) 0.5 Nm ramp disturbance torque acting at the handlebar about the yaw axis as well as the resulting (b) heading angle controller response and (c) system response with decoupled LQR and heading angle control.

In response to the yaw disturbance torque waveform depicted in Fig. 5-11a, the heading angle controller generates a heading angle command curve that is smooth and steady, as shown in Fig. 5-11b. Based on observations of Fig. 5-11c, it can be concluded that the heading angle controller has been successful in regulating the heading angle of the two-wheeled system with smooth and minimum jerk control. As a note, since there were no disturbance forces acting about the vehicle's pitch axis, the linear displacement,

linear velocity, pitch angle, and pitch velocity state variables remained unchanged and were thus omitted from Fig. 5-11c.

5.4. Overall Control Strategy Performance

Simulation results demonstrated that the decoupled LQR control scheme is capable of satisfactorily stabilizing the two-wheeled system and rejecting external disturbance forces. Furthermore, the ability of the state variable controllers to regulate the linear displacement and heading angle of the system with smooth and minimum jerk control was also simulated and verified. Simulation results also revealed the ability of the pitch controller to reduce the vehicle's linear displacement due to external disturbance forces and improve disturbance rejection. With the development and evaluation of a control strategy complete, the next step is to identify the electrical hardware required to construct and deploy a functional prototype of the two-wheeled system.

CHAPTER 6

HARDWARE ARCHITECTURE AND CIRCUIT DESIGN

This chapter discusses hardware selection and architecture as well as the design process of the two custom printed circuit boards (PCB) used to power all on-board electronics and implement the software designed to control the two-wheeled system. First, all hardware used for this project will be identified followed by a diagram illustrating hardware architecture and bus interface. Next, the circuit schematics for both the main and power management boards will be presented along with an in-depth description of the circuit components driving the hardware. Lastly, a detailed list of all electronic components and hardware utilized will be composed into three separate tables.

6.1. Hardware Identification and Architecture

In this section, all hardware used for computation, actuation, measurement, communication, data logging, signal conditioning, and power management will be identified. The overall hardware diagram summarizing hardware architecture and bus interface is presented in Fig. 6-1.

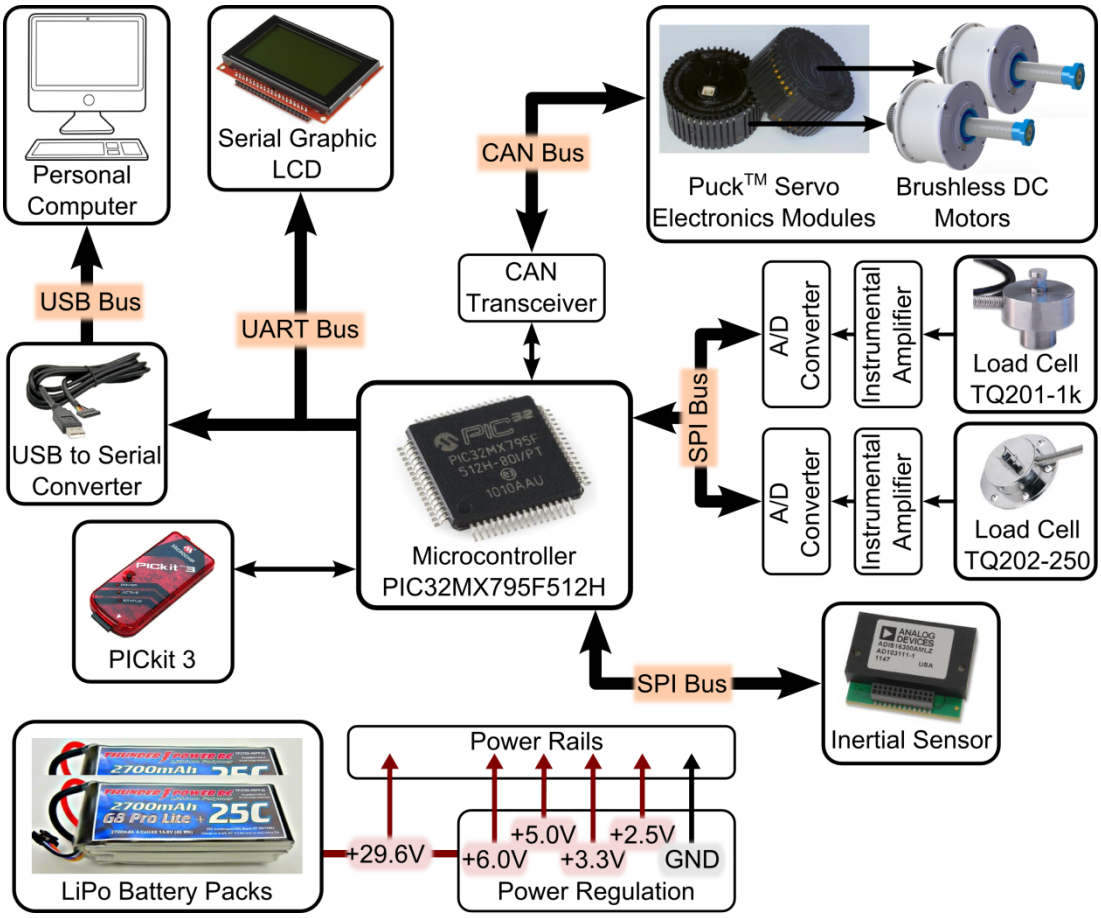

Figure 6-1. Hardware architecture and bus interface.

6.1.1. Microcontroller

During the search for a qualified microcontroller capable of interacting with all on-board hardware and performing high speed data processing necessary to control the two-wheeled system, a selection criteria for the on-board computing was formulated. To minimize cost and power requirements of the processor, a microcontroller-based platform was selected. Since the control algorithm would require mathematical operations on floating point numbers which are known to be computationally expensive, it was desired that the microcontroller should be capable of operating at clock speeds upwards of 80 MHz. A 32-bit computer architecture was demanded for increased computation speed of 32-bit floating point values as well as easier handling and processing of sensor data.

In light of hardware communication requirements, the microcontroller would need at least two SPI modules, one UART module, one CAN module, and one PWM module. Lastly, a few analog and digital pins would have to be reserved for the battery monitor circuit, LEDs, tactile switches, debugging, and future unplanned hardware needs. These hardware specification requirements led to the selection of a PIC32MX795F512H microcontroller shown in Fig. 6-2.

The PIC32MX795F512H microcontroller has a 32-bit architecture, maximum clock speed of 80 MHz, flash memory of 512 KB, 128 KB of RAM, and all peripheral modules needed to drive the on-board hardware. The microcontroller operates on input voltages between +2.3V and +3.6V, comes in a 64-pin thin quad flat package (TQFP), and supports 53 I/O pins.

Figure 6-2. 32-bit PIC microcontroller (PIC32MX795F512H).

6.1.2. Brushless DC Motors

The actuators used to balance the platform are two custom brushless DC motors (HT03002-E01) from Allied Motion Technologies Inc., as shown in Fig. 6-3. The torque and speed requirements for the motors were selected based on motor configuration and simulation results. Since the motors were going to be designed as direct drive systems, it was necessary to select motors capable of generating torque and speed outputs high enough to satisfy simulation results. With the motors arranged in a direct drive

configuration, the drive system will consist of much less moving parts and will not need to rely on any transmission components to transfer power or provide a mechanical advantage. As a result, the direct drive mechanism will experience increased mechanical efficiency, reduced noise, increased life cycle, faster and more precise angular position sensing, and reduced drive stiffness. In exchange for these advantages, the selected motors will end up being heavier and larger than the motors with gearboxes alternative due to the increase in winding that is necessary to generate higher torque outputs. The specifications for the selected motors are listed in Table 6-1.

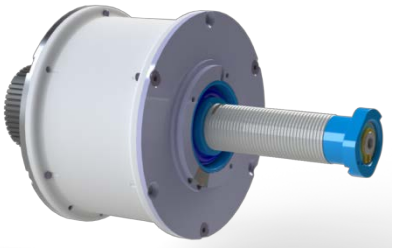

Figure 6-3. Brushless DC motor (HT03002-E01) from Allied Motion Technologies.

Table 6-1. Brushless DC motor specifications (HT03002-E01).

Variable	Description	Value	Unit
V_P	Design voltage	48	V
T_R	Maximum rated torque	21.765	$N \cdot m$
T_C	Maximum continuous stall torque	1.490	$N \cdot m$
P_C	Continuous power output	13.884	W
T_P	Peak torque, ±25%	6.310	$N \cdot m$
I_P	Peak current, ±15%	13.822	A
S_{NL}	No load speed	957.015	RPM
R_m	Terminal resistance	3.473	Ω
L_m	Terminal inductance	4.068	mH
K_m	Motor constant	0.245	$N \cdot m / \sqrt{W}$
K_t	Torque constant, ±10%	0.457	$N \cdot m / A$
K_b	Back EMF constant	0.457	$V / rad / s$
R_a	Armature resistance	3.473	Ω
W_t	Motor weight housed	1.422	kg

6.1.3. Puck™ Servo Electronics Module

Appended to the back of each brushless DC motor is a servo electronics module trademarked as Puck (B3880) by Barrett Technology Inc., shown in Fig. 6-4. Among numerous other features, each Puck consists of an electronic servo motor controller, 32-bit digital signal processor (DSP), magnetic encoder, power amplifier, temperature sensor, and precision current sensing. These modules operate on a power supply range of +24V to +90V, communicate using CAN protocol, and can drive currents of up to 8 amps.

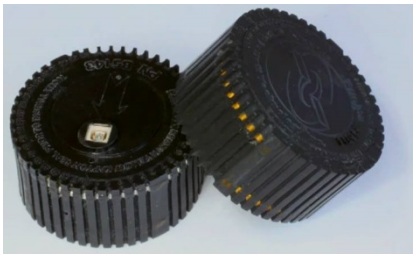

Figure 6-4. Puck™ (B3880) servo electronics module from Barrett Technology.

6.1.4. High-speed CAN Transceiver

The CAN module in the 32-bit PIC microcontroller is compliant with the CAN 2.0B standards. For compatibility with the CAN protocol, each node in the CAN bus network requires an external transceiver to level shift the digital signals generated by the microcontroller's CAN module to output voltages suitable for transmission over the bus cabling. To satisfy this compatibility requirement, a high-speed CAN transceiver (MCP2551) was used.

The MCP2551 is suitable for 12V and 24V systems, supports data transfer rates of up to 1 Mb/s, and implements ISO-11898-2 physical layer specification. The transceiver requires +4.5V to +5.5V input voltages for operation and contains a thermal shutdown

circuit for protection against excessive current loading. The device also provides protection against short-circuit conditions and high-voltage transients that can occur on the CAN bus.

6.1.5. Inertial Measurement Unit

Measurements of the two-wheeled vehicle's angular displacement and velocity about its pitch axis was accomplished using the four degrees of freedom inertial sensor (ADIS16300) depicted in Fig. 6-5. This is the same high quality sensor that has been responsible for estimating the pitch angles and velocities of uBot-5 and uBot-6 [53], [54]. The inertial sensor has a maximum sampling rate of 819.2 samples per second, passband bandwidth of 330 Hz, 12-bit temperature sensor, 14-bit digital gyroscope, and 14-bit tri-axis accelerometer. The device operates on a power supply range of +4.75V to +5.25V, communicates using SPI protocol, and also includes a programmable Bartlett window finite impulse response (FIR) filter for additional noise reduction on all output data registers.

Figure 6-5. Four degrees of freedom inertial sensor (ADIS16300).

6.1.6. Graphic Display

For the purposes of debugging and displaying real-time information about the two-wheeled system and its state variables, the 128x64 pixel serial graphic LCD (LCD-

09351) shown in Fig. 6-6 was used. This particular display comes with a serial graphic LCD backpack which provides the user easier control of the display using SPI communication. The display operates on a power supply range of +6V to +7V, consumes a maximum of 220 mA, and can be configured to operate at speeds of up to 115,200 bps.

Figure 6-6. Serial graphic LCD 128x64 (LCD-09351).

6.1.7. USB to Serial Converter

In order to allow MATLAB to collect and log real-time data about the two-wheeled system, a cable interface between the on-board microcontroller and an off-board computer had to be established. The task of transferring UART data bytes carrying system information to the USB port of an off-board computer was accomplished using a TTL-232R-3V3 cable, shown in Fig. 6-7. The TTL-232R-3V3 is a TTL to USB serial converter cable that contains embedded hardware designed to level shift digital signals received from the microcontroller's UART module to signals suitable for transmission over the USB port. The cable supports all UART configuration settings and data transfer rates from 300 bps to 3 Mbps.

Figure 6-7. TTL to USB serial converter cable (TTL-232R-3V3).

6.1.8. Reaction Torque Load Cells

To better estimate the user's intentions while interacting with the two-wheeled platform, two reaction torque load cells are used. One load cell will be used to collect measurements of disturbance torque acting about the pitch axis of the pendulum while the other will measure disturbance torque acting about the yaw axis of the handlebar.

The TQ201-1k load cell, shown Fig. 6-8, will be responsible for measuring disturbance torque acting about the pitch axis of the pendulum. This load cell has a rated torque capacity of 1000 in-lb and calibrated sensitivity of 2.844 mV/V. The measured input and output resistances were measured to be 376.0 Ω and 350.9 Ω, respectively.

Figure 6-8. Reaction torque load cell, 1000 in-lb (TQ201-1k).

The TQ202-250 load cell, shown in Fig. 6-9, will be responsible for measuring disturbance torque acting about the yaw axis of the handlebar. This load cell has a rated torque capacity of 250 in-lb and calibrated sensitivity of 2.238 mV/V. The measured input and output resistances were measured to be 374.4 Ω and 350.7 Ω, respectively.

Figure 6-9. Reaction torque load cell, 250 in-lb (TQ202-250).

6.1.9. Signal Amplifiers and A/D Converters

Each load cell will be supplied with a +5.0V excitation voltage. If the load cells are loaded to their maximum rated torque, the TQ201-1k load cell will output 14.22 mV while the TQ202-250 load cell will output 11.19 mV. Since it is unlikely that the load torque experienced by each load cell will surpass 50% the maximum rated torque, signal amplification is required to allow higher digital resolution during analog-to-digital conversion.

The output of each load cell is amplified by a factor of 100 using a high performance, rail-to-rail output instrumentation amplifier (AD8220). After signal amplification and buffering, a six channel delta sigma A/D converter (MCP3903) is utilized for signal conditioning and analog-to-digital conversion. At the front-end of each channel in the MCP3903 reside a programmable gain amplifier (PGA) which allows the differential voltage input to be further amplified. The MCP3903 is capable of outputting ADC data with 24-bit resolution and communicating to external hardware using SPI protocol at speeds of up to 10 MHz.

6.1.10. Power Supply

In order to allow the two-wheeled system to operate without being tethered to an external power supply, a lightweight and compact battery pack was procured. Among the various types of batteries available, the lithium polymer (LiPo) battery cell was discovered to be among the most suitable battery packs capable of fulfilling the two-wheeled system's power requirements. One of the main advantages about using LiPo batteries is their ability store higher energy per mass and energy per volume compared to other battery types, as shown in Fig. 6-10.

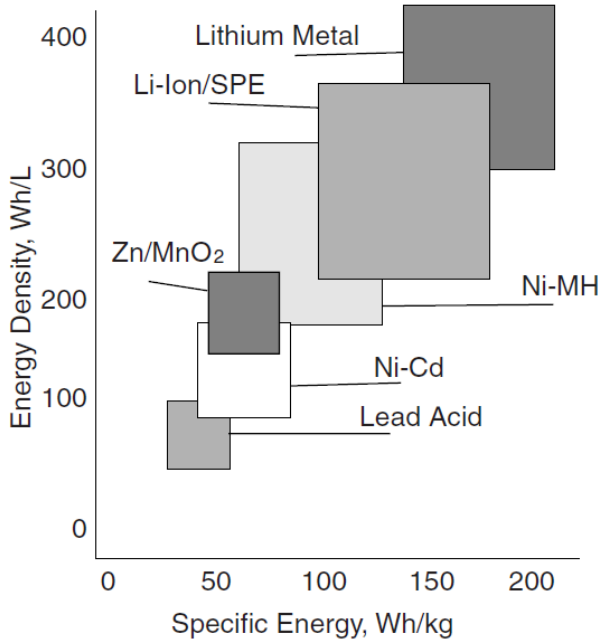

Figure 6-10. Comparison of the energy storage capability of various rechargeable batteries [55].

In addition to higher energy storage capability and light weight, another great advantage about using LiPo battery cells is their ability to discharge energy at higher rates. Therefore, in order to meet the power requirements of the two-wheeled system, two 2700 mAh LiPo battery packs (Thunder Power, TP2700-4SPP25) were purchased, shown in Fig. 6-11. The TP2700-4SPP25 is a 4-cell battery pack capable of supplying a nominal

voltage of 14.8V at a maximum continuous discharge rate of 25C or 67.5 A. Since the motor drivers require at least 24V to drive the motors, the two battery packs are connected in series for a combined nominal voltage of 29.6V and energy capacity of 2700 mAh.

Figure 6-11. 2700 mAh 4-cell/14.8V/25C LiPo battery pack (Thunder Power, TP2700-4SPP25).

6.2. Circuit Design and PCB Layout

To balance the two-wheeled prototype, two custom PCBs were designed using CadSoft EAGLE. One board designed to manage power supply, while the other designed to accommodate a 32-bit PIC microcontroller communicating on CAN, SPI, and UART buses to multiple electronic devices. Prior to finalizing the circuit schematics and producing the PCBs, extensive breadboard prototyping and circuit simulation was performed to minimize the potential for design errors in the circuit boards.

6.2.1. Breadboard Prototyping and Circuit Simulation

As part of circuit design process, all essential circuit components and communication protocols were tested before producing the printed circuit boards. During the breadboard prototyping process, two 16-bit PIC microcontrollers were wired and configured to communicate with each other through CAN, SPI, and UART protocols.

Load cell and signal amplification requirements were also prototyped on the breadboard. Circuit simulation of all components in the power management board were performed in LTspice IV. Simulation results confirmed and reassured that the circuits for the switching regulator, linear regulators, and battery monitor were properly designed.

6.2.2. Main Board

Once all the essential circuit components had been tested and simulated, the circuit components were combined and captured in CadSoft EAGLE. Due to size constraints, the circuit components were distributed into two separate circuit boards: a main board and a power management board. The complete circuit schematics and PCB layout for the main board can be found in Appendix C.

The first schematics sheet of the main board contains the PIC32MX795F512H microcontroller along with all electrical components required for its proper function. The connection headers leading to the power supply, LCD, PICkit 3, ADIS16300, and Pucks are also found in this schematics sheet. Additionally, the circuits for the CAN transceiver, LEDs, +2.5V voltage reference, and tactile switches are also included.

The second schematics sheet of the main board contains the circuit components used to perform signal processing on the output signals received from the load cells and force-sensing resistors (FSR). On the upper half of the schematics sheet, the output signal from each load cell is amplified by a factor of 100 at the instrumental amplifier (AD8220). In order to allow manual voltage adjustments to load cell output, a tuning circuit comprised of a voltage buffer and potentiometer has been implemented directly at the output signal line from each load cell. Once the output signal has passed the

instrumental amplifier, the signal enters the MCP3903 where A/D conversion and signal processing is performed. The MCP3903 will then continue to process analog signals received from the load cells until the microcontroller sends a data request.

The lower half of the second schematics sheet is also designed to perform A/D conversion and signal processing using a MCP3903 but instead of load cells, the incoming analog signals are from FSRs. The purpose of the FSRs is to give the microcontroller more information about how the user is interacting with the handlebars, i.e. where the user is grabbing the handlebar and roughly how much force is being applied. However, due to time constraints and order of priorities during development of the two-wheeled system, it was not possible to implement the FSRs on the prototype.

6.2.3. Power Management Board

Since the circuit components responsible for regulating, distributing, and monitoring the power supply could not fit inside the main board, a separate board was designed. The power management board is capable of regulating input voltages ranging from +10V to +40V down to +6V, +5V, and +3.3V. The board also contains a battery monitor circuit designed to output around +3.3V when the battery is in full charge and roughly +0V when the battery is nearing 20% of its energy capacity where the risk of permanent battery damage is imminent. The complete circuit schematics and PCB layout for the power management board can be found in Appendix D.

In light of the ability of switching regulators to step down high input voltages efficiently and with minimal power loss, the task of reducing the input battery voltage from +24-33.6V down to +6V is accomplished using a step-down switching regulator

(LT1076CT). The remaining two output voltages required from the power management board are +5V and +3.3V. However, now that the input voltage is +6V, it becomes more cost effective and efficient to use linear regulators. Using linear regulators will also help reduce ripple and reject noise on the input voltage. To regulate the +6V input voltage down to +5V, a low dropout regulator (LT1529CT-5) was used. Similarly, another low dropout regulator (LT1529CT-3.3) was used to drop the +6V input voltage down to +3.3V.

The battery monitor circuit in Appendix C was designed to monitor a 6-cell LiPo battery pack. However, to reconfigure the circuit to monitor an 8-cell LiPo battery pack, the +18V Zener diode simply needs to be replaced by a +24V Zener diode. It should be noted that the unity gain differential amplifier in the battery monitor circuit was routed incorrectly. Instead of receiving negative feedback, the OPA2137P op-amp was configured to have positive feedback. Unfortunately, this error wasn't discovered until the manufactured PCB was probed and tested. Thus, the battery monitor circuit is currently unusable. This is an error that will have to be fixed in future work.

An important step in managing a power supply is overcurrent protection of on-board hardware and electronics. To protect the circuit boards and hardware, a 2.5 A fast response fuse was placed between the power supply and the power management board. The criteria used to select the 2.5 A fuse was the need to ensure that the maximum possible current that all hardware could simultaneously draw was equal to 75% of the fuse' current rating. Meanwhile, to protect the motors and their servo electronic modules, a 40 A slow response fuse was placed between the power supply and the Pucks. The

criteria used to select the 40 A fuse was the same as the criteria used in the 2.5 A fuse selection.

6.3. Bill of Materials – Electronic Components

Refer to Appendix E for a list of all electronic components used in the main and power management boards as well as the hardware and electrical accessories used in the two-wheeled system.

CHAPTER 7

FIRMWARE AND SOFTWARE DEVELOPMENT

Chapter 5 defined the control system responsible for controlling the two-wheeled system while the previous chapter identified all hardware and electronic components that are necessary to realize the control system. Now that all control algorithms and hardware have been identified, the next step is to develop the firmware and software that will drive the hardware and actualize the control system. The next step will be taken in this chapter.

In this chapter, firmware and software development will be discussed. The discussion will start with a brief introduction to the software application and hardware used to develop, compile, and load the program into the microcontroller. Section 7.2 will provide an overview of software architecture followed by identification and configuration of the communication protocols used to drive the hardware. Meanwhile, Section 7.4 will focus on the development of hardware drivers and identification of hardware configuration settings. In Section 7.5, the steps taken and firmware written to implement data logging will be discussed. Section 7.6 will examine the implementation of the control algorithms. To end the chapter, the last section will identify the basic procedures and set of tools employed throughout software development to diagnose and troubleshoot software errors.

7.1. Introduction

All firmware and software were written in C programming language and compiled using Microchip's Integrated Development Environment (IDE) software application known as MPLAB IDE. Debugging and programming of the PIC

microcontroller was performed using Microchip's PICkit 3 In-Circuit Debugger/Programmer in conjunction with MPLAB IDE. The complete software developed to control the two-wheeled system is provided in Appendix F. To facilitate the access and retrieval of task specific functions, the software has been organized into 12 different files. The contents these files will be highlighted and summarized in subsequent sections.

7.2. Software Architecture

The software is configured to execute the control algorithm at a fixed sampling rate of 100 Hz. The software is capable of running faster than 100 Hz but at higher sampling rates, low velocity estimations become increasingly erroneous. As an example, consider the Puck's encoder resolution of 4096 counts/revolution with a software sampling rate of 250 Hz. The minimum rotation that can be detected is 0.0002441 revolutions, which gives a velocity resolution of 3.662 rpm or 21.97 deg/s $\left(\frac{0.0002441}{0.004} \frac{\text{rev}}{s} \times \frac{360}{1} \frac{\text{deg}}{\text{rev}} \right)$ when sampled at 250 Hz or every 0.004 s. At moderate to high speeds, this velocity resolution may be satisfactory, e.g. 1% error at 366.2 rpm. However, at angular velocities below 3.662 rpm, the velocity estimate would erroneously be zero much of the time.

With a sampling rate of 100 Hz, the angular velocity resolution for the shaft of the motors becomes 1.465 rpm or 8.79 deg/s. Although this resolution may still be slightly large for low velocity estimates, signal conditioning was performed in every sensor output in order to achieve more accurate position and velocity estimates. The primary method for conditioning sensor outputs was digital filtering. The digital filters that were

applied to sensor outputs will be identified in this chapter and further elaborated in Chapter 9. Meanwhile, to ensure that the microcontroller would be able to execute the control algorithm fast enough to satisfy the desired sampling rate, the microcontroller's clock speed was configured to operate at 80 MHz for the entire programming phase.

The preliminary steps in the software's initialization routine is to select the microcontroller's internal oscillator, configure clock rate, and define the states of all I/O pins that are going to be used by the hardware and other electronic components. Once these steps are complete, the communication modules needed to interact with peripheral hardware are configured and initialized. With the communication between the microcontroller and hardware established, custom configuration settings for each hardware are transmitted and iteration of control algorithm can initiate. Within each iteration of software execution, the control algorithm fetches all necessary sensor data, processes it, filters it, and uses it to calculate torque commands that will be fed to each motor. For a visual description of software architecture, see Fig. 7-1.

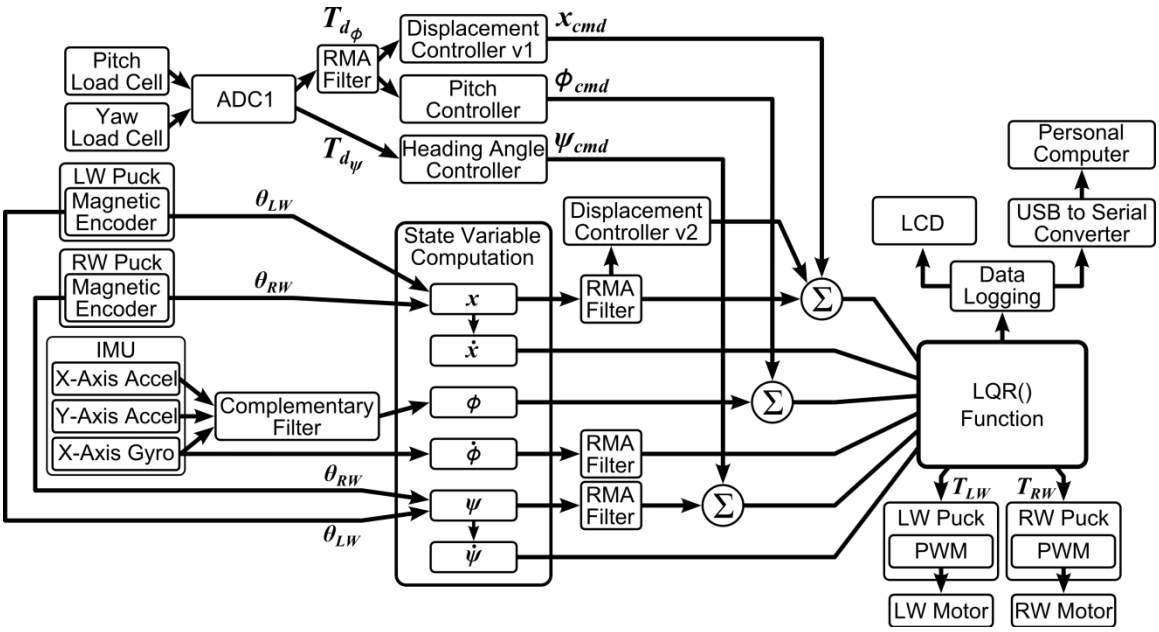

Figure 7-1. Software architecture.

7.3. Serial Data Communication Protocols

In order for the microcontroller to communicate with the on-board hardware, the microcontroller needs to transmit and receive data according to the communication protocol and timing specifications used by the hardware. Since the PIC microcontroller selected for this project already contains integrated CAN, SPI, and UART modules, establishing the necessary communication protocols only requires a few lines of code. Once the configuration bits are set and the communication modules are enabled, the last step is to develop the functions that will transmit and receive data according to requirements of each communication protocol. The following subsections will define the configuration settings used for each communication module and highlight the functions driving the module.

7.3.1. Controller Area Network (CAN)

The Puck^{TM} servo electronics modules controlling the brushless DC motors uses CAN protocol to communicate with the microcontroller. Furthermore, in order for communication between the two devices to function properly, the microcontroller's CAN module must be configured to match the Puck's bit timing and message format requirements. In summary, the CAN module was configured to transfer data at 1 Mbps baud rate with 8 time quanta per bit, sampling point at 75% of nominal bit time, synchronization jump width of 1 time quanta, and 11-bit message identifier. Meanwhile, the only functions driving the module are *readCANmsg()* and *sendCANmsg()*. For more information about these functions and the configuration settings for the CAN module, refer to CAN.c file in Appendix F.

7.3.2. Serial Peripheral Interface (SPI)

Both the ADIS16300 inertial measurement unit (IMU) and the MCP3903 analog-to-digital converters (ADCs) rely on the SPI protocol for communication. However, since SPI settings for these hardware units are not compatible, two separate SPI modules on the microcontroller were used. The SPI module communicating with the IMU was configured to transfer data at 1 MHz with 16-bit wide data, active-low clock polarity, data capture occurring during clock's falling edge, and input data sampled at the end of data output time. Meanwhile, the SPI module communicating with the ADCs uses the exact same settings except that data transfers must be 8-bit wide instead of 16-bit wide. The functions responsible for driving both SPI modules are *readSPImsg()* and *sendSPImsg()*. For more information about these functions and the configuration settings for the SPI modules, refer to SPI.c file in Appendix F.

7.3.3. Universal Asynchronous Receiver/Transmitter (UART)

The UART protocol is used to communicate with the serial graphic LCD as well as the Serial to USB Converter cable (TTL-232R-3V3). Since the LCD and converter cable have compatible UART settings, only one UART module was required. The converter cable can support any UART setting but the LCD requires that the UART frame contains 8 data bits, 1 stop bit, and no parity. In order to allow the LCD and converter cable to share communication lines, the UART module was configured to transfer data at 115,200 bps with 8 data bits, 1 stop bit, and no parity. Meanwhile, the only functions needed to drive the UART module are *readUARTmsg()* and

sendUARTbyte(). For more information about these functions and the configuration settings for the UART module, refer to UART.c file in Appendix F.

7.4. Hardware Driver Development and Configuration

Configuring and enabling the communication modules in the microcontroller is the first step to establishing communication with the on-board hardware. The next step is to develop the functions that will configure and drive the hardware while satisfying the timing specifications of each hardware. In the following subsections, the configuration settings used for each hardware will be identified followed by a brief overview of key functions developed to configure and drive each hardware.

7.4.1. Puck™ Servo Electronics Module

The angular position and speed of each brushless DC motor is controlled by a Puck™ servo electronics module. These compact and power efficient high performance servomotor controllers greatly reduce software complexity. However, in order to utilize these modules without miscommunication between the software and hardware, it is necessary to develop functions that will configure and drive the hardware according to CAN message format and timing specifications. In summary, each Puck was configured to transfer data at 9600 bps, operate in torque mode, and limit the maximum applied motor torque to 2400 mA. An overview of key functions developed to configure and drive each Puck is presented in Table 7-1. For more information about these functions and the configuration settings for the Pucks, refer to Puck.c file in Appendix F.

Table 7-1. Overview of key functions developed to configure and drive each Puck.

Function Name	Description
config_pucks()	Configure Puck status, baud rate, mode, and torque limit.
setPropertyPuck()	Set specified Puck property to transmitted data value.
getPropertyPuck()	Query Puck property and process returned data.
parseCANmsgPuck()	Process CAN message received from Puck.
setTorquePuck()	Send torque command to left or right motor.
getPositionPuck()	Get encoder count and calculate motor position.
encoderZeroCrossing()	Detect and correct for zero crossing.
convertUnits_Puck_to_SI()	Convert data from Puck units to SI units.
convertUnits_SI_to_Puck()	Convert data from SI units to Puck units.
getConversionFactorsPuck()	Query and store returned Puck conversion factors.
resetPucks()	Reset and reconfigure Pucks.

7.4.2. Inertial Measurement Unit (IMU)

In order for the control algorithm to robustly balance the two-wheeled system, it is very important that the IMU (ADIS16300) generates accurate low noise angular displacement and velocity measurements. It is unavoidable that the IMU measurements will become slightly distorted and noisy due to platform vibrations induced by motor output torque. Fortunately, the IMU contains a 350 Hz signal conditioning circuit and a programmable Bartlett window FIR filter that can provide additional noise reduction on all output measurements. However, simply increasing the filtering capacity of the Bartlett filter can result in noticeable signal latency or lag. Thus, it is necessary to configure the IMU's Bartlett filter to a setting where the compromise between noise and signal latency is acceptable.

After testing various different settings, the most favorable Bartlett setting was found to be the 9-tap filter setting with a gyroscope sensitivity of ±150 deg/s. For ideal operation, the IMU was configured to an internal sampling rate of 819.2 samples per second, gyroscope sensitivity of ±150 deg/s, and 9-tap Bartlett filter. An overview of key

functions developed to configure and drive the IMU, according to SPI message and timing requirements, can be found in Table 7-2. For more information about these functions and the configuration settings for the IMU, refer to IMU.c file in Appendix F.

Table 7-2. Overview of key functions used to configure and drive the IMU.

Function Name	Description
config_IMU()	Configure IMU mode, gyroscope, and Bartlett filter.
setPropertyIMU()	Set specified IMU property to transmitted data value.
getPropertyIMU()	Query IMU property and process returned data.
parseSPImsgIMU()	Process SPI message received from IMU.
convertUnits_IMU_to_SI()	Convert data from IMU units to SI units.
get_IMUgyro()	Get IMU's x-axis gyroscope output.
get_IMUxaccl()	Get IMU's x-axis accelerometer output.
get_IMUyaccl()	Get IMU's y-axis accelerometer output.

7.4.3. Serial Graphic LCD

The complexity of the functions required to drive the individual pixels on the LCD is greatly reduced by the integrated embedded hardware. Therefore, instead of having to develop functions to drive individual pixels, the functions only need to transmit a few bytes in ASCII format through the UART module in order for a single or set of pixels on the LCD to update. The only adjustable setting is the hardware's baud rate which was kept at 115,200 bps. An overview of key functions developed to configure and drive the LCD is presented in Table 7-3. For more information about these functions and the configuration settings for the LCD, refer to LCD.c file in Appendix F.

Table 7-3. Overview of key functions used to configure and drive the LCD.

Function Name	Description
config_LCD()	Configure LCD mode and backlight intensity.
printVariable()	Print variable name, floating-point number, and units.
printString()	Print string.
printInteger()	Print integer value.
printFloat()	Print floating point value.
terminateLine()	Print blank spaces until LCD jumps to next line.
clearScreen()	Clears the screen of a written pixels.
setBacklightIntesity()	Changes the backlight intensity.
setBaudRate()	Changes the baud rate between six predefined settings.

7.4.4. Analog-to-Digital Converter (ADC)

In order to satisfy the communication requirements for the MCP3903 device, the first byte transmitted by the driving function must always be the control byte. The control byte is used to identify which ADC will receive the message, the target register address, and whether the specified target register address is going to be read or written. If the control byte has the write bit active then the 24-bit wide data value that is going to be stored at the target register address must follow the control byte.

Once the sequence of SPI messages comply with communication and timing requirements, it will become possible to configure ADC's internal registers and retrieve processed load cell data. Since further signal amplification was not necessary, the PGA gain of all analog input channels were configured 1. To allow the ADC to produce output data with 24-bit resolution, the delta sigma A/D conversion oversampling ratio was configured to 256 and the ADC channel output data word width was set to 24-bit mode. An overview of key functions developed to configure and drive the ADC is presented in Table 7-4. For more information about these functions and the configuration settings for the ADC, refer to ADC.c file in Appendix F.

Table 7-4. Overview of key functions used to configure and drive the ADC.

Function Name	Description
config_ADC()	Set ADC phase, gain, status, and configuration bits
init_MasterClockADC()	Initialize PWM to configure ADC clock rate.
setPropertyADC()	Set specified ADC property to transmitted data value.
getPropertyADC()	Query ADC property and process returned data.
parseSPImsgADC()	Process SPI message received from ADC.
getLoadCellData_pitch()	Get pitch load cell data, normalize and convert to SI units.
getLoadCellData_yaw()	Get yaw load cell data, normalize and convert to SI units.
getLoadCellOffset_pitch()	Collect N pitch load cell data and calculate average.
getLoadCellOffset_yaw()	Collect N yaw load cell data and calculate average.

7.5. Data Logging

The ability to measure and record real-time data about the two-wheeled system is an important tool during software development and system evaluation. Access to a live stream of system parameters via sensors can provide insight into the performance of the sensors, digital filters, control algorithm, and motor. Data logging was implemented by using the microcontroller's UART module to transmit a batch of comma-delimited data containing system parameters to the USB port of an external computer. However, as identified in the previous chapter, the serial data exiting the UART module must first pass through a Serial to USB converter cable for proper voltage level shifting.

Once the batch of comma-delimited data arrives at the external computer's USB port, the data is forwarded to MATLAB where it gets processed, stored, and plotted in real-time. For a copy of the MATLAB file developed to receive, process, plot, and store the batch of comma-delimited data arriving from the microcontroller, refer to Appendix G. On the other hand, the functions used to convert data into ASCII format, breakdown the floating-point numbers into a sequence of bytes, and drive the UART module are the same functions that were presented in Table 7-3 and stored inside the LCD.c file.

Meanwhile, the function developed to transfer the batch of comma-delimited data to MATLAB is *logLQRdata()* which can found in the LQR.c file.

7.6. Control Algorithms

Now that all software components necessary to drive the on-board hardware and sensors are complete, the control algorithms can be developed. As covered in Chapter 5, the behavior of the two-wheeled system is dictated by a decoupled LQR controller and three state variable controllers. The following subsections will focus on detailing the developmental process and identifying the software components that make up each control algorithm.

7.6.1. Decoupled LQR Controller

The decoupled LQR controller is responsible for balancing the platform quickly with minimal overshoot and steady-state error. The LQR algorithm developed in software, denoted by *LQR()*, accomplishes this task by calculating and updating motor torque several times per second. However, before motor torque can be calculated, the *LQR()* function first needs to calculate six state variables representing the vehicle's linear displacement, linear velocity, pitch angle, pitch velocity, heading angle, and heading velocity.

The *LQR()* function begins by requesting the cumulative angular position of each motor followed by the IMU's x-axis gyroscope, x-axis accelerometer, and y-axis accelerometer outputs. The cumulative angular position of the left and right motors are used to calculate the vehicle's linear displacement, linear velocity, heading angle, and

heading velocity. Meanwhile, the vehicle's pitch velocity is calculated using gyroscope measurements while the pitch angle is calculated using a complementary filter that combines the IMU's gyroscope and accelerometer measurements. In order to reduce sensor noise, a recursive moving average (RMA) filter is applied to the calculated linear displacement, pitch velocity, and heading angle of the vehicle.

If the state variable controllers are active then the next step in the algorithm will be to request load cell torque measurements and use these values to calculate new values for the vehicle's linear displacement, pitch angle, and heading angle. Once all six state variables have been calculated and updated by the state variable controllers, these variables are used to calculate new motor torque commands for both the left and right wheels. These motor torque commands are transmitted to the Pucks and, if desired, the state variables are forwarded to an external computer for data logging before the *LQR()* function terminates and the process repeats.

An overview of key functions developed to drive the LQR algorithm is presented in Table 7-5. For more information about these functions and the configuration settings for the LQR algorithm, refer to LQR.c file in Appendix F.

Table 7-5. Overview of key functions used to drive the LQR algorithm.

Function Name	Description
LQR()	Computes motor torques required to perform LQR control.
getStateVariable_x()	Computes the vehicle's linear displacement.
getStateVariable_dx()	Computes the vehicle's linear velocity.
getStateVariable_phi()	Computes the vehicle's pitch angle.
getStateVariable_psi()	Computes the vehicle's heading angle.
getStateVariable_dpsi()	Computes the vehicle's heading velocity.
RMAfilter()	Implements n-point recursive moving average filter.
DisplacementControl()	Computes linear displacement using pitch load cell output.
DisplacementControl_v2()	Computes linear displacement using encoder output.
PitchControl()	Computes pitch angle using pitch load cell output.
HeadingAngleControl()	Computes heading angle using yaw load cell output.
MinJerkTraj()	Computes a smooth trajectory path.
PIcontroller()	Implements proportional-integral (PI) control.
printLQRdataLCD()	Prints all state variables on LCD screen.
logLQRdata()	Delimits input values and forwards them to MATLAB.

7.6.2. State Variable Controllers

The software contains three state variable controllers that are designed to control the vehicle's linear displacement, pitch angle, and heading angle according to load cell measurements. These algorithms were incorporated into the *LQR()* function and are implemented in the same way that they were described in the Simulink state variable controller models from Chapter 5.

The three state variables controllers in the *LQR()* function are called *DisplacementControl()*, *PitchControl()*, and *HeadingAngleControl()*. Aside from a few parameters, these three functions operate very similarly. Each function contains multiple IF statements but only one will execute depending on which condition is satisfied by the input load cell measurement. In two of the IF statements, minimum jerk trajectory and PI control are implemented in order to minimize and smoothen jerking in the values

calculated by these functions. For more information about these functions, refer to LQR.c file in Appendix F.

7.7. Error Handling

At the current stage, the software does not contain any error handler programs designed to resolve run-time errors. This section will only be briefly covering the tools used to fix development errors such as syntax and logic errors. Syntax errors were easily fixed through proofreading but logic errors relied on the aid of several different tools to resolve. The primary tool for resolving logic errors was MPLAB IDE's debugger. Other tools used were oscilloscope, CAN/SPI bus analyzers, and data logging.

CHAPTER 8

PROTOTYPE DEVELOPMENT AND CONSTRUCTION

In this chapter, the development and construction process of a prototype for the TWIP robotic walker is detailed. The first section will discuss and present the solid models developed to assist in the physical construction and machining of accurately sized mechanical parts used in prototype assembly. In the subsequent section, the construction process and tools used during the manufacture of individual mechanical components will be identified. Finally, the last section will present a detailed list of all mechanical components used during prototype construction.

8.1. Solid Modeling

Prior to constructing a physical prototype of the two-wheeled system, a 3D solid model of the prototype assembly was created in SolidWorks. The prototype development process began from a two-wheeled chassis that was contributed to the TWIP robotic walker project. Using the two-wheeled chassis as the starting point, a mechanical handlebar with integrated sensors was designed and constructed to mount on top of the chassis. To finalize the prototype assembly, metal and plastic mounts for on-board electronics were fabricated.

8.1.1. Two-Wheeled Chassis Assembly

The two-wheeled chassis, presented in Fig. 8-1, was the starting solid model and structure for the TWIP prototype. This solid model along with its physical counterpart was contributed by Dr. Grupen from the Laboratory for Perceptual Robotics (LPR) at the

University of Massachusetts Amherst. The chassis was constructed entirely out of aluminum and designed with enough space to house all on-board electronics and hardware. The contributed chassis also included two brushless DC motors and two Pucks. This contribution decreased production costs and greatly expedited the prototype development process by eliminating the need to construct a two-wheeled chassis and select appropriate DC motors and motor electronics.

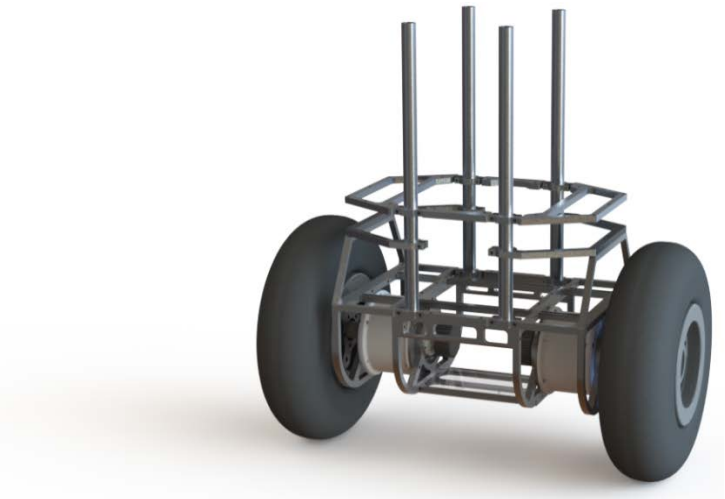

Figure 8-1. Two-wheeled chassis of the TWIP prototype.

8.1.2. Handlebar Assembly

The handlebar assembly, shown in Fig. 8-2, was designed and constructed by undergraduate student, Natalie Zucker, as part of her coursework requirements. The handlebar assembly was fabricated mostly from aluminum but also contains some parts made of steel and bronze. One load cell is fixated to the mounting plate while another load cell is housed inside the cylindrical shell between the two grip bars. The load cell fixtures and housing were designed to ensure that all forces acting on the handlebar about the pitch axis would be supported by the lower load cell while all forces acting about the

yaw axis would be supported by the upper load cell. For more information regarding the developmental and construction process of the handlebar assembly, refer to Zucker's thesis manuscript [56].

Figure 8-2. Handlebar assembly of the TWIP prototype.

8.1.3. Electronics Mounting Plates and Fixtures

In order to secure all on-board electronics and hardware, two aluminum mounting plates were designed. The first aluminum mounting plate, shown in Fig. 8-3, was designed to hold the printed circuit boards, fuse holders, and plastic 3D printed mounting fixtures. The plastic mounting fixtures were designed to properly secure the inertial measurement unit, LCD, and toggle switch to the aluminum plate. The second aluminum mounting plate, shown in Fig 8-4, was designed to hold the LiPo batteries. Both aluminum mounting plates also contain multiple holes for wire routing.

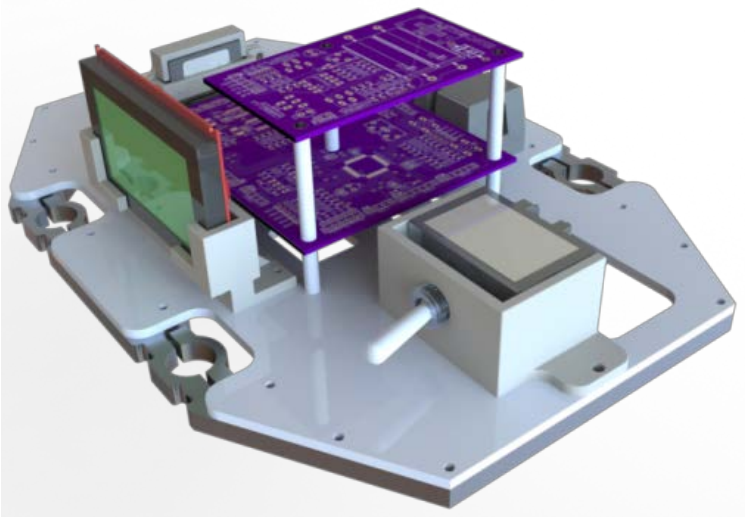

Figure 8-3. Aluminum and plastic mounting fixtures for on-board electronics.

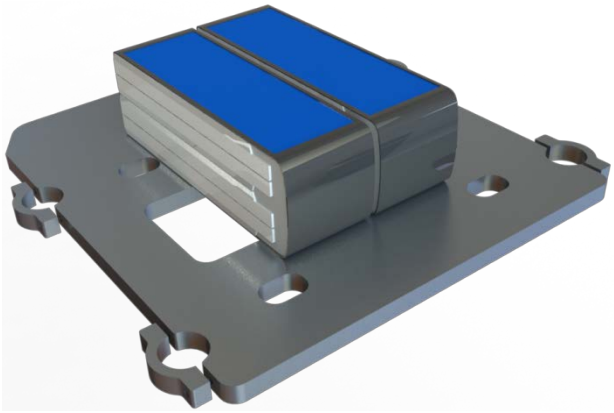

Figure 8-4. Aluminum mounting plate for LiPo batteries.

8.1.4. Prototype Solid Model

The complete prototype solid model erected from combining the two-wheeled chassis assembly, handlebar assembly, and electronics mounting plates and fixtures can be seen in Fig. 8-5.

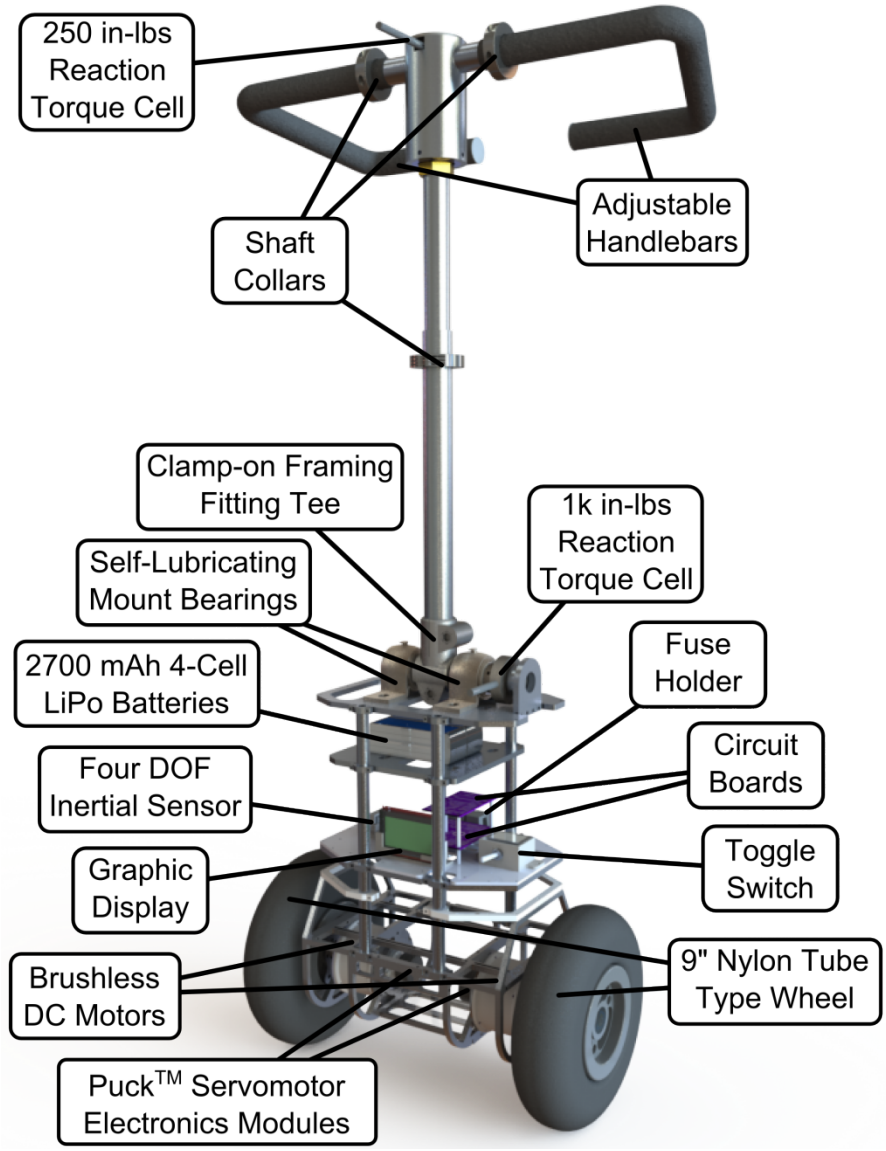

Figure 8-5. Complete prototype solid model of the two-wheeled system.

8.2. Prototype Construction and Assembly

Using the solid models created in the previous section, a physical prototype of the two-wheeled system was constructed. The construction process began with handlebar which was designed, constructed, and assembled by Zucker. The handlebar was mounted on top of the two-wheeled chassis and mechanically tested for design flaws. Some

mechanical flaws were discovered and fixed through component redesign, milling, welding, and lathing.

Following handlebar construction, the aluminum mounting plate for on-board electronics was water jet cut, sanded, and tapped. The plastic mounting fixtures for the IMU, LCD, and toggle switch were fabricated on a 3D printer and mounted on the electronics plate along with the populated main and power management printed circuit boards. The fuse holders and fuses protecting the electronics from overcurrent damage were also mounted on this electronics plate. Wires were crimped, housed, soldered, labeled, and properly insulated. A top view of the assembled electronics plate with all wires routed, secured, and connected is presented in Fig. 8-6.

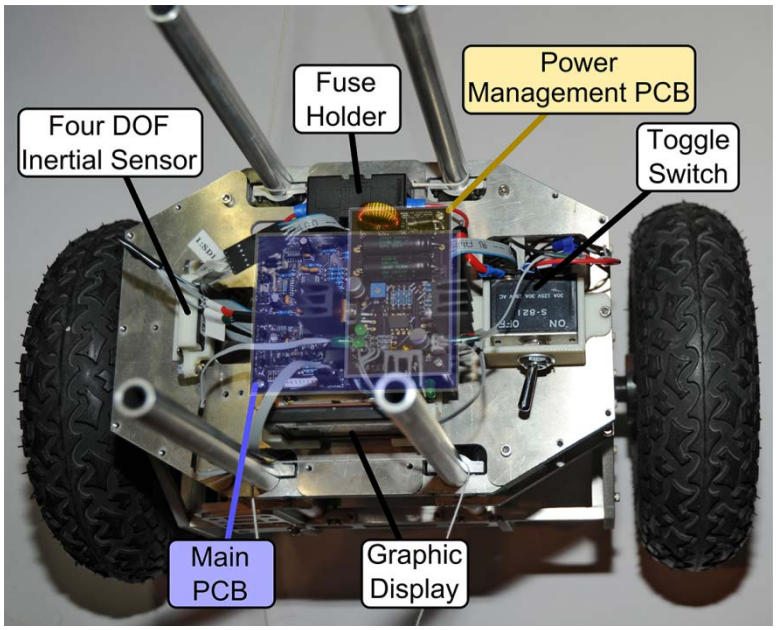

Figure 8-6. Top view of assembled electronics plate.

For a close up view of the two-wheeled base with all electronics assembled, see Fig. 8-7.

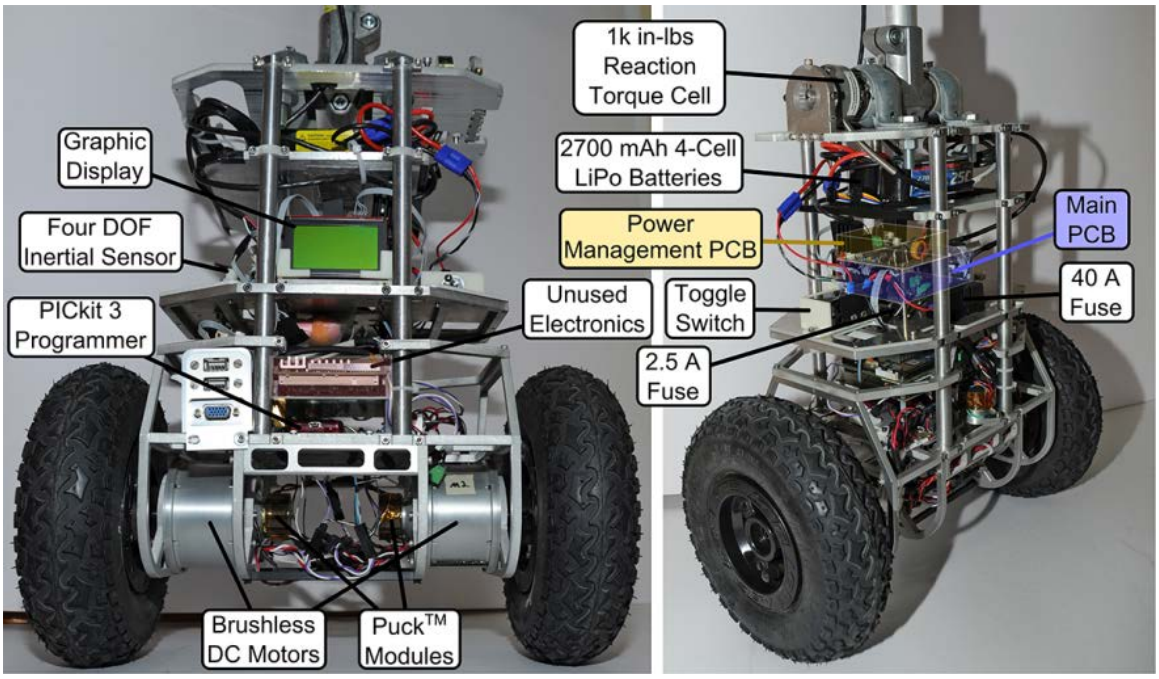

Figure 8-7. Close up view of the two-wheeled base with all electronics assembled.

The completed and assembled prototype of the two-wheeled system can be seen in Fig. 8-8.

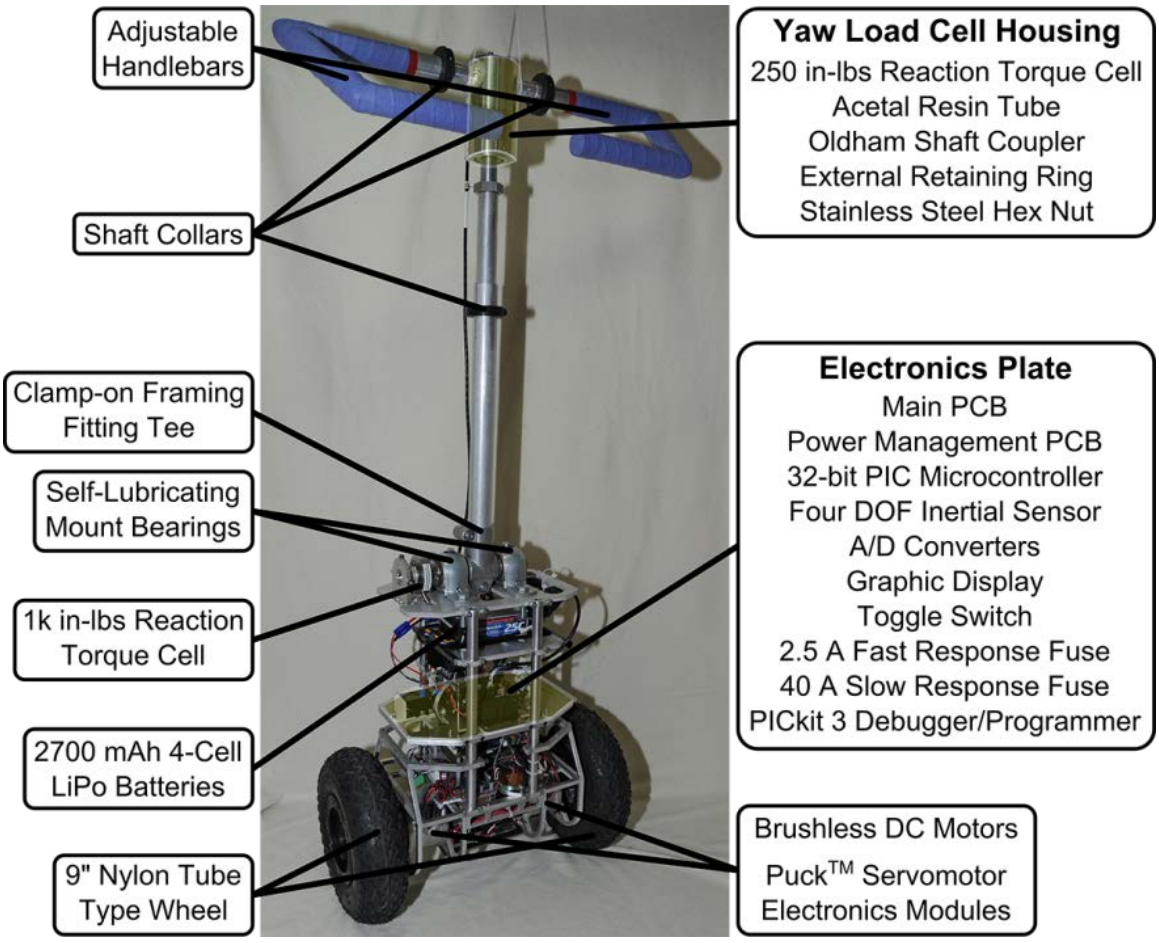

Figure 8-8. Completed prototype assembly of the two-wheeled system.

To provide a size perspective, an image with a user interacting with the two-wheeled platform can be seen in Fig. 8-9.

Figure 8-9. User interacting with the two-wheeled prototype.

8.3. Bill of Materials – Mechanical Components

Refer to Appendix H for a list of all mechanical components used in the construction of the prototype for the two-wheeled system.

CHAPTER 9

PLATFORM EVALUATION

With the construction of a prototype, the integration of on-board electronics, and the successful implementation of a control algorithm, the overall performance of the two-wheeled system can finally be evaluated. The first section of this chapter will discuss the calibration and tuning procedures performed on the digital filters, LQR gain matrices, and state variable controllers. Section 9.2 will investigate the ability of the LQR controller to stabilize the platform and reject external disturbance forces. Meanwhile, Section 9.3 will individually test the performance of state variable controllers in regulating the linear displacement and heading angle of the platform. To close the chapter, an evaluation of overall system performance will be presented.

9.1. Calibration and Tuning

Two of the most important elements affecting the performance of the two-wheeled system while balancing are the calibration and tuning of digital filters, LQR gain matrices, and state variable controllers. Fine-tuning these parameters is a critical step towards optimizing the robustness of the two-wheeled system. Mistuning any of these parameters can easily cause the platform to exhibit a low or high frequency oscillatory behavior capable of driving the system to instability. The following subsections will discuss and identify the procedures performed to adjust the behavior of the two-wheeled system and optimize its robustness.

9.1.1. Digital Filters Configuration and Tuning

Including the digital filter integrated into the IMU, there are a total of three filters that must be configured and tuned. The IMU's Bartlett window FIR filter holds the greatest influence over the performance of the platform. This filter is responsible for filtering noise in gyroscope and accelerometer measurements. If the Bartlett window filter is not properly configured and tuned then the control algorithm will be unable to stabilize the platform about its pitch axis. After testing and analyzing the platform's behavior under several different Bartlett window filter settings, the most favorable setting was discovered to be the 9-tap filter. In addition to significantly reducing noise, the 9-tap Bartlett window filter also ensures very low delay or lag in the IMU's filtered measurements.

Another important digital filter that can greatly affect the performance of the platform is the first-order complementary filter which is implemented in software. The complementary filter combines gyroscope and accelerometer measurements in order to provide a responsive, drift-free, and more accurate estimate of the vehicle's pitch angle. This filter can be tuned by adjusting the time constant, τ, which is used to define how much time the integrated gyroscope angle estimate is given precedence before the accelerometer average gains more priority.

Tuning the complimentary filter's time constant, τ, can be a straightforward process as long as a basic understanding of how the time constant affects pitch angle estimates is known. The time constant must be a value between 0 and 1. Setting the time constant, τ, less than 0.5 allows the pitch angle estimations to be more accurate when the IMU is not in motion by giving the accelerometer greater priority. Alternatively, setting τ greater than 0.5 allows the pitch angle estimations to be more accurate when the IMU is

in motion by giving the gyroscope measurements greater priority. Since the platform will be in constant motion, it is best to keep the time constant higher than 0.5. Upon testing and analyzing the vehicle's pitch angle estimates in response to various different time constant values, the most favorable setting was found to be $\tau = 0.8$.

The last of the three digital filters is the N-point recursive moving average (RMA) filter which is also implemented in software. Although the influence of the RMA filter over the platform's performance is not as great as the IMU's Bartlett window filter or complementary filter, proper tuning of the RMA filter is still required to optimize system performance. This filter is easily implemented by storing the last N-points of a specific parameter and calculating a new average every software execution. The number of points, N, is selected based on the presence of noise and filtering requirements.

The software contains four RMA filters that were each applied to the vehicle's calculated linear displacement, pitch velocity, heading angle, and pitch load cell torque. The RMA filters applied to the calculated linear displacement, heading angle, and pitch load cell torque are configured to conduct 5-point averages. On the other hand, the calculated pitch velocity uses a 3-point RMA filter since extra filtering was necessary in addition to the 9-tap Bartlett window filter applied IMU's gyroscope measurements.

9.1.2. LQR Gain Matrices Tuning

The performance of the LQR algorithm in stabilizing the two-wheeled system is determined by how well the LQR gain matrices or LQR gains are tuned. To facilitate the tuning process, the LQR gains were tuned based on a couple of requirements dictating the desired behavior of the LQR algorithm. The primary requirement imposed on the LQR

algorithm is to make the upright stability of the platform extremely resistant to external disturbances, which can be accomplished by configuring the LQR gain of the vehicle's pitch angle to be the highest gain. Meanwhile, the secondary requirement imposed on the LQR algorithm is to minimize the distance traveled in response to a disturbance force, which can be achieved by configuring the LQR gain of the vehicle's linear displacement to be the second highest gain.

In order to optimize the robustness of the two-wheeled system, the LQR gains have to be maximized while satisfying the aforementioned tuning requirements. However, there are several factors limiting how high the LQR gains can be specified before the balancing performance of the platform begins to deteriorate. The greatest limiting factor in LQR robustness is sensor resolution. The on-board sensors are capable of measuring physical quantities with high resolutions. Unfortunately, when these sensor measurements are multiplied by the LQR gains inside the LQR algorithm, the lowest calculable motor torque command becomes equivalent to the resolution of the sensor multiplied by the LQR gain. This is not a problem for low LQR gains but when the LQR gains are high enough, the minimum calculable motor torque command becomes large enough to cause the platform to vibrate relentlessly.

Another factor limiting the robustness of the LQR algorithm is the software's sampling rate. Higher sampling rates are possible as long as the physical quantities measured by the sensors are changing at rates greater than twice the sensor resolution per sample period; this ensures that the Nyquist frequency is satisfied and velocity calculations are alias-free. However, when the software is sampling much faster than the sensor is able to detect a difference between the current and previous physical quantity,

the error in the vehicle's calculated velocities become large enough to deteriorate LQR performance. For the platform experiments performed in this thesis, a 100 Hz sampling rate was sufficient but future experiments may require higher sampling rates. However, since the software's maximum sampling rate is limited by sensor resolution, increasing sensor resolution would be enough to allow higher sampling rates.

The third limiting factor in optimizing the robustness of the LQR algorithm is the maximum allowable motor torque. Since it would not be safe or power efficient for the motor to be overexerting itself, the LQR gains should not be high enough to cause the motors to output high torque values. However, this limiting factor is not a concern since the limitations imposed on the LQR gains by sensor resolution restricts the calculated motor torque commands to values well below the maximum rated torque of the motors.

With the aforementioned requirements and limiting factors kept in mind, the LQR gains were adjusted until the two-wheeled system was exhibiting its optimal behavior. The LQR gains were calculated in MATLAB and tested on the platform. After testing various LQR gain combinations, the platform finally reached its best behavior when the following LQR gain matrices were implemented:

$$\mathbf{K}_{\phi} = \begin{bmatrix} -100.0000 & -60.7486 & 370.5861 & 80.6936 \end{bmatrix} \tag{9.1}$$

$$\mathbf{K}_{\psi} = \begin{bmatrix} 31.6228 & 3.6995 \end{bmatrix} \tag{9.2}$$

The corresponding weighting matrices used to achieve these LQR gains were:

$$\mathbf{Q}_{\phi} = \begin{bmatrix} 10000 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{\phi} = 1 \quad \mathbf{Q}_{\psi} = \begin{bmatrix} 1000 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{R}_{\psi} = 1 \tag{9.3}$$

9.1.3. State Variable Controllers Tuning

The software contains three state variable controllers designed to control the vehicle's linear displacement, pitch angle, and heading angle based on pitch and yaw load cell torque measurements. Unfortunately, due to the presence of heavy noise corrupting pitch load cell measurements, it was not possible to implement the state variable controllers that rely on these torque measurements. This means that the pitch controller cannot be used while the displacement controller will have to be reconfigured to use the vehicle's calculated linear displacement as the input instead of pitch load cell torque measurements. Therefore, this subsection will only be discussing the procedures used to tune the linear displacement and heading angle controllers.

Both of the linear displacement and heading angle controllers contain one input variable and four adjustable parameters that can be used to specify how fast the platform is allowed to displace along the x-axis or turn about its yaw axis. The input variables for the displacement and heading angle controllers are the vehicle's calculated linear displacement and yaw load cell torque measurement, respectively. The first two adjustable parameters are used to define the input variable's lower and upper limits that will determine when the platform will displace or turn at minimum/maximum speeds. Meanwhile, the other two adjustable parameters are used to specify the minimum and maximum speeds that the platform will displace along the x-axis or turn about its yaw axis.

For the displacement controller, the input variable's lower and upper limits were set to 0.02 m and 0.1 m, respectively, while the minimum and maximum allowable speeds along the x-axis were set to 0 m/s and 0.8 m/s, respectively. The lower and upper

limits for the input variable were selected based on controller performance during platform testing. Meanwhile, since the maximum allowable speed will vary depending on the gait characteristics of the user, the only criteria used to specify the speed limit was the interest to keep the value below the average human walking speed of 1.4 m/s [57].

For the heading angle controller, the input variable's lower and upper limits were set to 0.05 Nm and 1 Nm, respectively, while the minimum and maximum heading velocities were set to 0 rad/s and 1.0 rad/s, respectively. The lower and upper limits for the input variable were selected based on controller performance during platform tests. The maximum heading velocity was selected by observing controller performance using different speed limits.

9.2. Stability and Robustness Analysis

As part of assessing and evaluating the overall performance of the two-wheeled system, it is necessary to investigate the ability of the LQR algorithm to maintain platform stability in both the absence and presence of external disturbance forces. The tuning process and LQR gain matrices have already been identified in the previous section. Since this section will only be analyzing the performance of LQR controller, the state variable controllers have been disabled. The LQR algorithm will be executing at the software's sampling rate of 100 Hz. Platform data will be transmitted, processed, and stored in MATLAB at the same rate as the software's sampling speed.

9.2.1. Unperturbed Stability Analysis

One of the best ways to demonstrate and verify that the digital filters and LQR gains have been properly tuned is to conduct an analysis on the system's state variables while the platform is attempting to stabilize itself without external disturbances. This approach makes it much easier to diagnose and locate potential sources of error that may be causing the platform to behave erratically or out of tune. On the other hand, if the platform is able to balance upright with minimal motor effort and platform vibration then that would be a very good indication of a successfully implemented and properly tuned LQR controller.

The response of the two-wheeled platform with decoupled LQR control while experiencing no external disturbance forces can be seen in Fig. 9-1. The vehicle's linear displacement oscillates between ±2.5 mm while its linear velocity stay within ±0.015 m/s. Similarly, there is also barely any change to pitch angle and velocity. The yaw angle and velocity are also close to zero which indicates that the left and right wheel motors are rotating with equal but opposite angular displacements. These results indicate that the LQR controller is able to keep the platform balancing upright without allowing the platform to drift along the x-axis or turn about its yaw axis.

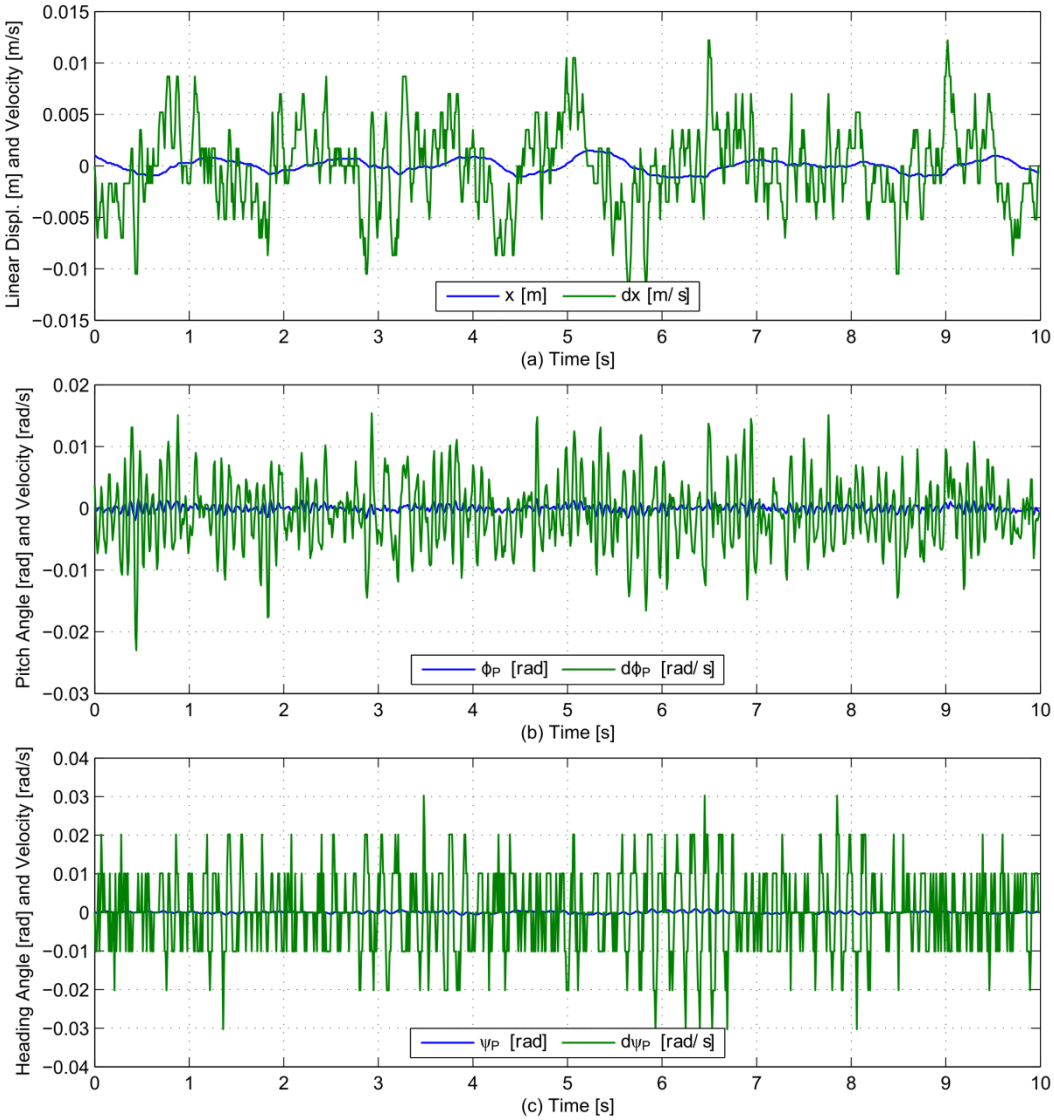

Figure 9-1. Response of the two-wheeled platform with decoupled LQR control while experiencing no external disturbance forces: (a) linear displacement and velocity, (b) pitch angle and velocity, and (c) yaw angle and velocity.

The effort exerted by the left and right wheel motors while the two-wheeled system is unperturbed can be seen in Fig. 9-2. The torque exerted by each motor while the platform is undisturbed varies between ±0.6 Nm, which is a reasonable torque range that is well within the capabilities of the motors. Furthermore, the graph shows that the calculated motor torques for the left and right wheels are nearly the same, as it would be

expected from a platform that is not rotating about its yaw axis. These results indicate that the LQR controller is able to balance the two-wheeled system with low motor effort.

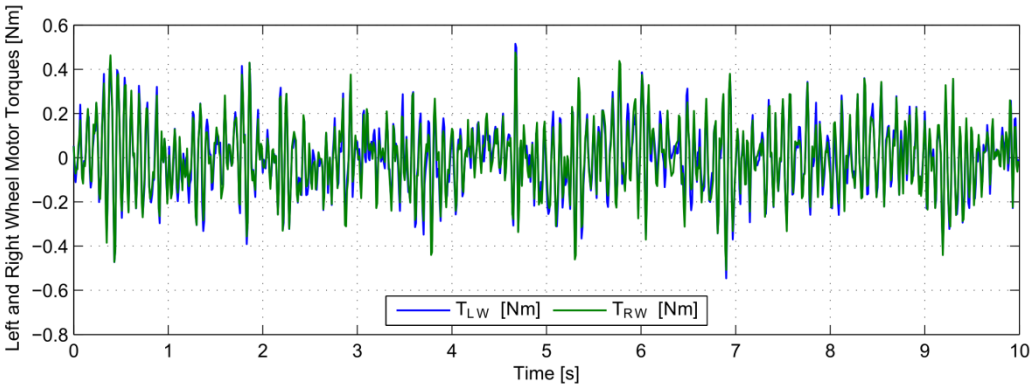

Figure 9-2. Left and right wheel motor responses while the two-wheeled platform is unperturbed.

9.2.2. Perturbed Stability Analysis

Ensuring that the LQR algorithm had been tuned well enough to balance the two-wheeled system with minimal motor effort was the first step in evaluating the performance of the LQR controller. The next step is to examine the ability of LQR algorithm to stabilize the platform and concurrently respond to external disturbance forces. Considering how the two-wheeled system is being designed to physically interact with a human, it is necessary that the LQR algorithm is robust enough to maintain the platform upright while rejecting disturbance forces generated by a user utilizing the device.

The response of the two-wheeled platform with decoupled LQR control while experiencing disturbance torques acting at the handlebars about the pitch axis is depicted in Fig. 9-3. The disturbance torque profile, shown in Fig. 9-3a, was generated by manually applying a torque about the pitch axis of the handlebars and quickly removing

the torque to observe the platform's ability to reclaim balance. The torque measurements from the pitch load cell were collected and relayed to MATLAB where the data was processed and a 2^{nd} order low-pass Butterworth filter with a cutoff frequency of 20 Hz was implemented to remove noise induced by platform vibration.

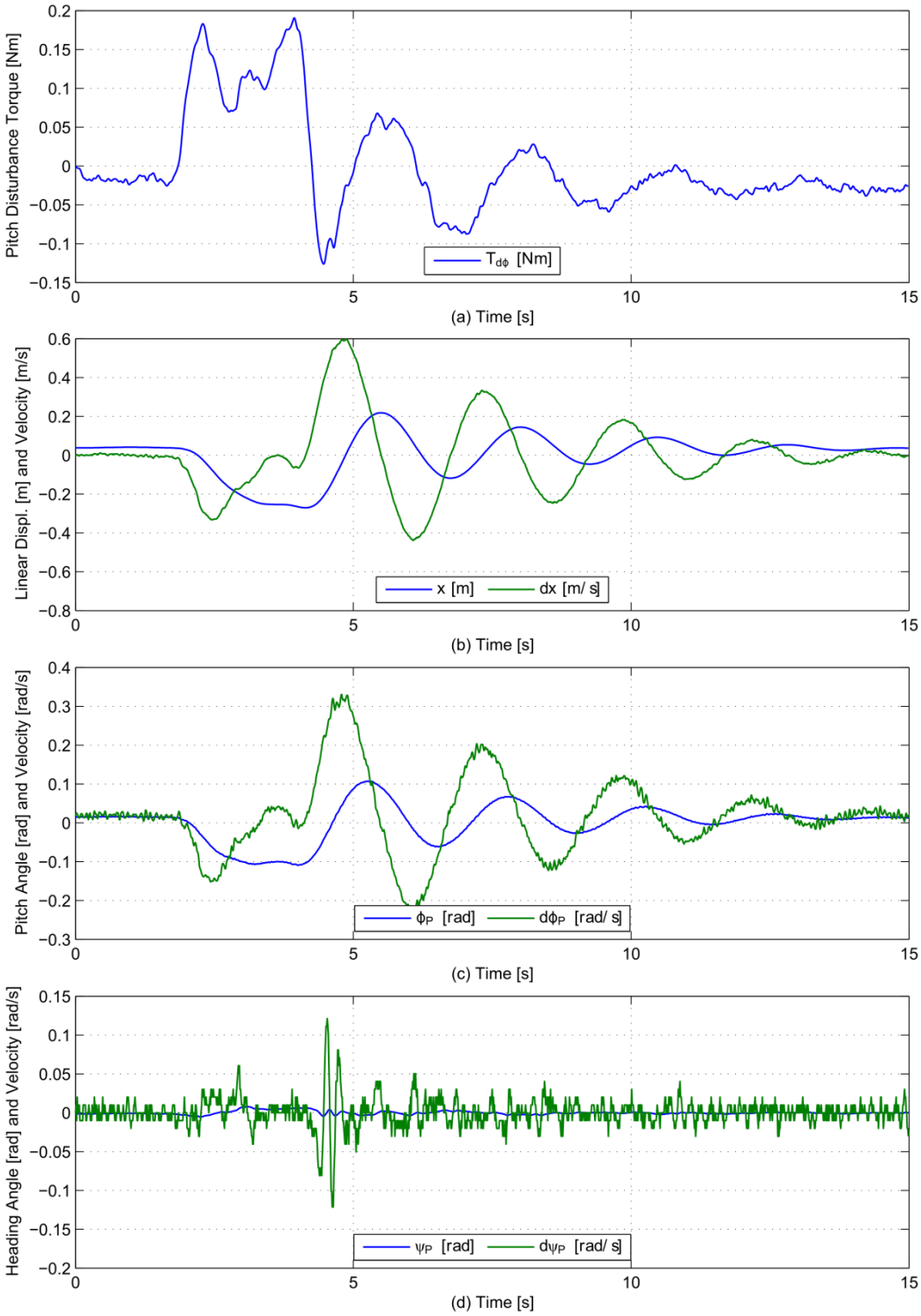

Figure 9-3. Response of the two-wheeled platform with decoupled LQR control while experiencing disturbance torques acting at the handlebars about the pitch axis: (a) pitch disturbance torque profile, (b) linear displacement and velocity, (c) pitch angle and velocity, and (d) yaw angle and velocity.

In response to the disturbance torque profile, the linear displacement and velocity of the vehicle increases to a peak of about 0.25 m and 0.6 m/s, respectively, and gradually begins to return to zero in oscillatory motions. A similar behavior is observed in the vehicle's pitch angle and velocity, which reached peaks of 0.1 rad and 0.32 rad/s, respectively. Meanwhile, since there were barely any disturbance torques acting about the yaw axis of the platform, the yaw angle and velocity remained close to zero. These results have favorably demonstrated the ability of the LQR controller to robustly reject external disturbance forces acting about the pitch axis while maintaining the platform in an upright position with optimal overshoot and settling performance.

The torque exerted by the left and right wheel motors while the two-wheeled system was stabilizing and rejecting disturbance torques acting about the pitch axis can be seen in Fig. 9-4. The motor behavior displayed in this figure corresponds to the state variable responses observed in Fig. 9-3. The peak torque response is roughly 1.4 Nm which is well below the maximum rate torque of 21.765 Nm per motor. Ideally, the motors should be exerting greater torques since that would increase the disturbance rejection performance of the LQR controller. However, since the LQR gains have already been tuned to exhibit maximum robustness within the limitations of the sensors, these results represent the current disturbance rejection capacity of the LQR algorithm.

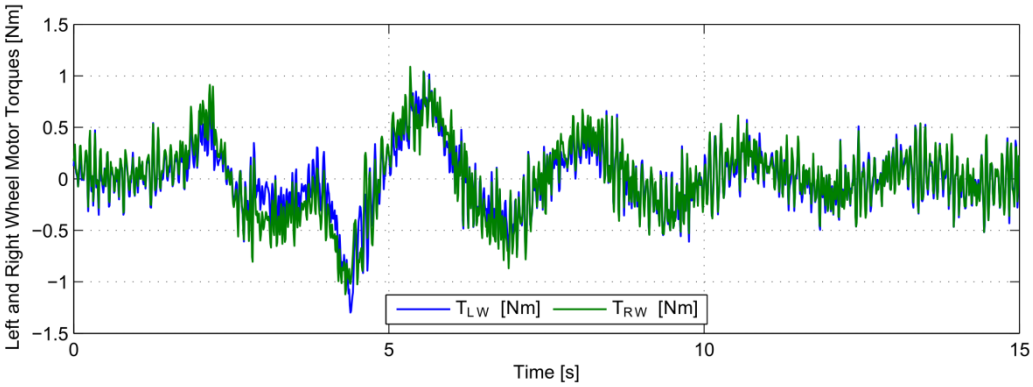

Figure 9-4. Left and right motor responses while the two-wheeled platform is subjected to disturbance torques acting at the handlebars about the pitch axis.

The preceding analysis was focused on examining the robustness of the LQR gains affecting the linear displacement, linear velocity, pitch angle, and pitch velocity of the two-wheeled system. The proceeding analysis will examine the robustness of the LQR gains affecting the yaw angle and velocity of the platform.

The response of the two-wheeled platform with decoupled LQR control while experiencing disturbance torques acting at the handlebars about the yaw axis is depicted in Fig. 9-5. The disturbance torque profile, shown in Fig. 9-5a, was generated by manually applying a brief torque about the platform's yaw axis followed by a short intermission before applying another torque about the yaw axis but in the opposite direction. The torque measurements were collected by the yaw load cell and forwarded to MATLAB for processing. However, unlike the pitch load cell, no additional filtering was necessary since the measured signal had very low noise.

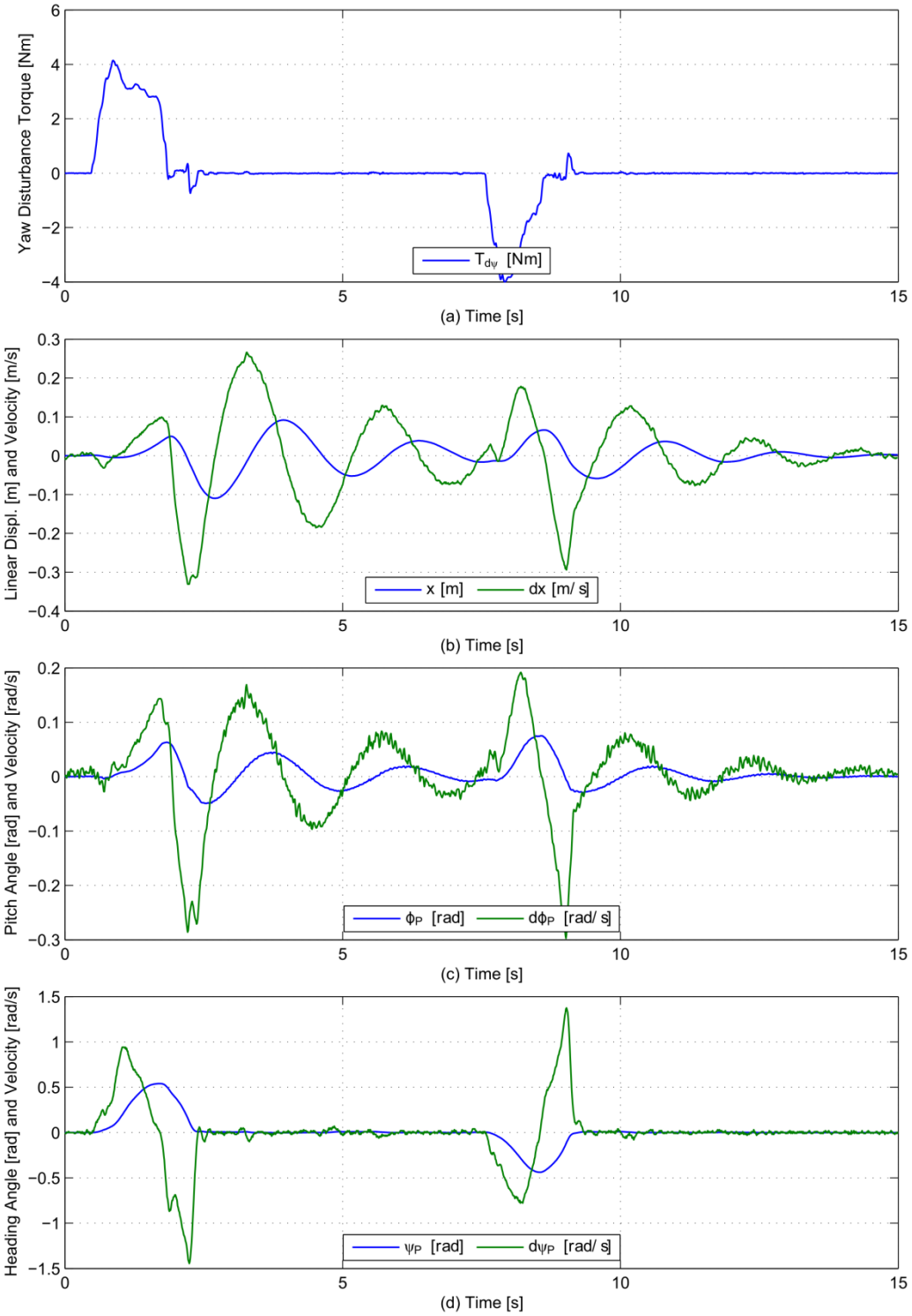

Figure 9-5. Response of the two-wheeled platform with decoupled LQR control while experiencing disturbance torques acting at the handlebars about the yaw axis: (a) yaw disturbance torque profile, (b) linear displacement and velocity, (c) pitch angle and velocity, and (d) yaw angle and velocity.

In response to the disturbance torque profile, the yaw angle and velocity of the vehicle reached peaks of about 0.5 rad and 1.5 rad/s, respectively, but quickly returned to zero once there was no longer any disturbance force acting about the handlebar's yaw axis. Some change in the vehicle's linear displacement, linear velocity, pitch angle, and pitch velocity did occur but that is because it was not possible to apply a manual disturbance torque about handlebar's yaw axis without some slight perturbation along the pitch axis. Overall, these results have favorably demonstrated the ability of the LQR controller to robustly reject external disturbance forces acting about the yaw axis while balancing the platform.

The torque exerted by the left and right wheel motors while the two-wheeled system was stabilizing and rejecting disturbance forces acting about the yaw axis can be seen in Fig. 9-6. The motor behavior displayed in this figure corresponds to the state variable responses observed in Fig. 9-5. Although each motor is capable of exerting a peak torque of 6.310 Nm, a torque saturation at 1.828 Nm can be observed. The motors are not allowed to exert peak torques of 6.310 Nm because each Puck^{TM} module has been programmed to limit the maximum amount of current drawn by each motor to 4 Amps. Given that the torque constant (K_t) for the motors are 0.457 Nm/Amps, a 4-Amp limit is equivalent to a torque limit of 1.828 Nm per motor. For the experiments performed in this thesis, a 1.828 Nm limit to the outputs of each motor has proven to be satisfactory. At this torque limit, the actuators still have enough torque to robustly return the yaw angle and velocity to zero with low overshoot and fast settling time.

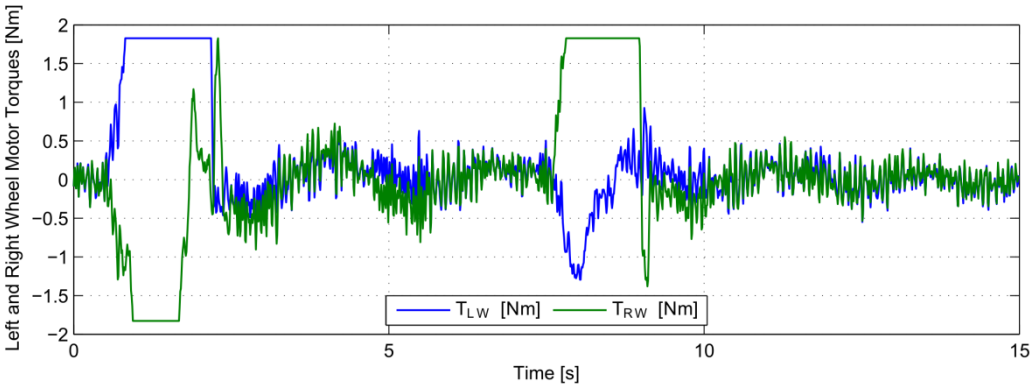

Figure 9-6. Left and right motor responses while the two-wheeled platform is subjected to disturbance torques acting at the handlebars about the yaw axis.

9.3. State Variable Controllers Evaluation

The robustness of the LQR algorithm was demonstrated and evaluated in the previous section. This section will examine the ability of the state variable controllers to issue position commands using disturbance forces generated by a user interacting with the platform. Another aspect of the algorithm that will be investigated in this section will be the ability of the state variable controllers to regulate the position of the platform without interfering with the LQR controller or compromising the general stability of the system. For the experiments performed in this section, the algorithm was executed at fixed sampling rate of 100 Hz. Platform data were collected and processed at the software's sampling rate. Furthermore, the performance of each state variable controller was tested and analyzed individually.

9.3.1. Displacement Controller

Smooth control of the vehicle's linear displacement can be accomplished using one of two displacement controllers available inside the software. One of these controllers uses pitch load cell measurements for calculating linear displacement

commands while the other controller relies on wheel encoder measurements. Since the pitch load cell measurements are too heavily corrupted by noise, the displacement controller that will be evaluated in this section is the *DisplacementControl_v2()* function.

If the wheels are located a certain distance in front or behind the platform's starting position, the displacement controller interprets this displacement as a user's desire to move the platform forwards or backwards. To test the performance of this controller, the platform was pushed forwards and backwards at a moderate walking speed. The linear displacement commands generated by the displacement controller in response to changes observed in the platform's linear displacement is recorded in Fig. 9-7a. The calculated linear displacement commands are smooth and steady as expected from the integrated minimum jerk trajectory model and PI controller.

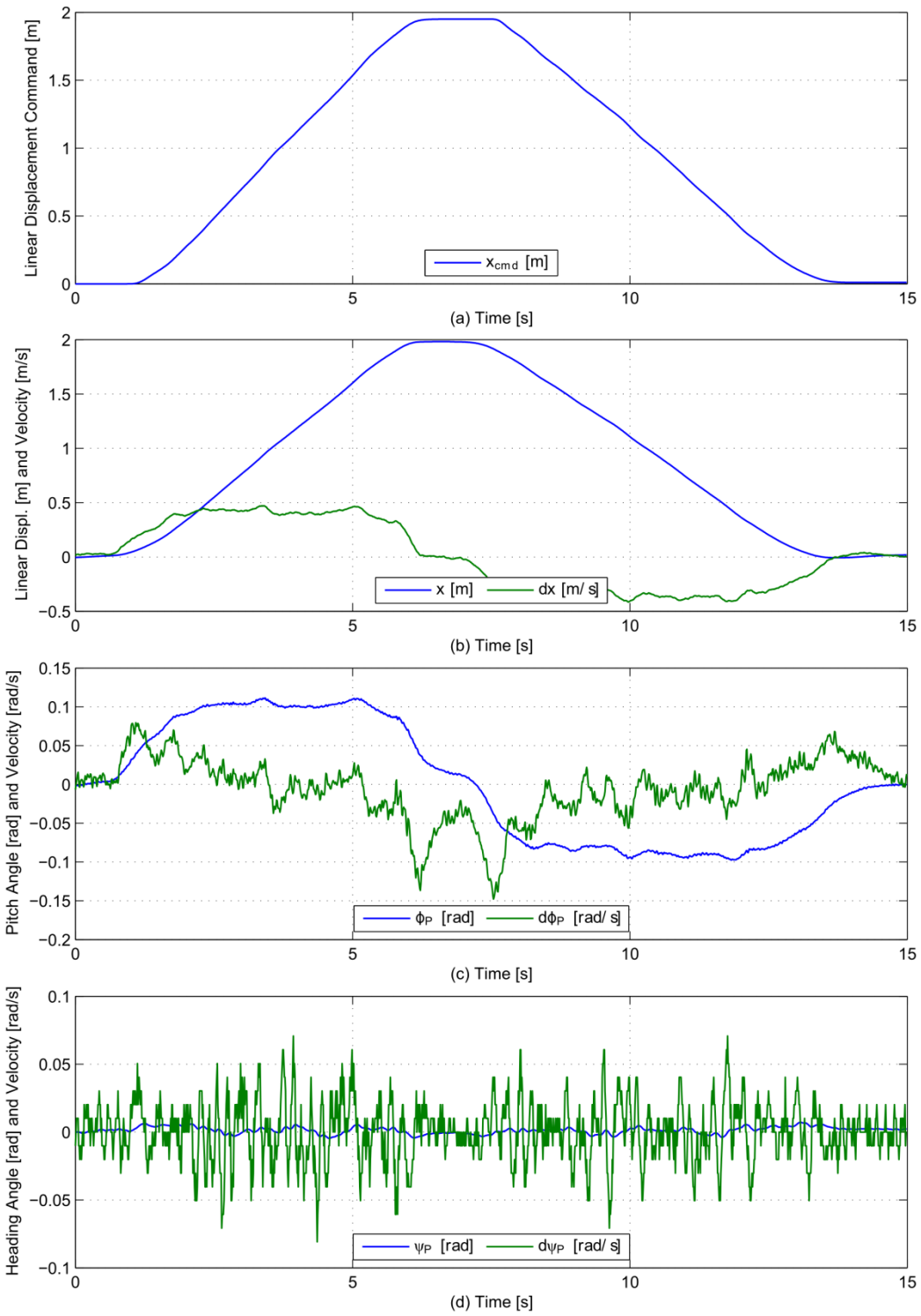

Figure 9-7. Response of the two-wheeled platform with decoupled LQR and displacement control while the platform is manually driven along the x-axis: (a) displacement controller response, (b) linear displacement and velocity, (c) pitch angle and velocity, and (d) yaw angle and velocity.

The effect of the displacement controller on the response of the two-wheeled system can be observed in the last three graphs of Fig. 9-7. As the linear displacement between the platform's equilibrium point and actual position along the x-axis begins to increase, the displacement controller smoothly shifts the equilibrium point until the user-induced disturbance terminates. Furthermore, it can be observed that the displacement controller is capable of regulating the vehicle's linear displacement without interfering with the LQR controller or compromising the general stability of the system.

The torque exerted by the left and right wheel motors while the two-wheeled system was stabilizing and manually driven along the x-axis can be seen in Fig. 9-8. The motor behavior displayed in this figure corresponds to the state variable responses observed in Fig. 9-7. The peak-to-peak torque command supplied to left and right wheel motors averages to about ±0.5 Nm, which is slightly greater than the peak-to-peak torque command for the unperturbed stability analysis case.

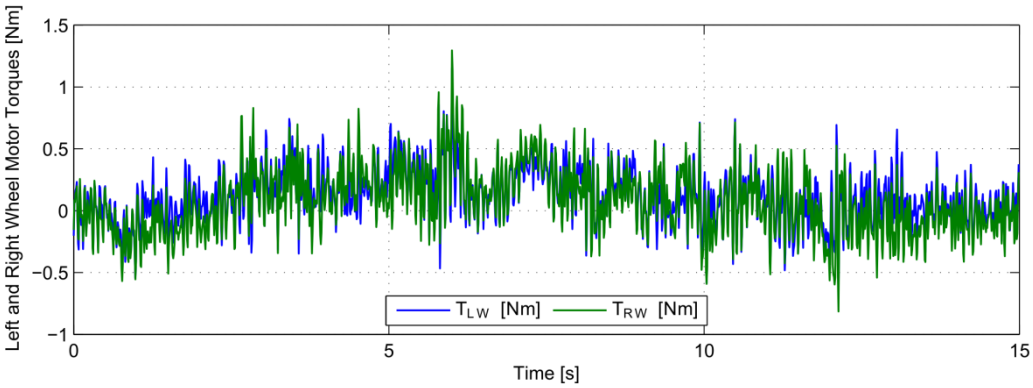

Figure 9-8. Left and right motor responses while the two-wheeled is manually driven along the x-axis.

9.3.2. Pitch Controller

As previously mentioned, it was not possible to implement the pitch controller because the pitch load cell torque measurements of the user's input torque were too small in magnitude and, therefore, indistinguishable from noise induced by platform vibrations. However, as aforementioned, the implementation of a 2^{nd} order low-pass Butterworth filter with a cutoff frequency of 20 Hz in MATLAB was sufficient to remove most of the noise that was corrupting the pitch load cell data seen in Fig. 9-3a. Therefore, it is quite likely that implementing a more robust low-pass filter in software may solve the noise problem.

9.3.3. Heading Angle Controller

The heading angle controller is responsible for controlling the heading angle of the vehicle using torque measurements collected from the yaw load cell. If the user applies a differential force on the handlebars that produces a torque reading on the yaw load cell, the heading angle controller interprets this differential force as the user's desire to change the heading angle of the platform. To test the performance of this controller, a manual disturbance torque was applied at the handlebar about the platform's yaw axis and measured by the yaw load cell. Since the yaw load cell measurements are already sufficiently filtered in software, no additional filtering was performed in MATLAB. The measured profile of this manual disturbance torque is presented in Fig. 9-9.

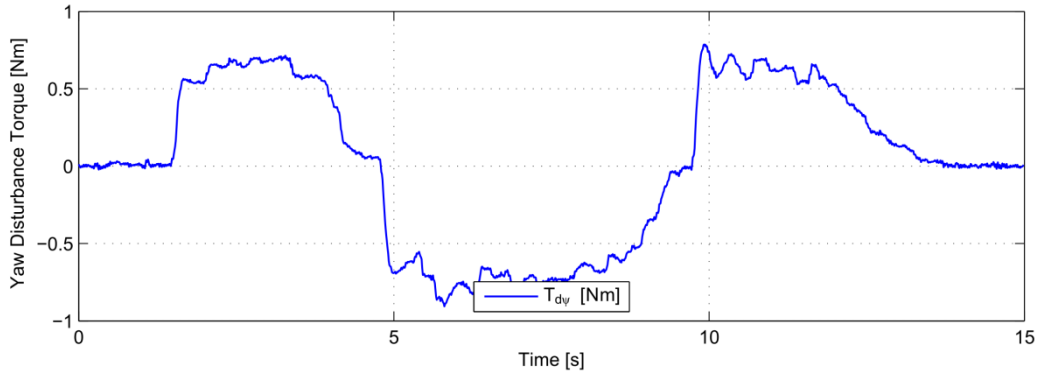

Figure 9-9. Manually generated disturbance torque profile acting at the handlebars about the yaw axis of the two-wheeled system.

The heading angle commands generated by the heading angle controller in response to changes observed in the platform's yaw load cell as a result of the disturbance torque acting about the yaw axis is recorded in Fig. 9-10. The performance of the heading angle controller is just as smooth and steady as the displacement controller, which is to be expected since both controllers share the minimum jerk trajectory model and PI controller.

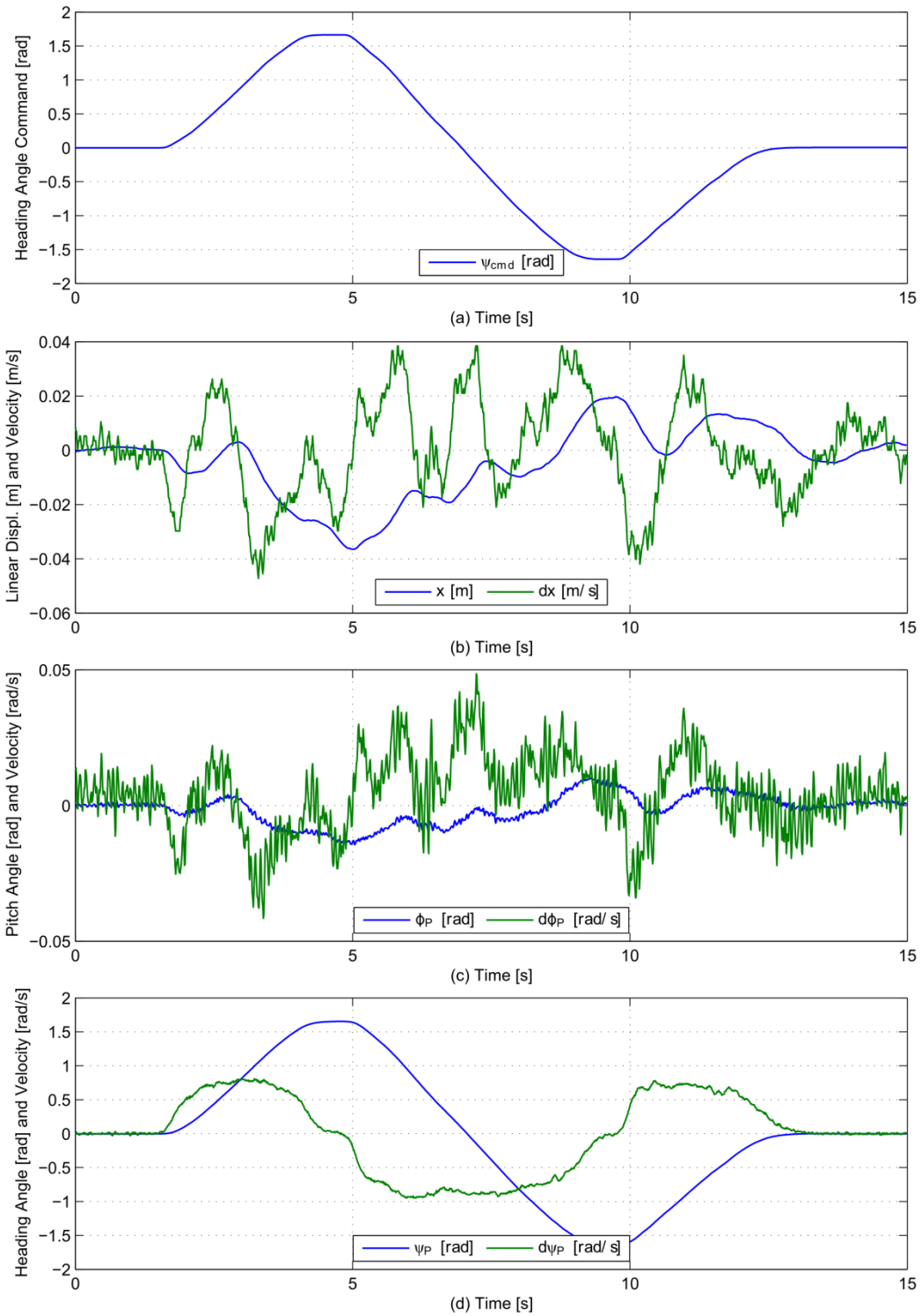

Figure 9-10. Response of the two-wheeled platform with decoupled LQR and heading angle control while the platform is manually rotated about the yaw axis: (a) heading angle controller response, (b) linear displacement and velocity, (c) pitch angle and velocity, and (d) yaw angle and velocity.

The effect of the heading angle controller on the response of the two-wheeled system can be observed in Fig. 9-10. As the torque applied to the handlebars increase past the controller's lower torque threshold, the heading angle controller smoothly changes the vehicle's heading angle until the user is no longer applying enough torque to activate the controller. While the heading angle controller is active and changing the vehicle's heading angle, the linear displacement, linear velocity, pitch angle, and pitch velocity responses are slightly affected by unintentional manual perturbations acting about the pitch axis. These results are strong indication that the heading angle controller is capable of regulating the vehicle's heading angle without interfering with the LQR controller or compromising the general stability of the system.

The torque commands supplied to the left and right wheel motors while the two-wheeled system was stabilizing and manually rotated about the yaw axis can be seen in Fig. 9-11. The motor behavior displayed in this figure corresponds to the state variable responses observed in Fig. 9-10. The average torque command supplied to each motor remain close to zero which is to be expected considering how the linear displacement and velocity of the platform is barely changing.

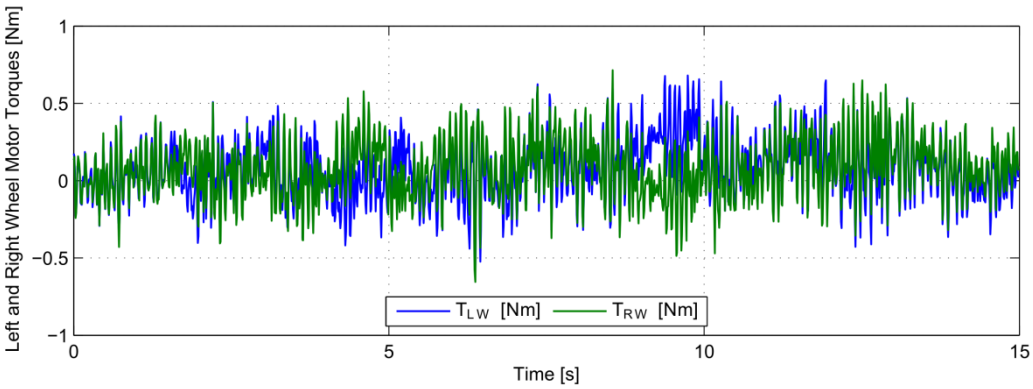

Figure 9-11. Left and right motor responses while the two-wheeled platform is manually rotated about the yaw axis.

9.4. Overall System Performance

In the unperturbed stability analysis, it was demonstrated that the LQR controller has been tuned well enough to balance the two-wheeled system with minimal motor effort and variation in the system's state variables. The perturbed stability analysis further substantiated the findings in the unperturbed stability analysis by exhibiting the ability of the LQR controller to robustly resist external disturbance forces. These results prove that the proposed LQR controller is capable of effectively balancing the two-wheeled system with robust stability and disturbance rejection performance.

After evaluating the robustness of the LQR controller, the state variable controllers were individually tested and evaluated. The ability of the displacement and heading controllers to smoothly regulate the platform's state variables with minimal jerk were thoroughly investigated and evaluated. It was also proven that the state variable controllers are capable of controlling the behavior of the platform without inhibiting the performance of the LQR controller or general stability of the two-wheeled system.

To finalize the evaluation of system performance, the two-wheeled platform was subjected to an experiment that would exhibit the ability of the algorithm to balance the platform while simultaneously regulating the vehicle's linear displacement and heading angle. The experiment can be broken down into 7 motions. (1) The user remains stationary while holding the platform at its handlebar; (2) the user pushes the platform forward approximately 2 meters; (3) a brief pause is taken; (4) the user turns the platform counterclockwise by approximately 90 degrees; (5) another brief pause is taken; (6) the user walks forward for another 2 meters; (7) the user stops. The behavior of the platform while the user pushes the device through these set of motions can be seen in Fig. 9-12.

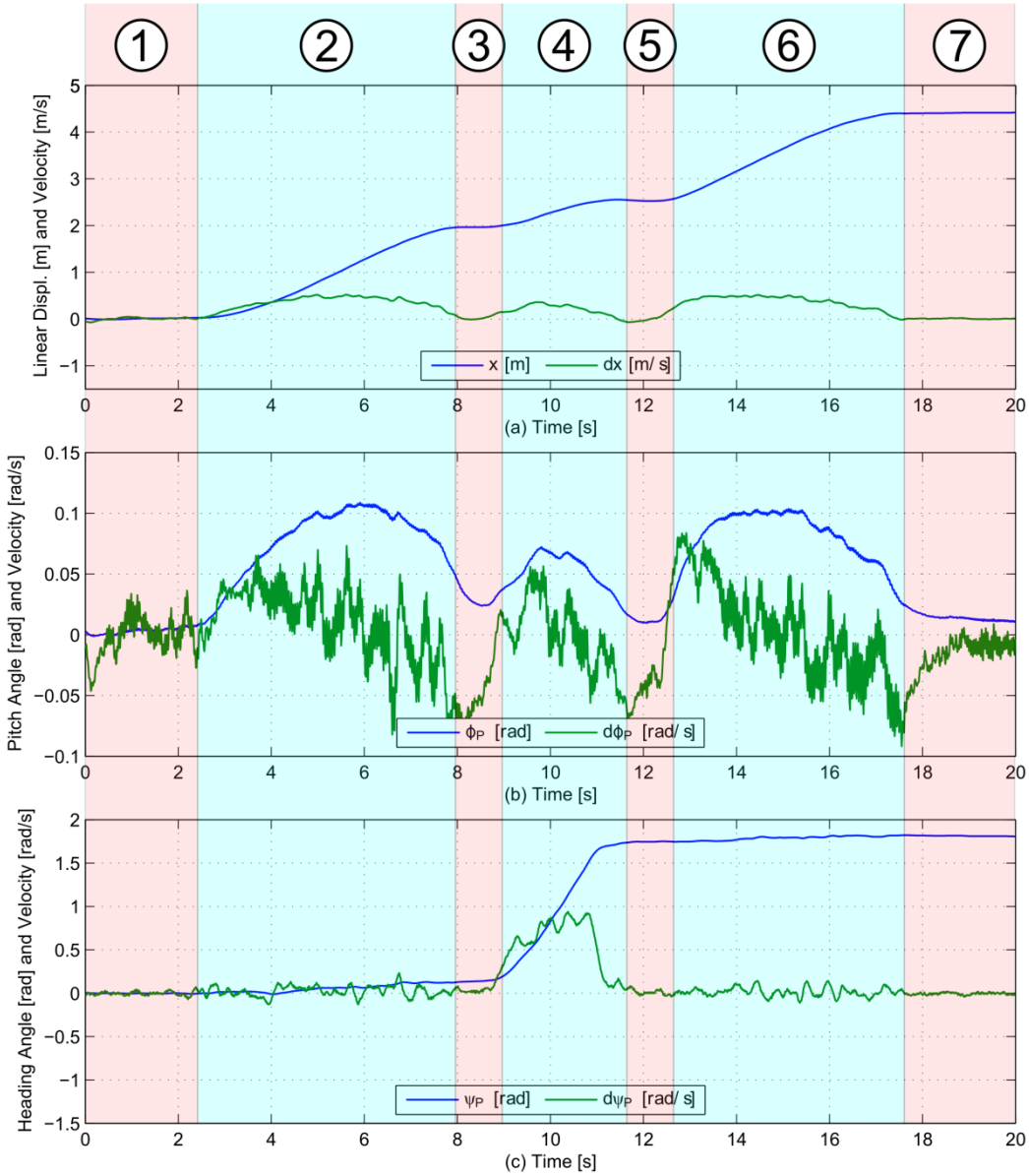

Figure 9-12. Response of the two-wheeled platform with decoupled LQR along with linear displacement and heading angle control while the platform is subjected to a series of motions: (a) linear displacement and velocity, (b) pitch angle and velocity, and (c) yaw angle and velocity.

The pitch disturbance torque applied to the handlebars along with the response of the displacement controller while the platform was being subjected to the set of motions described in the previous figure can be seen in Fig. 9-13.

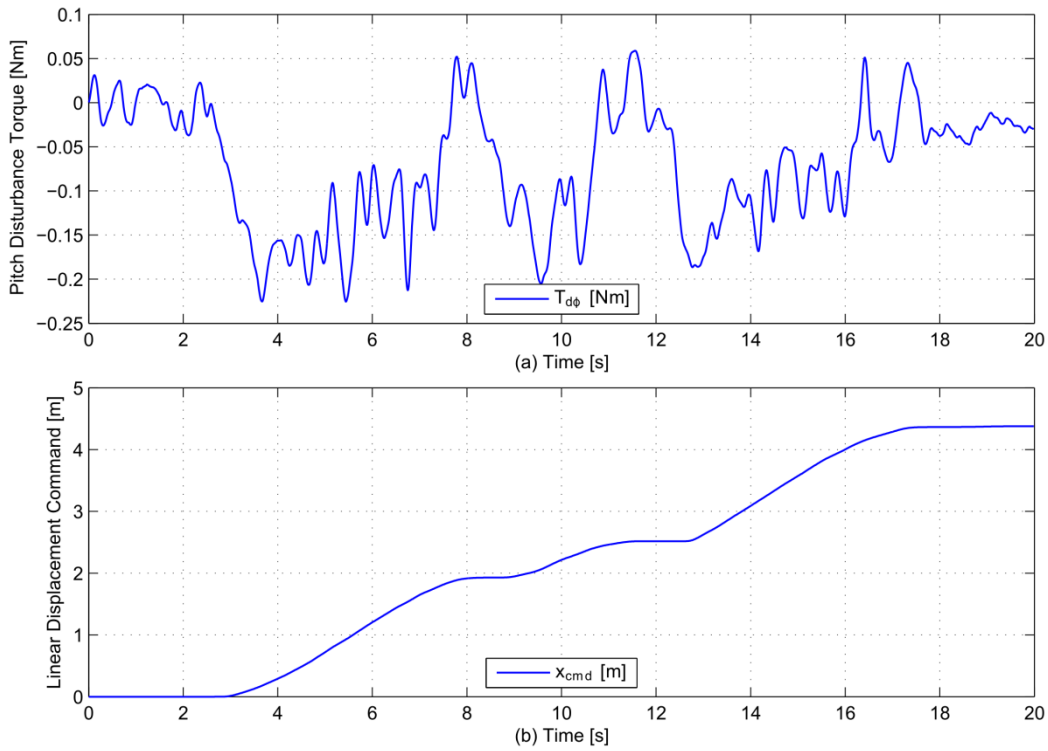

Figure 9-13. (a) Manually generated disturbance torque profile acting at the handlebars about the pitch axis of the two-wheeled system along with the (b) response produced by the displacement controller during the set of motions performed in Fig. 9-12.

The yaw disturbance torque applied to the handlebars along with the response of the heading angle controller while the platform was being subjected to the set of motions described in Fig. 9-12 can be seen in Fig. 9-14.

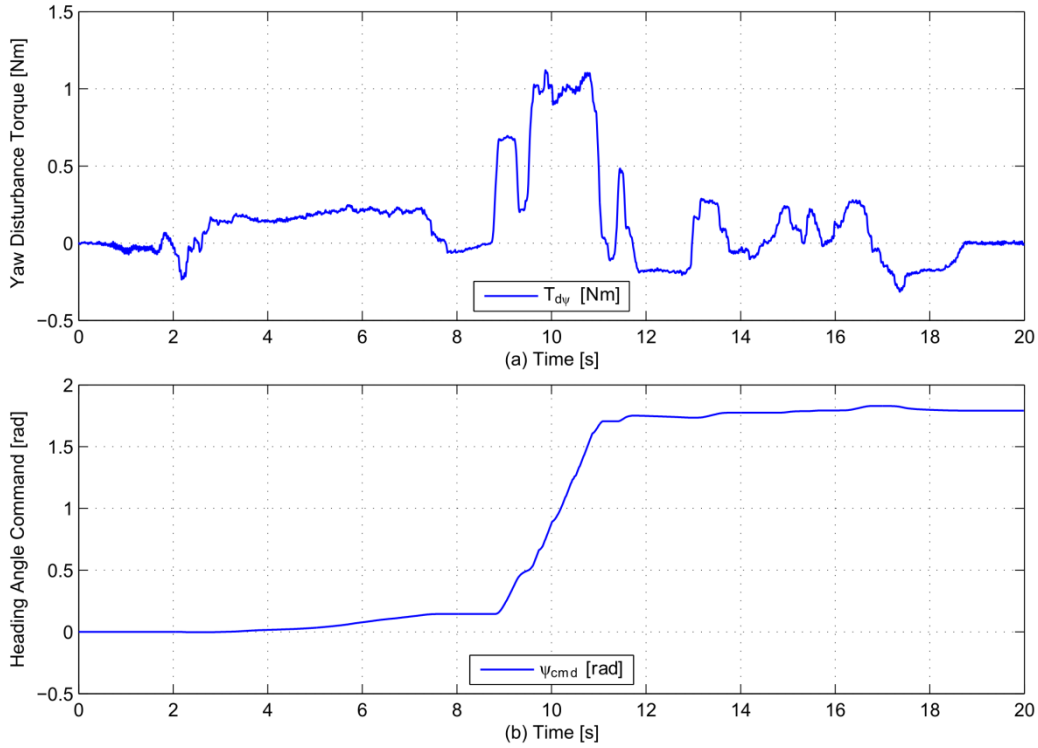

Figure 9-14. (a) Manually generated disturbance torque profile acting at the handlebars about the yaw axis of the two-wheeled system along with the (b) response produced by the heading angle controller during the set of motions performed in Fig. 9-12.

Meanwhile, the torque commands supplied to the left and right wheel motors while the two-wheeled system was subjected to the series of motions described in Fig. 9-12 can be seen in Fig. 9-15.

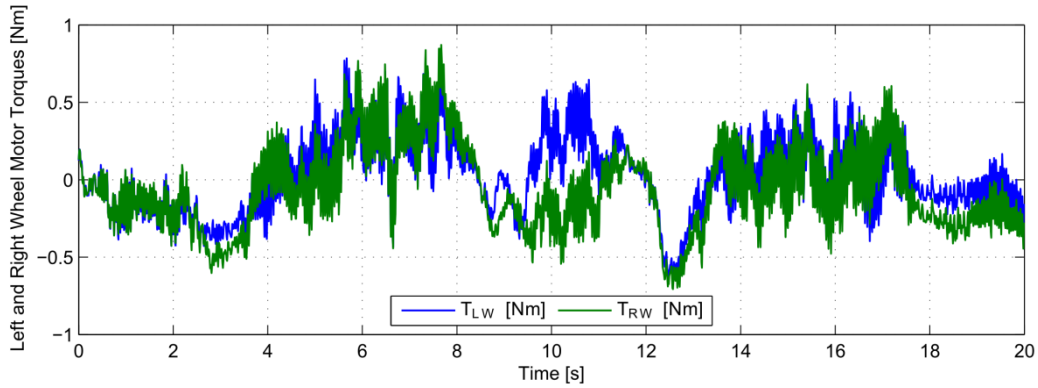

Figure 9-15. Left and right motor responses while the two-wheeled platform was subjected to the set of motions performed in Fig. 9-12.

Overall, the LQR and state variable controllers were able to effectively cooperate and control the behavior of the two-wheeled system without interfering each other. However, considering how the two-wheeled system is intended to be used by mobility-impaired users, it would be preferable to make the system more robust by increasing the stability and disturbance rejection performance of the LQR algorithm.

CHAPTER 10

CONCLUSION AND FUTURE WORK

This thesis introduced a robotic walker design and derived a mathematical model to emulate the dynamics of the two-wheeled system using Newtonian mechanics. A control strategy consisting of a decoupled LQR controller and three state variable controllers was developed to stabilize the platform and regulate its behavior with robust disturbance rejection performance. Simulation results revealed that the LQR controller was capable of robustly stabilizing the platform and rejecting external disturbances while the state variable controllers simultaneously regulated the system's position with smooth and minimum jerk control.

Following the development and simulation of a control strategy, the construction of a prototype for the two-wheeled system was set in motion. All system hardware were identified along with the development process of the two custom printed circuit boards used to power all on-board electronics and implement the control algorithm. With the circuit boards tested and functioning, the firmware and software components responsible for driving the hardware and controlling the two-wheeled system were developed and successfully implemented.

A detailed overview of the development and construction process of a prototype for the two-wheeled system was delivered, starting with solid model development followed by fabrication and assembly of mechanical components. Upon completing the physical prototype and integrating all on-board electronics, the software was implemented and tuned until the control algorithms were capable of stabilizing the prototype and regulating its position with optimal performance. To evaluate platform

performance, several experiments were conducted, confirming the ability of the LQR controller to robustly balance the platform while the state variable controllers regulate the platform's position with smooth and minimum jerk control.

10.1. Future Work

The work described in this thesis provides the foundation for future research on the development of a robotic walker designed to assist mobility impaired users with balance enhancement and fall prevention. Due to time limitations, there are still many areas of this research that could be improved and expanded in future work. Some of these areas include but are not limited to improvements on the prototype, circuit boards, software, and concept model.

10.1.1. Prototype Improvements

One of the improvements that could be performed on the current prototype is to fix or redesign the mechanical fixture holding the pitch load cell in place. Currently, one of the threads securing the pitch load cell is ripped, weakening the ability of the mounting fixture to rigidly hold the load cell in place. After some cycles of manually disturbing the pendulum, the screws holding pitch load cell eventually becomes loose due to non-optimal design and ripped thread. Considering how redesigning the mounting fixture wouldn't take much time, it would be better to make a more robust mechanical fixture for the pitch load cell instead of trying to fix the ripped thread.

Considering the amount of noise that the swinging pendulum induces on pitch load cell measurements, one recommendation would be to reduce the mass and inertia

directly above the load cell by placing the load cell higher along the pendulum. Making this change would reduce the torque that the load cell would measure due to the pendulum's swinging inertia and allow the load cell to more accurately measure the force acting on the handlebars. Another improvement would be to reduce the overall weight of the platform by replacing the aluminum components that do not need to be strong with a lighter material such as hardened plastic.

10.1.2. Circuit Board

In the power management PCB, a problem that needs to be fixed in future work is the battery monitor circuit which has a unity gain differential amplifier that was incorrectly routed. The 18V Zener diode used in the battery monitor circuit would also have to be updated to a 24V Zener diode in order to accommodate the two 4-cell LiPo battery packs that are connected in series. Another minor recommendation to the power management PCB would be to increase the hole size used in the footprint for the toroid power inductor, PE-92102. Currently, the leads of this toroid inductor did not fit into the footprint and had to be soldered standing up on the PCB's surface. This is a problem because it makes the solder joint very fragile and a small force applied to the toroid could break the joint, opening the circuit.

In the main PCB, the +2.5V voltage reference circuit is no longer needed due to a circuit design error in the A/D converter circuits. The A/D converter circuits were designed under the false pretense that the analog input channels could handle input voltages between 0 and +5V. However, it was later discovered that the analog input channels of the A/D converters can only support input voltages between -1V and +1V. A

temporary solution that did not involve redesigning the circuit boards was implemented at the cost of a 50% reduction to the torque range that each load cell is capable of measuring. Another change that could be made to the main PCB would be to remove some of the LEDs that were intended for debugging during the early stages of software development.

10.1.3. Software Development

Once the battery monitor circuit is fixed or altered, an important addition to the software would be to add battery management and monitoring. Currently, there is nothing preventing the platform from over-discharging the LiPo battery packs, which can result in permanent battery pack damage or a fire hazard. Some recommended updates to the software would be to optimize the performance algorithm by replacing inefficient code segments with more efficient alternatives. Furthermore, it would also be favorable to reduce the microcontroller's clock rate and baud rate of peripheral modules in order to minimize power consumption.

Another important consideration would be to check the software for code segments that could cause arithmetic errors. A code segment that is susceptible to an arithmetic error is the *static float x_cmdold* variable, which is used to hold the vehicle's total linear displacement in meters, located inside the *DisplacementControl_v2()* function. The problem with how this variable is handled is that once the vehicle's total displacement exceeds the available storage space assigned to the 32-bit floating-point value *x_cmdold*, a case of integer overflow will occur.

Although it may be computationally expensive, sensor filtering may be improved by replacing the RMA filters with Kalman filters. Increased LQR controller robustness is desired in order for the platform to achieve a behavior that is appropriate for a mobility-impaired user. Thus, any software modification that could improve the performance of the control algorithm is recommended. Additionally, considering how MATLAB's built-in Butterworth filter was able to filter out the noise corrupting pitch load cell measurements, implementing a better low-pass filter in software may be a viable solution to enabling pitch control.

10.1.4. Conceptual Model Revision

There are several modifications to the concept model introduced in this thesis that could improve the general performance of the robotic walker design. A major problem with the current two-wheeled model is that if a user wants to perform a 180 degree rotation, the user is required to walk along the vehicle's turning radius. Compared to a standard rollator walker that can simply be dragged sideways while the user remains at the center of rotation, the current robotic walker design is greatly lacking in maneuverability. To increase the maneuverability, it would be best to grant the platform omni-directional capabilities by implementing motorized omni wheels.

Another improvement to platform maneuverability would be to modify the overall geometry of the platform so that the user's hip is intersecting or adjacent to the wheels' axis of rotation. For this modification to work, the chassis of the vehicle would have to be reconfigured to arch around the user. In this arch chassis configuration, there would not be enough space between the user and a doorway to place the motors in line with the

wheels' axis of rotation. Therefore, in addition to designing a unique configuration where the motors are perpendicular to the wheels' axis of rotation, all on-board electronics would also have to be cleverly relocated. Redesigning the platform to such extent may be a daunting task but the new concept model would have superior maneuverability and fall prevention capabilities. Another improvement to the platform's fall prevention capabilities would be to use telescoping robotic arms instead of multi-joint robotic arms. Lastly, but not least, would be addition of a touch screen interface.

APPENDIX A

MinJerkTraj FUNCTION SCRIPT

Script of the *MinJerkTraj* function used in the Simulink block diagram of the displacement controller, Fig. 5-2.

```
function dx_cmd = MinJerkTraj(xi,xf,yi,yf,t)
%#eml
dx_cmd = yi + (yf - yi)*(10*((t-xi)/(xf-xi))^3 - 15*((t-xi)/(xf-xi))^4
+ 6*((t-xi)/(xf-xi))^5);
```

Script of the *MinJerkTraj* function used in the Simulink block diagram of the pitch controller, Fig. 5-4.

```
function phi_cmd = MinJerkTraj(xi,xf,yi,yf,t)
%#eml
phi_cmd = yi + (yf - yi)*(10*((t-xi)/(xf-xi))^3 - 15*((t-xi)/(xf-xi))^4
+ 6*((t-xi)/(xf-xi))^5);
```

Script of the *MinJerkTraj* function used in the Simulink block diagram of the heading angle controller, Fig. 5-5.

```
function dpsi_cmd = MinJerkTraj(xi,xf,yi,yf,t)
%#eml
dpsi_cmd = yi + (yf - yi)*(10*((t-xi)/(xf-xi))^3 - 15*((t-xi)/(xf-
xi))^4 + 6*((t-xi)/(xf-xi))^5);
```

APPENDIX B

SYSTEM SIMULATIONS, MATLAB FILE

```
% Name: DecoupledLQRwStateControl_TWIP.m
% Creation Date: 06/15/2012
% Author: Airton R. da Silva Jr.
% Comments: This file contains the variables and functions used perform decoupled LQR control
%     of the TWIP robotic walker.

clc;
% close all;
% clear all;

%%  Variable initialization
m_T = 13.245;        % Total mass of assembled two-wheeled system [kg]
m_w = 0.90;          % Mass of the wheel [kg]
m_pc = m_T - 2*m_w;  % Combined mass of the pendulum and chassis [kg]
D_1 = 0.346;         % Dist. between the L & R wheels along the y-axis [m]
l_g1 = 0.3;          % Dist. from point O to the CG of the pendulum [m]
r = 0.114;           % Radius of the wheel [m]
g = 9.81;            % Gravitational acceleration [m/s^2]
J_p = m_pc*l_g1^2;   % Moment of iner. of the pendulum about the y-axis [kg*m^2]
J_w = m_w*(0.7*r)^2; % Moment of iner. of the wheel about the y-axis [kg*m^2]
J_wz = 0.06045;      % Moment of iner. of both wheels about the z-axis [kg*m^2]
J_cz = 0.163686;     % Moment of iner. of the pend/chassis about the z-axis [kg*m^2]
J_py = J_p + 2*J_w;  % Moment of iner. of the pend/chassis about the y-axis [kg*m^2]
J_pz = J_wz + J_cz;  % Moment of iner. of the pend/chassis/wheels about the z-axis [kg*m^2]

% Decoupling controller parameters
G11 = 0.5;
G12 = 0.5;
G21 = 0.5;
G22 = -0.5;

%% State-space equation
% Variable definition
J_psi = J_pz + (D_1^2/2)*(m_w + J_w/r^2);
J_theta = J_py + m_pc*l_g1^2;
beta = 2*m_w + 2*J_w/r^2 + m_pc;
alpha = J_theta*beta - m_pc^2*l_g1^2;

% Matrix elements
A23 = (m_pc^2*g*l_g1^2)/alpha;
A43 = (m_pc*g*l_g1*beta)/alpha;

B21 = (J_theta - m_pc*l_g1*r)/(r*alpha);
B22 = B21;
B23 = (m_pc*l_g1)/alpha;

B41 = (m_pc*l_g1 - r*beta)/(r*alpha);
B42 = B41;
B43 = beta/alpha;

B61 = D_1/(2*r*J_psi);
B62 = -B61;
B64 = 1/(J_psi);

% State-space representation of coupled system dynamics
A = [0   1   0   0   0   0;
     0   0   A23 0   0   0;
     0   0   0   1   0   0;
     0   0   A43 0   0   0;
     0   0   0   0   0   1;
     0   0   0   0   0   0];
B = [0   0   0   0;
     B21 B22 B23 0;
     0   0   0   0;
     B41 B42 B43 0;
     0   0   0   0;
     B61 B62 0  B64];
C = eye(6);
```

```
D = zeros(6,4);

% Omit disturbance forces
B_ND = B*[1 0 0 0; 0 1 0 0]';
D_ND = D*[1 0 0 0; 0 1 0 0]';

% State-space representation of decoupled system dynamics: vehicle displ. and pitch dynamics
A1 = [0  1   0   0;
      0  0   A23 0;
      0  0   0   1;
      0  0   A43 0];
B1 = [0;
      B21;
      0;
      B41];
C1 = eye(4);
D1 = zeros(4,1);
% State-space representation of decoupled system dynamics: yaw dynamics
A2 = [0  1;
      0  0];
B2 = [0;
      B61];
C2 = eye(2);
D2 = zeros(2,1);

%% Calculating the LQR gain matrices
% x is the weighting for the vehicle's linear displacement
% y is the weighting for the vehicle's pitch angle
% z is the weighting for the vehicle's heading angle
x = 10000; y = 1000; z = 1000;

% Calculating LQR gain matrix for vehicle displ. and pitch dynamics
Q1 = [x        0        0     0 ;      % Weighting matrix for the outputs: x, dx, phi, dphi
      0        1        0     0 ;
      0        0        y     0 ;
      0        0        0     1];
R1 = 1;                                 % Weighting matrix for the input: T_phi
BRinverse1 = B1*inv(R1)*B1';
P1 = are(A1,BRinverse1,Q1);             % Algebraic Riccati Equation solution

% Calculating LQR gain matrix for yaw dynamics
Q2 = [z   0                             % Weighting matrix for the outputs: psi, dpsi
      0   1];
R2 = R1;                                % Weighting matrix for the input: T_psi
BRinverse2 = B2*inv(R2)*B2';
P2 = are(A2,BRinverse2,Q2);             % Algebraic Riccati Equation solution

% Feedback Gains for the decoupled dynamics of the two-wheeled system
K_phi = inv(R1)*B1'*P1;
K_psi = inv(R2)*B2'*P2;

%% Using MATLAB to simulate the decoupled LQR control scheme
% Implementing decoupled LQR control
K_phipsi = [K_phi    0  0;              % Decoupled LQR gain matrix
            0  0  0  0  K_psi];
G = [0.5 0.5; 0.5 -0.5];                % Decoupling controller matrix
Ac = (A-B_ND*G*K_phipsi);
Bc = B;
Cc = C;
Dc = D;
T = 0:0.01:12;
U = [0;0;1;1]*5*ones(size(T));
U(:,1:50) = 0;
U(:,901:1201) = 0;
[Y,X] = lsim(Ac,Bc,Cc,Dc,U,T);
figure(1)
plot(T,Y)
title('System Response to 5 N Step Disturbance w/ Decoupled LQR Control')
xlabel('Time [s]')
ylabel('Dimension Varies w.r.t. State Variables')
h = legend('$x$ $[m]$','$\dot{x}$ $[m/s]$','$\phi$ $[rad]$',...
    '$\dot{\phi}$ $[rad/s]$','$\psi$ $[rad]$','$\dot{\psi}$ $[rad/s]$');
set(h,'interpreter','latex')
grid on
```

APPENDIX C

MAIN CIRCUIT BOARD SCHEMATICS

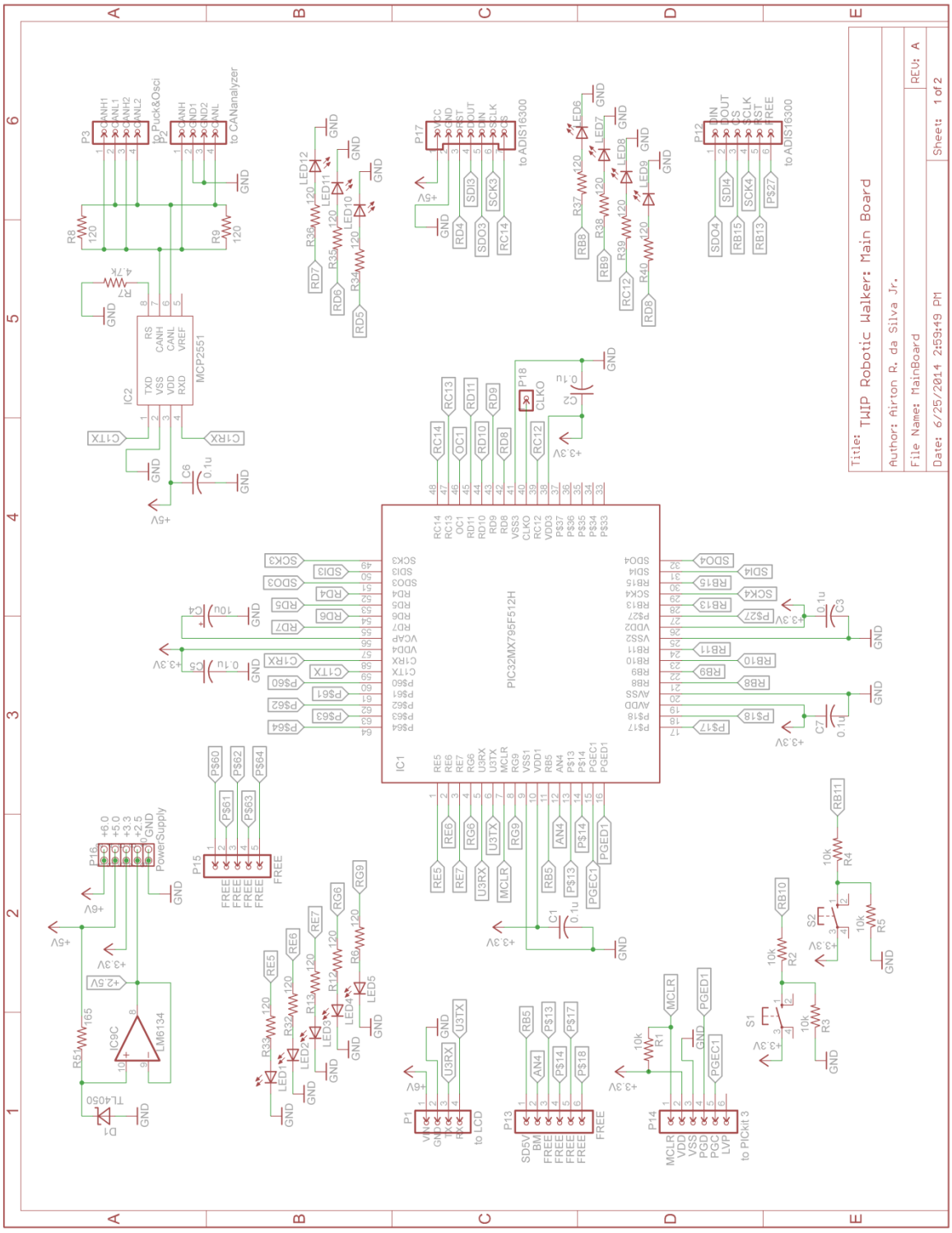

Figure C-1. Main circuit board schematics [Part 1].

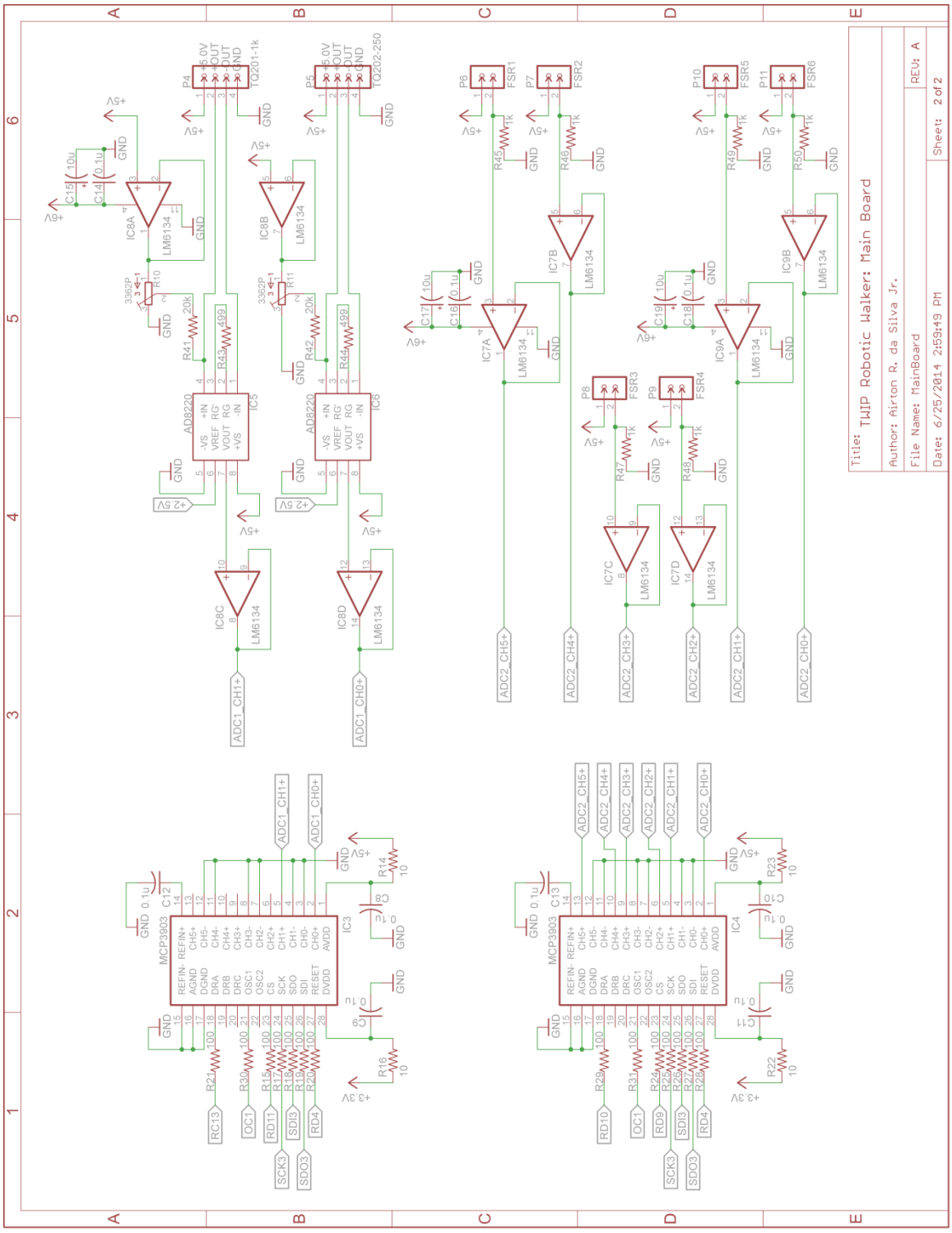

Figure C-2. Main circuit board schematics [Part 2].

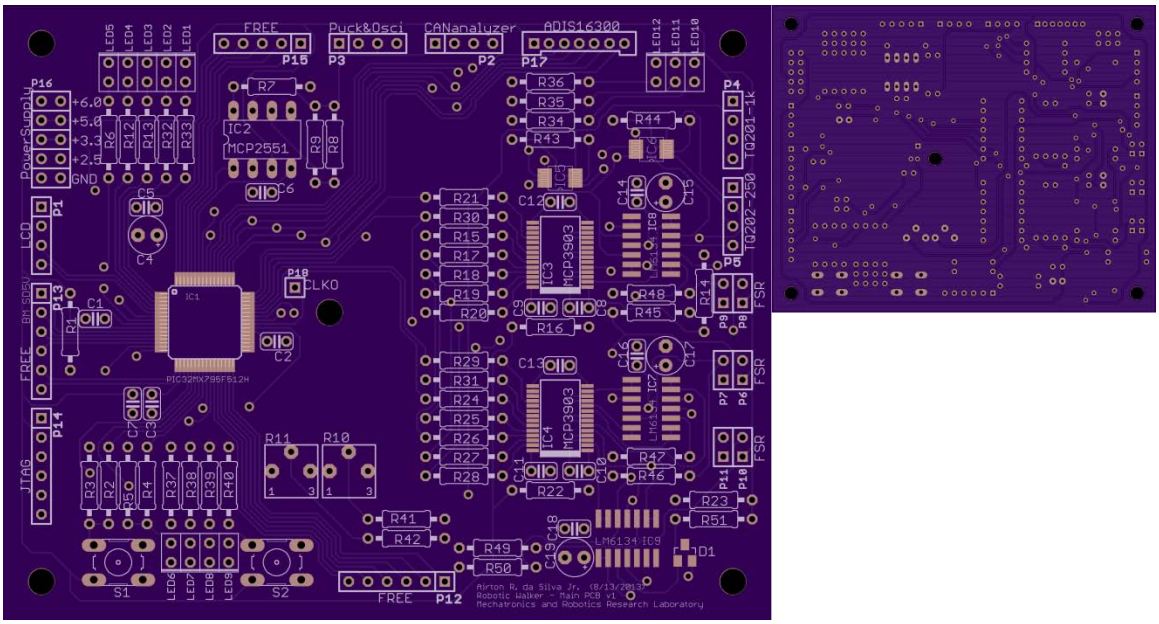

Figure C-3. Front and back view of main PCB layout.

APPENDIX D

POWER MANAGEMENT CIRCUIT BOARD SCHEMATICS

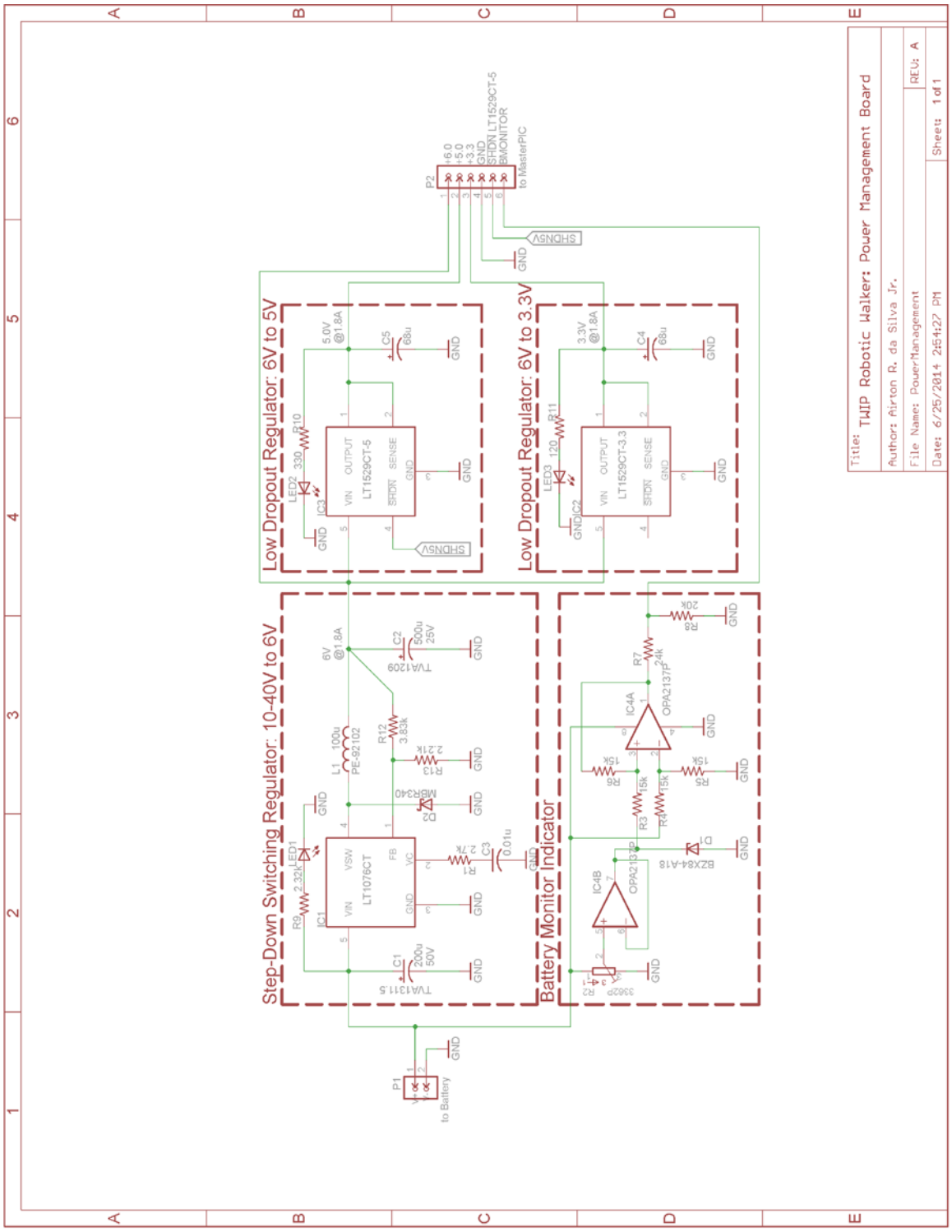

Figure D-1. Power management circuit board schematics.

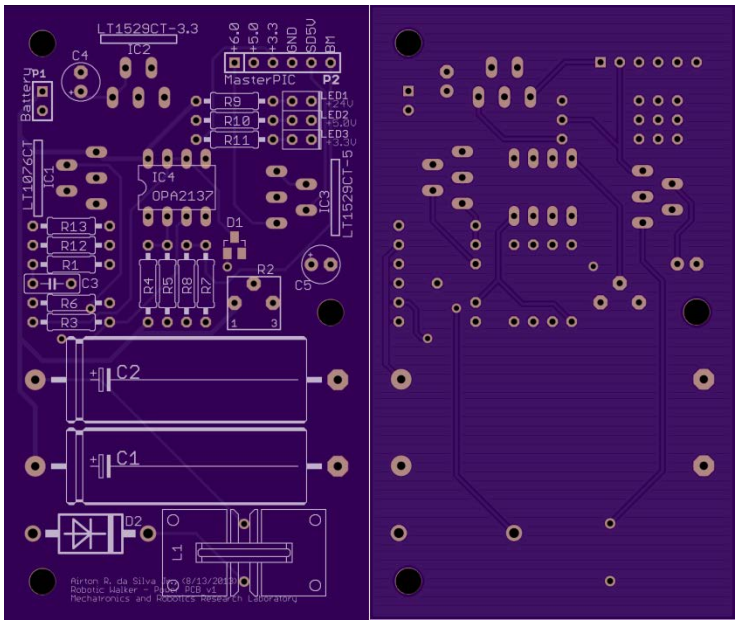

Figure D-2. Front and back view of power management PCB layout.

APPENDIX E

BILL OF MATERIALS – ELECTRONIC COMPONENTS

Table E-1. Bill of materials for the main circuit board.

Bill of Materials: Main Circuit Board						
Reference	**QTY**	**Part Number**	**Description**	**Supplier**	**Unit Pricing**	**Total Cost**
MAIN PCB	1	Main PCB	Main PCB	OSH Park	$21.33	$21.33
C1<>C3, C5<> C14, C16, C18	15	SR295E104MAR	CAP CER 0.1µF 50V ±20% RADIAL	Digi-Key	$0.24	$3.60
C4, C15, C17, C19	4	UPM1H100MDD	CAP ALUM 10µF 50V ±20% RADIAL	Digi-Key	$0.24	$0.96
D1	1	TL4050A25IDBZR	IC VREF SHUNT PREC 2.5V ±0.1%	Digi-Key	$2.07	$2.07
IC1	1	PIC32MX795F512H-80I/PT	IC MCU 32BIT 512KB FLASH	Digi-Key	$10.50	$10.50
IC2	1	MCP2551-I/P	IC TRANSCEIVER CAN HI-SPD	Digi-Key	$1.22	$1.22
IC3, IC4	2	MCP3903-I/SS	IC AFE 24BIT 64KSPS	Digi-Key	$5.12	$10.24
IC5, IC6	2	AD8220ARMZ	IC AMP INST JFET R-R 15MA	Digi-Key	$6.17	$12.34
IC7, IC8, IC9	3	LM6134BIM/NOPB	IC OP AMP QUAD H SPEED LP	Digi-Key	$3.78	$11.34
LED1<>LED12	13	SLR-56MG3F	LED GREEN DIFFUSED Vf = 2.1V, i = 10mA	Digi-Key	$0.53	$6.89
P1<>P5 (Male)	6	B4B-XH-A(LF)(SN)	CONN HEADER JST Shrouded	Digi-Key	$0.20	$1.20
P1<>P5 (Female)	6	XHP-4	CONN HOUSING JST Shrouded	Digi-Key	$0.10	$0.60
P6<>P11 (Male)	6	B2B-XH-A (LF)(SN)(P)	CONN HEADER JST Shrouded	Digi-Key	$0.14	$0.84
P6<>P11 (Female)	6	XHP-2	CONN HOUSING JST Shrouded	Digi-Key	$0.09	$0.54
P12<>P14 (Male)	3	B6B-XH-A(LF)(SN)	CONN HEADER JST Shrouded	Digi-Key	$0.26	$0.78
P12<>P14 (Female)	3	XHP-6	CONN HOUSING JST Shrouded	Digi-Key	$0.14	$0.42
P15 (Male)	1	B5B-XH-A(LF)(SN)	CONN HEADER JST Shrouded	Digi-Key	$0.24	$0.24
P15 (Female)	1	XHP-5	CONN HOUSING JST Shrouded	Digi-Key	$0.12	$0.12
P16 (Male)	1	B10B-XADSS-N(LF)(SN)	CONN HEADER JST Shrouded [PWRSPLY]	Digi-Key	$0.59	$0.59
P16 (Female)	1	XADRP-10V(P)	CONN HOUSING JST Shrouded [PWRSPLY]	Digi-Key	$0.32	$0.32
P17 (Male)	1	B07B-PASK(LF)(SN)	CONN HEADER JST Shrouded [ADIS16300]	Digi-Key	$0.44	$0.44
P17 (Female)	1	PAP-07V-S	CONN HOUSING JST Shrouded [ADIS16300]	Digi-Key	$0.17	$0.17
P18 (Male)	1	9-146274-0	CONN HEADER BREAKAWAY 40POS [CLKO]	Digi-Key	$2.55	$2.55
P1<>P17 (Crimp)	8	SXH-001T-P0.6	CONN Terminal Crimp XH 22-28 AWG	Digi-Key	$0.06	$0.48
R1<>R5	5	MFR-25FBF5210K	RES Metal Film 10kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.50
R6, R8, R9, R12, R13, R32<>R40	14	RNMF14FTC120R	RES Metal Film 120Ω 1/4W ±1% AXIAL	Digi-Key	$0.14	$1.96
R7	1	RNMF14FTC4K70	RES Metal Film 4.7kΩ 1/4W ±1% AXIAL	Digi-Key	$0.14	$0.14
R10, R11	2	3362P-1-103LF	TRIMMER 10kΩ 0.5W ±10% PC PIN	Digi-Key	$0.98	$1.96
R14, R16, R22, R23	4	MFR-25FBF5210R	RES Metal Film 10Ω 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.40
R15, R17<>R21, R24<>R31	14	MFR-25FBF52100R	RES Metal Film 100Ω 1/4W ±1% AXIAL	Digi-Key	$0.10	$1.40
R41, R42	2	MFR-25FBF5220K	RES Metal Film 20kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.20
R43, R44	2	MFR-25FBF52499R	RES Metal Film 499Ω 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.20
R45<>R50	6	MFR-25FBF521K	RES Metal Film 1kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.60
R51	1	MFR-25FBF52165R	RES Metal Film 165Ω 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.10
S1, S2	2	B3F-1000	SWITCH TACTILE SPST-NO 0.05A 24V	Digi-Key	$0.35	$0.70
				Grand Total:		$97.94

Table E-2. Bill of materials for the power management circuit board.

Bill of Materials: Power Management Circuit Board						
Reference	**QTY**	**Part Number**	**Description**	**Supplier**	**Unit Pricing**	**Total Cost**
POWER PCB	1	Power PCB	Power Management PCB	OSH Park	$10.13	$10.13
C1	1	TVA1311.5	CAP ALUM 200µF 50V ±20% AXIAL	Mouser	$3.80	$3.80
C2	1	TVA1209	CAP ALUM 500µF 25V ±20% AXIAL	Mouser	$4.51	$4.51
C3	1	C317C103K5R5TA	CAP CER 0.01µF 50V ±10% RADIAL	Digi-Key	$0.24	$0.24
C4, C5	2	32SEPF68M	CAP ALUM 68µF 32V 25mΩ ±20%	Digi-Key	$1.23	$2.46
D1	1	BZX84-A18,215	DIODE ZENER 18V 1/4W ±1%	Mouser	$0.41	$0.41
D2	1	MBR340G	DIODE Schottky Barrier Rectifier 40V 3A	Digi-Key	$0.55	$0.55
IC1	1	LT1076CT#PBF	IC LDO Volt. Regulator Adjustable Output	Digi-Key	$6.78	$6.78
IC2	1	LT1529CT-3.3#PBF	IC LDO Volt. Regulator 3.3V 3A	Digi-Key	$6.78	$6.78
IC3	1	LT1529CT-5#PBF	IC LDO Volt. Regulator 5V 3A	Digi-Key	$6.78	$6.78
IC4	1	OPA2137PA	High Power Supply Dual Op Amp	Digi-Key	$2.23	$2.23
L1	1	PE-92102NL	INDUCTOR Power Toroid 100µH	Mouser	$2.46	$2.46
LED1<>LED3	3	SLR-56MG3F	LED GREEN DIFFUSED Vf = 2.1V, i = 10mA	Digi-Key	$0.53	$1.59
P1 (Male)	1	B2B-XH-A (LF)(SN)(P)	CONN HEADER JST Shrouded [Battery]	Digi-Key	$0.14	$0.14
P1 (Female)	1	XHP-2	CONN HOUSING JST Shrouded [Battery]	Digi-Key	$0.09	$0.09
P2 (Male)	1	B6B-XH-A(LF)(SN)	CONN HEADER JST Shrouded [MasterPIC]	Digi-Key	$0.26	$0.26
P2 (Female)	1	XHP-6	CONN HOUSING JST Shrouded [MasterPIC]	Digi-Key	$0.14	$0.14
P1, P2 (Crimp)	8	SXH-001T-P0.6	CONN Terminal Crimp XH 22-28 AWG	Digi-Key	$0.06	$0.48
R1	1	RNMF14FTC2K70	RES Metal Film 2.7kΩ 1/4W ±1% AXIAL	Digi-Key	$0.14	$0.14
R2	1	3362P-1-103LF	TRIMMER 10kΩ 0.5W ±10% PC PIN	Digi-Key	$0.98	$0.98
R3<>R6	4	MFR-25FBF5215K	RES Metal Film 15kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.40
R7	1	RNMF14FTC24K0	RES Metal Film 24kΩ 1/4W ±1% AXIAL	Digi-Key	$0.14	$0.14
R8	1	MFR-25FBF5220K	RES Metal Film 20kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.10
R9	1	SFR16S0002371FR500	RES Metal Film 2.37kΩ 1/2W ±1% AXIAL	Digi-Key	$0.29	$0.29
R10	1	RNMF14FTC330R	RES Metal Film 330Ω 1/4W ±1% AXIAL	Digi-Key	$0.14	$0.14
R11	1	RNMF14FTC120R	RES Metal Film 120Ω 1/4W ±1% AXIAL	Digi-Key	$0.14	$0.14
R12	1	RNF18FTD3K83	RES Metal Film 3.83kΩ 1/8W ±1% AXIAL	Digi-Key	$0.15	$0.15
R13	1	MFR-25FBF522K21	RES Metal Film 2.21kΩ 1/4W ±1% AXIAL	Digi-Key	$0.10	$0.10
				Grand Total:		$52.41

Table E-3. Bill of materials for the hardware, electronic components, and other electrical accessories.

Bill of Materials: Hardware, Electronic Components, and Other Electrical Accessories						
Reference	**QTY**	**Part Number**	**Description**	**Supplier**	**Unit Pricing**	**Total Cost**
BLDC MOTOR	2	HT03002-X0X	Brushless DC Motor	Barrett	N/A	N/A
PUCK	2	Puck	Servo Electronics Module	Barrett	N/A	N/A
IMU	1	ADIS16300AMLZ	Four Degrees of Freedom Inertial Sensor	Digi-Key	$116.99	$116.99
LCD	1	LCD-09351	Serial Graphic LCD 128x64	sparkfun	$36.95	$36.95
PIC2USB CABLE	1	TTL-232R-3V3	CABLE USB EMBD UART 3.3V 0.1"HDR	Digi-Key	$20.00	$20.00
TQ201-1k	1	TQ201-1k	Reaction Torque Load Cell 1000 in-lb	Omega	$795.00	$795.00
TQ202-250	1	TQ202-250	Reaction Torque Load Cell 250 in-lb	Omega	$795.00	$795.00
LIPO BATTERY	2	TP2700-4SPP25	2700 mAh 4 Cell 14.8V G8 Pro Lite+ 25C LiPo	TPRC	$78.99	$157.98
FSR1<>FSR6	1	30-61710	SENSOR FSR408 24" STRIP W/TAB	Digi-Key	$14.70	$14.70
LIPO MONITOR	2	LIPO VOLTAGE MONITOR	1-8s LiPo Battery Low Voltage Buzzer Alarm	Amazon	$4.97	$9.94
WIRE HARNESS	1	EC3 WIRE HARNESS	EC3 Battery Series Wire Harness	amain	$8.25	$8.25
EC3 CONN	3	EC3 CONNECTOR M/F	EC3 Battery Connector 1xMale/1xFemale	amain	$3.69	$11.07
FUSE1	1	0498040.M	FUSE 40A 32VDC MIDI BOLT-ON SLOW	Digi-Key	$2.52	$2.52
FUSE1HOLDER	1	04980900ZXT	FUSE BLOCK MIDI 200A	Digi-Key	$13.31	$13.31
FUSE2	1	021702.5HXP	FUSE 250V IEC FA 5X20 2.5A	Digi-Key	$0.35	$0.35
FUSE2HOLDER	1	FX0385	FUSEHOLDER 5X20 IN LINE SEALED	Digi-Key	$5.55	$5.55
SWITCH	1	S821	SWITCH TOGGLE DPST 30A 125V	Digi-Key	$22.35	$22.35
DIP-24 ADAPTER	1	DR127D254P24F	2x12 1.27mm F.Header to DIP-24 Adapter	ProtoAdv	$10.49	$10.49
CBL ASM 24POS	1	FFSD-12-D-05.00-01-N	5"L 2x12 Double Ended Cable Assembly	Newark	$12.58	$12.58
R/A SHRD HDR	1	M50-4801245	CONN HEADER SHRD 1.27mm R/A 24POS	Digi-Key	$3.25	$3.25
V/A USHRD HDR	1	M50-3501242	CONN HEADER USHRD 1.27mm VERT 24POS	Digi-Key	$1.76	$1.76
R/A USHRD HDR	1	M50-3901242	CONN HEADER USHRD 1.27mm R/A 24POS	Digi-Key	$2.57	$2.57
V/A BOX HDR	1	3220-24-0200-00	BOX HEADER 1.27mm R/A 24 POS	Digi-Key	$1.29	$1.29
HSNG 12POS	2	WM5341-ND	CONN HOUSING 12POS 0.100" MOLEX	Digi-Key	$0.94	$1.88
HSNG 2POS	2	WM2800-ND	CONN HOUSING 2POS 0.100" MOLEX	Digi-Key	$0.47	$0.94
HEATSINK1 TO220	3	V5629G	HEATSINK ALUM Black Anodized Bolt-on	Digi-Key	$0.94	$2.82
HEATSINK2 TO220	3	581002B00000	HEATSINK TO-220 4.3W H=1.0 BLK	Digi-Key	$1.37	$4.11
MOUNTING KIT	3	4880G	MOUTING KIT for HS300-ND Heatsink	Digi-Key	$2.52	$7.56
TERMINAL KIT	1	76650-0040	RING/SPADE/BUTT TERMINAL KIT + TOOL	Digi-Key	$19.24	$19.24
POWER WIRE, Vcc	1	7587k951	Stranded Single-Cond. Wire, Black 18A	McMaster	$4.32	$4.32
POWER WIRE, Vss	1	7587k952	Stranded Single-Cond. Wire, Red 18A	McMaster	$4.32	$4.32
STANDOFFS	1	Standoff and Screw Kit	M3 Standoff and Phillips Head Screw Kit	Amazon	$8.87	$8.87
				Grand Total:		$2,095.96

APPENDIX F

FIRMWARE AND SOFTWARE

F-1: main.c

```
/*****************************************************************************************************************
Name: main.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Main file for TWIP control algorithm.
*****************************************************************************************************************/

#include <p32xxxx.h>            // Include p32MX795F512H header file
#include <plib.h>               // Include peripheral library master header file
#include <sys/kmem.h>           // Include kernel memory allocator
#include <peripheral/system.h>  // Include system configuration functions
#include <GenericTypeDefs.h>    // Include generic type definitions
#include "math.h"               // Include math library
#include "support.h"            // Include variable declarations and function prototypes

// Configuration bits for clock initialization parameters
#pragma config FNOSC = FRCPLL       // Enable Internal Fast RC Oscillator (8 MHz) w/ PLL
#pragma config FSOSCEN = OFF        // Disable Secondary Oscillator
#pragma config OSCIOFNC = ON        // Enable CLKO Output on the OSCO Pin
#pragma config FCKSM = CSECMD       // Enable Clock Switching & Fail-Safe Clock Monitor (FSCM)
#pragma config FWDTEN = OFF         // Disable Watchdog Timer (WDT)
#pragma config ICESEL = ICS_PGx1    // Set ICE/ICD Communication Channel to PGC1/PGD1

// Global variable definitions
unsigned char CANmsgReceived = FALSE;   // TRUE if CAN1 FIFO1 received a message (updated in CAN1ISR)
unsigned char resetEncoderCount = FALSE;// TRUE if user presses Button 2 to reset Puck (see Puck.c)
unsigned char IMUinitialized = FALSE;   // TRUE if get_IMUgyro has executed at least once (see IMU.c)
unsigned char PuckLWinitialized = FALSE;// TRUE if posnDataLW[] contains n-points (see Puck.c)
unsigned char PuckRWinitialized = FALSE;// TRUE if posnDataRW[] contains n-points (see Puck.c)
unsigned int run_time = 0;              // 1 msec increments, resets at 2^16 msec
unsigned int sysCLK = 80000000;         // Clock frequency is set to 80 MHz
unsigned int wait_flag = 0;             // Signals the end of 1 millisecond sample period
long jointPosition[2];                  // Puck.c: used in parseCANmsgPuck()

// Assign enough memory for 2 channels, each with 2 message buffers
unsigned int CANmsgBuffersFIFO[16];     // Assign buffer area to be used by the CAN module

// Structure Definitions
CANRxMessageBuffer rxCANmsg;    // Structure used to store received CAN messages inside RAM
PUCKmsgID PUCKdataSI[2];        // Structure used to store processed Puck messages inside RAM in SI units
PUCKconvFactorsID PUCKconvFactors[2];   // Structure used to store Puck conversion factors
IMUmsgID IMUdataSI;             // Structure used to store processed IMU messages inside RAM in SI units

int main()
{
/*=============================================================================================================*
Prototype:  unsigned int SYSTEMConfigPerformance(unsigned int sys_clock)
Arguments:  sys_clock   system clock in Hz
Descrptn.:  This macro sets flash wait states, PBCLK divider and DRM wait states based on the specified
            lock frequency. It also turns on the cache mode if avaialble. Based on the current frequency,
            the PBCLK divider will be set at 1:2. This knowledge is required to correctly set UART baud
            rate, timer reload value and other time sensitive setting.

Prototype:  void INTConfigureSystem(INT_SYSTEM_CONFIG config)
Arguments:  config      The interrupt configuration to set.
Descrptn.:  This routine configures the core to receive interrupt requests and configures the Interrupt
            module for Multi-vectored or Single Vectored mode.

Prototype:  unsigned int INTEnableInterrupts();
Descrptn.:  This routine enables the core to handle any pending interrupt requests.
Returns:    The previous state of the CP0 register Status.IE bit.
 *=============================================================================================================*/
    // Configure oscillator clock source
    config_clock();

    // Configure the device for maximum performance at 80 MHz
    SYSTEMConfigPerformance(sysCLK);

    // Configure the system for single vectored interrupts
    INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);

    // Enable the core to handle any pending interrupt requests
    INTEnableInterrupts();
```

```
    // Standard Initialization Routine
    config_pins();          // Configure pins
    init_samptime();        // Initialize sample time (1ms)
    init_Timer4();          // Initialize Timer4/5 as a 32-bit counter

    // CAN FIFO Initialization Routine
    init_CAN1();            // Initialize CAN1 FIFO (used by Pucks)

    // SPI Initialization Routine
    init_SPI3master();      // Initialize SPI3 Master Mode (used by ADC1, ADC2)
    init_SPI4master();      // Initialize SPI4 Master Mode (used by IMU)

    // Motor Driver (Puck) Initialization Routine
    config_pucks(MODE_TORQUE);      // Wake up pucks and configure them to TORQUE mode
    getConversionFactorsPuck(L_WHEEL_ID);
    getConversionFactorsPuck(R_WHEEL_ID);
    //setDefaultProperty();         // Set default puck properties
    //setSafetyLimits();

    // IMU Initialization Routine
    config_IMU();           // Configure IMU's sampling rate, gyroscope, and Bartlett filter

    // UART Initialization Routine
    init_UART3();           // Initialize UART3 (used by LCD, MATLAB)

    // LCD Initialization Routine
    config_LCD();

    // ADC Initialization Routine
    init_MasterClockADC(4000000,50);    // Configure OC1 to output a 4 MHz signal
    config_ADC();

    // Improper Device Configuration Check
    improperConfigCheck();

    // Loop infinitely
    while (1)
    {
        // Execute LQR algorithm
        LQR();

        // If Button 2 is pressed, reset current and cumulative encoder counters of each Puck
        if(readButton2() == 1)  {resetPucks();}

        while(wait_flag == 0) {};       // Wait for end of sample time
        wait_flag = 0;                  // Reset wait flag
    } // end while(1)

    return 0;
} // end main
```

F-2: ADC.c

```
/*****************************************************************************************************************
Name: ADC.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used for ADC (MCP3903) operation through SPI3.
    Relevant Hardware:
        MCP3903     six channel delta sigma analog-to-digital converter.
        AD8220      instrumental amplifier used to amplify the pitchy/yaw LC outputs by a gain of 100.
        TQ201-1k    load cell used to measure torque about the pitch axis of the vehicle.
        TQ202-250   load cell used to measure torque about the yaw axis of the vehicle.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include "support.h"

// The following definitions are used in multiple functions throughout this file
static short ADC_OSR = OSR_256;         // Set Delta Sigma ADC oversampling ratio to 256
static short ADC_WIDTH = WIDTH_24bit;   // Set CH0-CH5 output data word width to 24-bit mode

void config_ADC(void)
{
    int PHASEbits = 0;
    int GAINbits = 0;
    int COMbits = 0;
    int CONFIGbits = 0;

    LATDbits.LATD4 = 1;         // Raise Master Reset Logic to allow communication w/ all SPI3 slaves

    delay_ms(10);               // Wait for the ADC module to start-up before initializing communication

    // Configure phase delay register
    PHASEbits |= ADC_PHASEC(0b00000000);        // Set CH4 relative to CH5 phase delay to 0
    PHASEbits |= ADC_PHASEB(0b00000000);        // Set CH2 relative to CH3 phase delay to 0
    PHASEbits |= ADC_PHASEA(0b00000000);        // Set CH0 relative to CH1 phase delay to 0
    setPropertyADC(ADC1_ID,1,ADC_PHASE,PHASEbits);
    setPropertyADC(ADC2_ID,1,ADC_PHASE,PHASEbits);
    delay_ms(50);

    // Configure Programmable Gain Amplifier (PGA) gain register
    GAINbits |= PGA_CHN(PGA_GAIN_1,0);          // Set CH0 PGA gain to 1
    GAINbits |= PGA_CHN(PGA_GAIN_1,1);          // Set CH1 PGA gain to 1
    GAINbits |= PGA_CHN(PGA_GAIN_1,2);          // Set CH2 PGA gain to 1
    GAINbits |= PGA_CHN(PGA_GAIN_1,3);          // Set CH3 PGA gain to 1
    GAINbits |= PGA_CHN(PGA_GAIN_1,4);          // Set CH4 PGA gain to 1
    GAINbits |= PGA_CHN(PGA_GAIN_1,5);          // Set CH5 PGA gain to 1
    GAINbits |= BOOST_CH(BOOST_NORM,0);         // Set CH0's current scaling to NORMAL
    GAINbits |= BOOST_CH(BOOST_NORM,1);         // Set CH1's current scaling to NORMAL
    GAINbits |= BOOST_CH(BOOST_NORM,2);         // Set CH2's current scaling to NORMAL
    GAINbits |= BOOST_CH(BOOST_NORM,3);         // Set CH3's current scaling to NORMAL
    GAINbits |= BOOST_CH(BOOST_NORM,4);         // Set CH4's current scaling to NORMAL
    GAINbits |= BOOST_CH(BOOST_NORM,5);         // Set CH5's current scaling to NORMAL
    setPropertyADC(ADC1_ID,1,ADC_GAIN,GAINbits);
    setPropertyADC(ADC2_ID,1,ADC_GAIN,GAINbits);
    delay_ms(50);

    // Configure status/communication register
    COMbits |= COM_READ(0b00);              // Address not incremented, continually read single register
    COMbits |= COM_WMODE(1);                // Static addressing Write Mode
    COMbits |= COM_WIDTH_CHn(ADC_WIDTH);    // Set CH0-CH5 output data word width to ADC_WIDTH
    COMbits |= COM_DR_LTY(1);               // Data ready pulses after 3 DRCCLK periods (DEFAULT)
    COMbits |= COM_DR_HIZ(0);               // Pin state is logic LOW when data is NOT ready (DEFAULT)
    COMbits |= COM_DR_LINK(0);              // Data ready link turned OFF (DEFAULT)
    COMbits |= COM_DRC_MODE(0b00);          // Data ready pulses are output on DRC pin (DEFAULT)
    COMbits |= COM_DRB_MODE(0b00);          // Data ready pulses are output on DRB pin (DEFAULT)
    COMbits |= COM_DRA_MODE(0b00);          // Data ready pulses are output on DRA pin  (DEFAULT)
    setPropertyADC(ADC1_ID,1,ADC_COM,COMbits);
    setPropertyADC(ADC2_ID,1,ADC_COM,COMbits);
    delay_ms(50);

    // Configure MCP3903's configuration register
    CONFIGbits |= CONFIG_RESET_CHn(0b000000);       // Set CH0-CH5 reset mode to OFF (DEFAULT)
    CONFIGbits |= CONFIG_SHUTDOWN_CHn(0b000000);    // Set CH0-CH5 shutdown mode to OFF (DEFAULT)
    CONFIGbits |= CONFIG_DITHER_CHn(0b111111);      // Set CH0-CH5 dithering circuit to ON (DEFAULT)
    CONFIGbits |= CONFIG_OSR(ADC_OSR);              // Set oversampling ratio to ADC_OSR
    CONFIGbits |= CONFIG_PRESCALE(0b00);            // Set master clock prescaler to 1:1 (DEFAULT)
    CONFIGbits |= CONFIG_EXTVREF(0);                // Enable internal voltage reference (DEFAULT)
    CONFIGbits |= CONFIG_EXTCLK(1);                 // Master Clock (MCLK) generated by PIC32's OC1 pin
    setPropertyADC(ADC1_ID,1,ADC_CONFIG,CONFIGbits);
    setPropertyADC(ADC2_ID,1,ADC_CONFIG,CONFIGbits);
    delay_ms(50);
} // end config_ADC

void init_MasterClockADC(int PWMfrequency, char dutyCycle)
{
```

```
    short PWMperiod = 0;
    short PWMdutyCycle = 0;

    PWMperiod = sysCLK/PWMfrequency;
    PWMdutyCycle = PWMperiod * ((float) dutyCycle/100);

    T3CONbits.ON = 0;           // Disable timer
    T3CONbits.TCKPS = 0;        // Timer3 input clock prescaler set to 1:1
    TMR3 = 0;                   // Clear timer register
    PR3 = PWMperiod - 1;        // Set PWM period; PWM Period = (PRy + 1)

    OC1CONbits.ON = 0;          // Disable Output Compare 1 module
    OC1CONbits.OC32 = 0;        // OCxR<15:0> and OCxRS<15:0> are used for comparisons to 16-bit timer
    OC1CONbits.OCTSEL = 1;      // Timer3 is the clock source for this Output Compare module
    OC1CONbits.OCM = 0b110;     // Configure OC1 to PWM mode with Fault pin disabled
    OC1R = PWMdutyCycle;        // Initialize primary Compare register
    OC1RS = PWMdutyCycle;       // Set PWM duty cycle

    T3CONbits.ON = 1;           // Enable 16-bit Timer3
    OC1CONbits.ON = 1;          // Enable Output Compare 1 module
} // end init_MasterClockADC

/*=============================================================================================================*
    setPropertyADC() writes a control byte followed by data bytes to a register inside the ADC.

    Arguments:
        device      specifies which ADC microchip will be used, ADC1_ID or ADC2_ID.
        dev_addr    device address bits <7:6> (default device address bits are 01).
        property    register address bits <5:1>.
        value       24-bit data that will be loaded into ADC address location specified by 'property'.

    Description:
        This function writes an 8-bit wide control byte followed by 24-bit wide data bytes to a register
        inside the MCP3903.

        The 'Write' command bit is specified by setting the LSB in the control byte to 0 (0<<0).
 *=============================================================================================================*/
void setPropertyADC(char device, char dev_addr, char property, int value)
{
    char tx_dataC = 0;                  // Control byte
    char tx_dataH = 0;                  // High byte of data value to be transmitted
    char tx_dataM = 0;                  // Middle byte of data value to be transmitted
    char tx_dataL = 0;                  // Low byte of data value to be transmitted

    // Inserting device address, target register address, and 'write' command bit into a 8-bit sequence
    tx_dataC = (dev_addr << 6) | (property << 1) | (0<<0);  // Control byte

    // Distributing 24-bit data 'value' into three bytes
    tx_dataH = (value & 0x00FF0000) >> 16;
    tx_dataM = (value & 0x0000FF00) >> 8;
    tx_dataL = (value & 0x000000FF);

    // Lower ADC1 or ADC2 slave select line to initiate data exchange
    if(device == ADC1_ID) {LATDbits.LATD11 = 0;}    // Lower ADC1 CS line
    if(device == ADC2_ID) {LATDbits.LATD9 = 0;}     // Lower ADC2 CS line
    //delay_us(1);                      // CS setup time (Minimum: 50 ns)

    // Send first SPI message carrying control byte to ADC register specified by 'property'
    sendSPImsg(device, tx_dataC);

    // Send 24-bit wide data value to MCP3903
    sendSPImsg(device, tx_dataH);
    sendSPImsg(device, tx_dataM);
    sendSPImsg(device, tx_dataL);

    //delay_us(1);                      // CS hold time (Minimum: 100 ns)
    // Raise ADC1 or ADC2 slave select line to terminate data exchange
    if(device == ADC1_ID) {LATDbits.LATD11 = 1;}    // Raise ADC1 CS line
    if(device == ADC2_ID) {LATDbits.LATD9 = 1;}     // Raise ADC2 CS line
    //delay_us(1);                      // CS disable time (Minimum: 50 ns)
} // end setPropertyADC


/*=============================================================================================================*
    getPropertyADC() writes a control byte to a register inside the ADC and reads returned message.

    Description:
        This function sends an 8-bit wide control byte to one of the ADC microchips requesting data, stores
        the returned 24-bit value, and converts the returned data into a 32-bit signed integer for further
        processing.

        At 200 Hz sampling rate, the PIC32 is unable execute fast enough to require the need to implement
        delays during SPI transmission to satisfy MCP3903's minimum CS setup time, CS hold time, and CS
        disable time. Thus, these delays have been commented.
        The 'Read' command bit is specified by setting the LSB in the control byte to 1 (1<<0).
 *=============================================================================================================*/
void getPropertyADC(char device, char dev_addr, char property, int *reply)
{
```

```
    char tx_dataC = 0;                  // Control byte
    char rx_dataH = 0;                  // High byte of data value received from MCP3903
    char rx_dataM = 0;                  // Medium byte of data value received from MCP3903
    char rx_dataL = 0;                  // Low byte of data value received from MCP3903

    // Inserting device address, target register address, and 'read' command bit into a 8-bit sequence
    tx_dataC = (dev_addr << 6) | (property << 1) | (1<<0);  // Control byte

    // Lower ADC1 or ADC2 slave select line to initiate data exchange
    if(device == ADC1_ID) {LATDbits.LATD11 = 0;}    // Lower ADC1 CS line
    if(device == ADC2_ID) {LATDbits.LATD9 = 0;}     // Lower ADC2 CS line
    //delay_us(1);                      // CS setup time (Minimum: 50 ns)

    // Send first SPI message carrying control byte to ADC register specified by 'property'
    sendSPImsg(device, tx_dataC);

    // Read 24-bit wide ADC data received from MCP3903
    rx_dataH = readSPImsg(device);
    rx_dataM = readSPImsg(device);
    rx_dataL = readSPImsg(device);

    //delay_us(1);                      // CS hold time (Minimum: 100 ns)
    // Raise ADC1 or ADC2 slave select line to terminate data exchange
    if(device == ADC1_ID) {LATDbits.LATD11 = 1;}    // Raise ADC1 CS line
    if(device == ADC2_ID) {LATDbits.LATD9 = 1;}     // Raise ADC2 CS line
    //delay_us(1);                      // CS disable time (Minimum: 50 ns)

    // Store 24-bit wide ADC data inside 32-bit pointer
    *reply = 0;
    *reply |= ( (int)rx_dataH << 16) & 0x00FF0000;
    *reply |= ( (int)rx_dataM << 8 ) & 0x0000FF00;
    *reply |= ( (int)rx_dataL << 0 ) & 0x000000FF;

    // The contents of the received message must now be extracted and organized into usable form
    parseSPImsgADC(property, reply);    // This function will return processed data packet inside 'reply'
} // end getPropertyADC

void parseSPImsgADC(char property, int *rx_value)
{
    *rx_value &= 0x00FFFFFF;        // Drop highest byte; ADC data is limited to 24-bits wide

    switch(property)
    {
        case(ADC_CH0): case(ADC_CH1): case(ADC_CH2): case(ADC_CH3): case(ADC_CH4): case(ADC_CH5):
            if(ADC_WIDTH == WIDTH_24bit && ADC_OSR == OSR_256)
            { // At OSR_256, ADC output has a 24-bit resolution
                *rx_value = convert_24bit_to_32bit_signed(rx_value);    // 24-bit to 32-bit signed
            }
            if(ADC_WIDTH == WIDTH_24bit && ADC_OSR == OSR_128)
            { // At OSR_128, data has 23-bit resolution with 1 zero at the end to fill 24-bit WIDTH
                *rx_value = *rx_value >> 1;     // Shift right by 1 to eliminate redudant zero
                *rx_value = convert_23bit_to_32bit_signed(rx_value);    // 23-bit to 32-bit signed
            }
            if(ADC_WIDTH == WIDTH_24bit && ADC_OSR == OSR_64)
            { // At OSR_64, data has 20-bit resolution with 4 zeros at the end to fill 24-bit WIDTH
                *rx_value = *rx_value >> 4;     // Shift right by 4 to eliminate redudant zeros
                *rx_value = convert_20bit_to_32bit_signed(rx_value);    // 20-bit to 32-bit signed
            }
            if(ADC_WIDTH == WIDTH_24bit && ADC_OSR == OSR_32)
            { // At OSR_32, data has 17-bit resolution with 7 zeros at the end to fill 24-bit WIDTH
                *rx_value = *rx_value >> 7;     // Shift right by 7 to eliminate redudant zeros
                *rx_value = convert_17bit_to_32bit_signed(rx_value);    // 17-bit to 32-bit signed
            }
            // If ADC_WIDTH == WIDTH_16bit is used then additional code needs to be added
            break;
        default:
            printString("Invalid ADC property.");
            break;
    }
} // end parseSPImsgADC

int convert_24bit_to_32bit_signed(int *value)
{
    if(*value & 0x00800000)
    { // Check if bit 23 is 1
        *value = *value | 0xFF000000;   // Set highest 8 bits to 1
        return(*value);
    }
    else
    { // If bit 23 is 0 then the 24-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_24bit_to_32bit_signed

int convert_23bit_to_32bit_signed(int *value)
{
    if(*value & 0x00400000)
    { // Check if bit 22 is 1
```

```
        *value = *value | 0xFF800000;   // Set highest 9 bits to 1
        return(*value);
    }
    else
    { // If bit 22 is 0 then the 23-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_23bit_to_32bit_signed

int convert_20bit_to_32bit_signed(int *value)
{
    if(*value & 0x00080000)
    { // Check if bit 19 is 1
        *value = *value | 0xFFF00000;   // Set highest 12 bits to 1
        return(*value);
    }
    else
    { // If bit 19 is 0 then the 20-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_20bit_to_32bit_signed

int convert_17bit_to_32bit_signed(int *value)
{
    if(*value & 0x00010000)
    { // Check if bit 16 is 1
        *value = *value | 0xFFFE0000;   // Set highest 15 bits to 1
        return(*value);
    }
    else
    { // If bit 16 is 0 then the 17-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_17bit_to_32bit_signed

/*=================================================================================================*
    getLoadCellData_pitch() requests pitch load cell voltage measurements from ADC1 CH1, normalizes the
      data, and converts the TQ201-1k's normalized analog voltage output to SI units.

    Description:
        This function collects amplified/filtered voltage measurements from MCP3903 #1's CH1, calculates
        the offset in pitch load cell (LC) voltage measurements and uses it to normalize the data. The
        normalized pitch LC data [in ADC units] is then converted to TQ201-1k's analog voltage output [V].

        The TQ201-1k's analog voltage ouput passes through an instrumental amplifier (AD8220) with a gain
        of 100 prior to entering the MCP3903 #1's non-inverting analog input pin for Channel 1.

        The trimpot on the pitch LC voltage output line was adjusted until the offset voltage entering CH1
        of ADC1 was roughly 0.5 V when no torque was being applied to pitch LC.

        Voltage entering ADC1 CH1:
        CHn+ = [DATA_CHn * 2.35V]/[8,388,608 * PGA_GAIN * 3] = (2.35 / 25,165,824) * DATA_CHn

        Voltage entering AD8220, which is equivalent to load cell voltage output:
        V_outPitchLC = CHn+ / 100 = (0.0235 / 25,165,824) * DATA_CHn

        TQ201-1k conversion factor obtained from LC calibration datasheet: Sensitivity = 2.844 mV/V
        T_pitchLC = [1000 in-lbs / [(2.844 mV/V)*(5V)]] * [1000 mV / 1 V] = [7945.487 Nm/V] * V_outPitchLC

        The conversion factors used in this function assume PGA gain = 1 for all ADC channels.

        IMPORTANT! If the PGA gains in the ADC are modified then the conversion factors below must be
        adjusted accordingly!
 *=================================================================================================*/
float getLoadCellData_pitch(void)
{
    static unsigned char FirstTimeExecuting = TRUE;
    static int LCpitch_offset = 0;
    int LCpitch = 0;            // Load Cell (LC) pitch data
    int LCpitch_norm = 0;
    float LCpitch_scaled = 0;
    float LCpitch_SIunits = 0;

    // Calculate pitch load cell voltage offset [in ADC units] only once per firmware initialization
    if(FirstTimeExecuting == TRUE)
    {
        LCpitch_offset = getLoadCellOffset_pitch();
        FirstTimeExecuting = FALSE;
    }

    // Get amplified/filtered voltage measurements from TQ201-1k (Pitch Load Cell) in ADC1 CH1
    getPropertyADC(ADC1_ID,1,ADC_CH1,&LCpitch);

    // Calculate normalized load cell pitch data value [in ADC units]
    LCpitch_norm = LCpitch - LCpitch_offset;                // [in ADC units]

    // Convert MCP3903 measurements to TQ201-1k analog voltage output [Volts]
    LCpitch_scaled = (0.0235 / 25165824) * LCpitch_norm;    // [in Volts]
```

```
    // Convert TQ201-1k analog voltage output [V] to SI units [N*m] using LC calibration datasheet
    LCpitch_SIunits = 7945.487 * LCpitch_scaled;            // [in N*m]

    return(LCpitch_SIunits);                                // [N*m]
} // end getLoadCellData_pitch

/*=============================================================================================================*
    getLoadCellData_yaw() requests yaw load cell voltage measurements from ADC1 CH0, normalizes the data,
      and converts the TQ202-250's normalized analog voltage output to SI units.

    Description:
        This function operates very similar to getLoadCellData_pitch(). Trimpot tuning and ADC voltage
        calculations are also the same except for ADC to SI unit conversion factor which now uses yaw
        load cell calibration datasheet.

        TQ202-250 conversion factor obtained from LC calibration datasheet: Sensitivity = 2.238 mV/V
        T_yawLC = [250 in-lbs / [(2.238 mV/V)*(5V)]] * [1000 mV / 1 V] = [2524.237 Nm/V] * V_outYawLC
 *=============================================================================================================*/
float getLoadCellData_yaw(void)
{
    static unsigned char FirstTimeExecuting = TRUE;
    static int LCyaw_offset = 0;
    int LCyaw = 0;              // Load Cell (LC) yaw data
    int LCyaw_norm = 0;
    float LCyaw_scaled = 0;
    float LCyaw_SIunits = 0;

    // Calculate yaw load cell voltage offset [in ADC units] only once per firmware initialization
    if(FirstTimeExecuting == TRUE)
    {
        LCyaw_offset = getLoadCellOffset_yaw();
        FirstTimeExecuting = FALSE;
    }

    // Get amplified/filtered voltage measurements from TQ202-250 (Yaw Load Cell) in ADC1 CH0
    getPropertyADC(ADC1_ID,1,ADC_CH0,&LCyaw);

    // Calculate normalized load cell yaw data value [in ADC units]
    LCyaw_norm = LCyaw - LCyaw_offset;                  // [in ADC units]

    // Convert MCP3903 measurements to TQ202-250 analog voltage output [Volts]
    LCyaw_scaled = (0.0235 / 25165824) * LCyaw_norm;    // [in Volts]

    // Convert TQ202-250 analog voltage output [V] to SI units [N*m] using LC calibration datasheet
    LCyaw_SIunits = 2524.237 * LCyaw_scaled;            // [in N*m]

    return(LCyaw_SIunits);                              // [N*m]
} // end getLoadCellData_yaw

/*=============================================================================================================*
    getLoadCellOffset_pitch() gathers n load cell measurements and calculates the average.

    Description:
        This function collects 10 load cell measurements and calculates the average. This average is used
        in getLoadCellData_pitch() calculations to obtain normalized torque measurements from the load
        cell.
 *=============================================================================================================*/
int getLoadCellOffset_pitch(void)
{
    char i = 0, j = 0;
    char n = 10;            // number of points used to calculate voltage offset (User Defined)
    int LCpitchData[11] = {0};
    int LCpitch = 0;
    int LCpitch_sum = 0;
    int LCpitch_offset = 0;

    // Collect n points from the ADC channel leading to TQ201-1k (Pitch Load Cell)
    for (i = 0; i < n; i++)
    {
        // Get amplified/filtered voltage measurements from TQ201-1k (Pitch Load Cell) in ADC1 CH1
        getPropertyADC(ADC1_ID,1,ADC_CH1,&LCpitch);
        LCpitchData[i] = LCpitch;       // Store latest pitch load cell measurement from MCP3903
    }

    // Calculate the sum of the latest n-points obtained from external ADC (MCP3903)
    for (j = 0; j < n; j++) { LCpitch_sum += LCpitchData[j]; }

    // Calculate average of latest n-points to obtain pitch load cell voltage offset in ADC units
    LCpitch_offset = LCpitch_sum/n; // Note: trimming calculated average of this integer is not an issue

    return(LCpitch_offset);         // Returned valued is in MCP3903 units
} // end getLoadCellOffset_pitch

int getLoadCellOffset_yaw(void)
{
    char i = 0, j = 0;
    char n = 10;            // number of points used to calculate voltage offset (User Defined)
```

```
    int LCyawData[11] = {0};
    int LCyaw = 0;
    int LCyaw_sum = 0;
    int LCyaw_offset = 0;

    // Collect n points from the ADC channel leading to TQ202-250 (Yaw Load Cell)
    for (i = 0; i < n; i++)
    {
        // Get amplified/filtered voltage measurements from TQ202-250 (Yaw Load Cell) in ADC1 CH0
        getPropertyADC(ADC1_ID,1,ADC_CH0,&LCyaw);

        LCyawData[i] = LCyaw;           // Store latest yaw load cell measurement from MCP3903
    }

    // Calculate the sum of the latest n-points obtained from external ADC (MCP3903)
    for (j = 0; j < n; j++) { LCyaw_sum += LCyawData[j]; }

    // Calculate average of latest n-points to obtain yaw load cell voltage offset in ADC units
    LCyaw_offset = LCyaw_sum/n;     // Note: trimming calculated average of this integer is not an issue

    return(LCyaw_offset);           // Returned valued is in MCP3903 units
} // end getLoadCellOffset_yaw
```

F-3: CAN.c

```
/*****************************************************************************************************************
Name: CAN.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Function used to configure and initialize:
    CAN1 for transmission/reception with two motor controllers (Puck).

    Also contains driver functions used to send and read CAN messages.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include <plib.h>
#include <sys/kmem.h>
#include "support.h"

void init_CAN1(void)
{
/*===============================================================================================================*
    Step 1: Switch the CAN module ON and switch it to Configuration mode.
            Wait until the switch is complete.
 *===============================================================================================================*/
    C1CONbits.ON = 1;               // Enable CAN1 Module

/* REGISTER 23-1: CiCON: CAN MODULE CONTROL REGISTER [pp. 256] */
    C1CONbits.REQOP = 4;            // Set CAN1 Module to Configuration Mode
    while(C1CONbits.OPMOD != 4);    // Wait until CAN1 Module is in Configuration Mode

/*===============================================================================================================*
    Step 2: Configure the CAN module clock and timing.

    Barrett Technology's Puck data link specifications:
        Bit Rate = 1 Mbps
        Number of Time Quanta (Tq) per bit = 8
        Sampling Point = 75% of Nominal Bit Time
        Sync Jump Width = 1 Tq
        Standard CAN = 11-bit MsgID

        Device Oper. Freq.= 40 MIPS
        F_SYS  =  F_OSC   = 80 MHz
        Oscillator Freq.  = 80 MHz
        CAN Bit Rate      = 1 Mbps
        # of Tq per bit   = 8 Tq
        Propagation Delay = 1 Tq
        Phase Segment 1   = 4 Tq
        Phase Segment 2   = 2 Tq
        Sync Jump Width   = 1 Tq

        Equation 34-2: [pp. 85]
            F_TQ = N×F_BAUD
        Equation 34-3: [pp. 85]
            BRP = CiCFG<BRP> =[F_SYS /(2×N×F_BAUD)]– 1
        Example 34-16: CAN Bit Timing Calculation Example [pp. 85]

    References:
        Microchip's CAN Bit Timing Calculator
        AN1249 - EXAMPLE 3: CLOCK AND TIMING INITIALIZATION CODE {PIC33EP} [pp. 8]
        Section 21. ECAN - EXAMPLE 21-8: CAN BIT TIMING CALCULATION EXAMPLE {PIC33EP} [pp. 61]
        Section 34. CAN - EXAMPLE 34-17: CONFIGURING THE CAN MODULE [p. 86]
 *===============================================================================================================*/


/* REGISTER 23-2: CiCFG: CAN BAUD RATE CONFIGURATION REGISTER [pp. 258] */
    C1CFGbits.SEG2PHTS = 1;     // Phase Segment 2 time is set to be programmable
    C1CFGbits.SEG2PH = 1;       // Phase Segment 2 time is 2 TQ
    C1CFGbits.SEG1PH = 3;       // Phase Segment 1 time is 4 TQ
    C1CFGbits.PRSEG = 0;        // Propagation Segment time is 1 TQ
    C1CFGbits.SAM = 1;          // Bus line is sampled three times at the sample point
    C1CFGbits.SJW = 0;          // Synchronization Jump Width set to 1 TQ
    C1CFGbits.BRP = BRP_VAL;    // Baud Rate Prescale

/*===============================================================================================================*
    Step 3: Assign the buffer area to the CAN module.
            Update the CAN FIFO Base Address register (CiFIFOBA) with the base address of the FIFO.
            This should be the physical start address of Message Buffer 0 of FIFO0.
 *===============================================================================================================*/
/* FIFO Configuration
    FIFO0 (CH0): Transmit (TX) - 2 Message Buffers
    FIFO1 (CH1): Receive (RX) - 2 Message Buffers
    FIFO4 (CH4) through FIFO31 (CH31) - Not used

    Each CAN TX message buffer requires 4 words (16 bytes) of memory
    Each CAN RX message buffer requires 4 words (16 bytes) of memory (full receive message)
    Therefore, a total of 16 (4 message buffers x 4 words per message buffer) words of memory needs to
    be allocated.
```

```
    Notes:
    KVA_TO_PA((unsigned int) address): converts a virtual address into a physical address
*/

/* REGISTER 23-19: CiFIFOBA: CAN MESSAGE BUFFER BASE ADDRESS REGISTER [p. 283] */
    // Initialize C1FIFOBA register with physical address of CAN message Buffer
    C1FIFOBA = KVA_TO_PA(CANmsgBuffersFIFO);    // Define the base address of all message buffers

/*=============================================================================================================*
    Step 4: Specify size of the channel using the FSIZE<4:0> bits (CiFIFOCONn<20:16>).
            Select whether the FIFO is to be a transmit or receive FIFO (TXEN bit (CiFIFOCONn<7:0>)).
            Disable Remote Transmit Request (RTR) and specify the priority of the TX channel.

    References:
        Section 34. CAN - EXAMPLE 34-2: SETTING UP THE CAN MESSAGE FIFO [p. 52]
        PIC32 CAN Peripheral Library [C:...\MPLAB C32 Suite\doc\pic32-lib-help]
 *=============================================================================================================*/
/* Built-in functions and macros used to configure CAN filters and mask:
Prototype:  void CANConfigureChannelForTx(CAN_MODULE module, CAN_CHANNEL channel, UINT channelSize
                                          CAN_TX_RTR rtren, CAN_TXCHANNEL_PRIORITY priority);
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the desired CAN Channel.
            channelSize Size of the channel in messages. This value should be between 1 and 32
            rtren       Enables disables Remote Transmit Request:
                    CAN_TX_RTR_ENABLED - Remote Transmit Request is enabled.
                    CAN_TX_RTR_DISABLED - Remote Transmit Request is disabled.
            priority    Specifies the priority of the TX channel.

Prototype:  void CANConfigureChannelForRx(CAN_MODULE module, CAN_CHANNEL channel,
                                          UINT32 channelSize, CAN_RX_DATA_MODE dataOnly);
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the desired CAN RX channel.
            channelSize Specifies the number of received messages that the channel can buffer
                        before it overflows. This should be a value between 1 and 32.
            dataOnly    Specifies a full CAN message Receive channel or a data-only message channel:
                    CAN_RX_DATA_ONLY - Channel will be a data-only message receive channel.
                    CAN_RX_FULL_RECEIVE - Channel will be a full CAN message receive channel.
*/
    // Configure CAN Channels 0 and 1 for transmission and reception, respectively
    CANConfigureChannelForTx(CAN1,CAN_CHANNEL0,2,CAN_TX_RTR_DISABLED,CAN_LOW_MEDIUM_PRIORITY);
    CANConfigureChannelForRx(CAN1,CAN_CHANNEL1,2,CAN_RX_FULL_RECEIVE);

/*=============================================================================================================*
    Step 5: Configure filters and mask.
            Configure filters to accept SID messages with Puck IDs 0x423, 0x443, and 0x466.
            Configure filter mask 0 to accept ONLY IDs specified by the acceptance filters.
            Messages accepted by filters 0, 1, and 2 should be stored in channel 1 or FIFO1.

    References:
        PIC32 CAN Peripheral Library [C:...\MPLAB C32 Suite\doc\pic32-lib-help]
 *=============================================================================================================*/
/* Built-in functions and macros used to configure CAN filters and mask:
Prototype:  void CANConfigureFilter(CAN_MODULE module, CAN_FILTER filter,
                                    UINT32 id, CAN_ID_TYPE filterType);
Arguments:  module      Identifies the desired CAN module.
            filter      Identifies the desired CAN RX Filter.
            id          A value in the range 0x0 to 0x7FF for SID filter type
                        or 0x0 to 0x1FFFFFFF for EID filter type.
            filterType  Specifies the type of filter:
                    CAN_EID: Filter is an extended ID message filter.
                    CAN_SID: Filter is a standard ID message filter.

Prototype:  void CANConfigureFilterMask (CAN_MODULE module, CAN_FILTER_MASK mask, UINT32 maskbits,
                                         CAN_ID_TYPE idType, CAN_FILTER_MASK_TYPE mide);
Arguments:  module      Identifies the desired CAN module.
            mask        Identifies the desired CAN RX Filter Mask.
            maskbits    A value in the range 0x0 to 0x7FF for SID range
                        or 0x0 to 0x1FFFFFFF for the EID range.
            idType      Specifies the value range of maskbits parameter:
                    CAN_EID: Value range of maskbits parameter is 0x0 (ignore all 29 bits of the
                     incoming message ID) to 0x1FFFFFFF (compare all 29 bits of the incoming message ID)
                    CAN_SID: Value range of maskbits parameter is 0x0 (ignore all 11 bits of the
                     incoming message ID) to 0x7FF (compare all 11 bits of the incoming message ID).
            mide        Specifies ID masking options :
                    CAN_FILTER_MASK_IDE_TYPE: Masking will be performed by Filter Type only.
                     If the filter is set up for SID messages, the mask will decline all
                     incoming EID messages. If the filter is set up for EID messages,
                     the mask will decline all incoming SID messages.
                    CAN_FILTER_MASK_ANY_TYPE: Masking will be performed regardless of the
                     incoming message ID type. The message will be accepted on a Filter and
                     Message SID match or a Filter and Message SID/EID match.

Prototype:  void CANLinkFilterToChannel(CAN_MODULE module, CAN_FILTER filter,
                                        CAN_FILTER_MASK mask, CAN_CHANNEL channel);
Arguments:  module      Identifies the desired CAN module.
            filter      Identifies the desired CAN Filter.
            mask        Identifies the mask to be used with this filter.
```

```
            channel     Identifies the channel to be linked to this filter. If a TX channel is linked,
                        the TX channel should have it's Remote Transmit Request feature enabled.

Prototype:  void CANEnableFilter(CAN_MODULE module, CAN_FILTER filter, BOOL enable);
Arguments:  module      Identifies the desired CAN module.
            filter      Identifies the desired CAN Message Acceptance Filter.
            enable      Enables or disables the filter:
                    TRUE: Enables the filter.
                    FALSE: Disables the filter.
*/
    // Filter Configuration
    CANConfigureFilter(CAN1,CAN_FILTER0,0x420,CAN_SID); // Filter 0: Allow Puck ID 1 to enter Buffer 1
    CANConfigureFilter(CAN1,CAN_FILTER1,0x440,CAN_SID); // Filter 1: Allow Puck ID 2 to enter Buffer 1
    CANConfigureFilter(CAN1,CAN_FILTER2,0x460,CAN_SID); // Filter 2: Allow Puck ID 3 to enter Buffer 1

    // Acpt. Mask set to accept all Puck GroupIDs but ONLY Puck IDs specified by Acpt. Filters
    CANConfigureFilterMask(CAN1,CAN_FILTER_MASK0,0x7F0,CAN_SID,CAN_FILTER_MASK_IDE_TYPE);

    // Link Filters 0, 1, and 2 to CAN Receive FIFO1 (Channel 1)
    CANLinkFilterToChannel(CAN1,CAN_FILTER0,CAN_FILTER_MASK0,CAN_CHANNEL1);
    CANLinkFilterToChannel(CAN1,CAN_FILTER1,CAN_FILTER_MASK0,CAN_CHANNEL1);
    CANLinkFilterToChannel(CAN1,CAN_FILTER2,CAN_FILTER_MASK0,CAN_CHANNEL1);

    // Enable CAN Filters 0, 1, and 2
    CANEnableFilter(CAN1,CAN_FILTER0,TRUE);
    CANEnableFilter(CAN1,CAN_FILTER1,TRUE);
    CANEnableFilter(CAN1,CAN_FILTER2,TRUE);

/*================================================================================================*
    Step 6: Enable interrupt and events.
            Enable the receive channel not empty event (channel event) and the receive
            channel event (module event).
            The interrrupt peripheral library is used to enable the CAN interrupt to the CPU.

    References:
        PIC32 CAN Peripheral Library [C:...\MPLAB C32 Suite\doc\pic32-lib-help]
        PIC32 Interrupt Peripheral Library [C:...\MPLAB C32 Suite\doc\pic32-lib-help]
 *================================================================================================*/
/* Built-in functions and macros used to enable CAN interrupts and events:
Prototype:  void CANEnableChannelEvent(CAN_MODULE module, CAN_CHANNEL channel,
                                       CAN_CHANNEL_EVENT flags, BOOL enable);
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the desired CAN Channel.
            flags       Identifies the CAN channel event(s) to be affected.
                        Several events can be controlled by logically OR'ed combination of events
            enable      Enables or disables the specified event:
                    TRUE: Channel event is enabled.
                    FALSE: Channel event is disabled.

Prototype:  void CANEnableModuleEvent(CAN_MODULE module, CAN_MODULE_EVENT flags, BOOL enable);
Arguments:  module      Identifies the desired CAN module.
            flags       Identifies the CAN module level events to be affected.
                        Several events can be controlled by logically OR'ed combination of events
            enable      Enables or disables the specified event:
                    TRUE: Module event is enabled.
                    FALSE: Module event is disabled.

Prototype:  void INTSetVectorPriority(INT_VECTOR vector, INT_PRIORITY priority);
Arguments:  vector      Interrupt vector.
            priority    Interrupt vector's priority.

Prototype:  void INTSetVectorSubPriority(INT_VECTOR vector, INT_SUB_PRIORITY subPriority);
Arguments:  vector      Interrupt vector.
            subPriority Interrupt vector's sub-priority.

Prototype:  void INTEnable(INT_SOURCE source, INT_EN_DIS enable);
Arguments:  source      Interrupt source.
            enable      Enable state to set.
*/
    // Configure the CAN module to generate a CPU interrupt when the RX buffer is full
    // Enable RX channel event: CAN RX Channel Not Empty Event Mask
    CANEnableChannelEvent(CAN1,CAN_CHANNEL1,CAN_RX_CHANNEL_NOT_EMPTY,TRUE);
    CANEnableModuleEvent(CAN1,CAN_RX_EVENT,TRUE);   // Enable CAN1 RX interrupt

    INTSetVectorPriority(INT_CAN_1_VECTOR,INT_PRIORITY_LEVEL_4);        // Set CAN1 vector IPL to 4
    INTSetVectorSubPriority(INT_CAN_1_VECTOR,INT_SUB_PRIORITY_LEVEL_0); // Set CAN1 vector subIPL to 0
    INTEnable(INT_CAN1,INT_ENABLED);                                    // Enable CAN1 interrupt

/*================================================================================================*
    Step 7: Switch the CAN mode to normal mode.
 *================================================================================================*/
/* REGISTER 23-1: CiCON: CAN MODULE CONTROL REGISTER [pp. 256] */
    C1CONbits.REQOP = 0;            // Set CAN1 Module to Normal Operation Mode
    while(C1CONbits.OPMOD != 0);    // Wait until CAN1 Module is in Normal Operation Mode

} // end init_CAN1

/*================================================================================================*
```

```
    readCANmsg() reads and processes received messages from a message buffer inside RX FIFO.

    Step 1: Use any of the available interrupts to determine if there is a message in the FIFO.
    Step 2: Read the CiFIFOUAn register and process one message (16 bytes) from this address.
    Step 3: Set the UINC bit in (CiFIFOCONn<13>) to update the CiFIFOUAn register.
    Step 4: Perform steps 1 and 2 until the interrupt condition gets cleared.

    References:
        Section 34. CAN - S34.6.3: Accessing Received Messages in the FIFO [p. 54]
        Section 34. CAN - TABLE 34-5: RECEIVE MESSAGE FORMAT AS STORED IN RAM [p. 75]
        REGISTER 23-20: CiFIFOCONn: CAN FIFO CONTROL REGISTER 'n' (n = 0 THROUGH 31) [pg. 285]
 *=============================================================================================*/
/* Built-in functions and structures used to read messages from CAN1 FIFO message buffers:
Structure:  CANRxMessageBuffer
Summary:    Defines the structure of the CAN RX Message Buffer.
Definition:
        typedef union
        {
            struct
            {
                CAN_RX_MSG_SID msgSID;  // This is SID portion of the CAN RX message
                CAN_MSG_EID msgEID;     // This is EID portion of the CAN RX message
                BYTE data[8];           // This is the data payload section of the received message
            };
            BYTE dataOnlyMsgData[8];    // This can be used if the type of CAN RX Channel is Data-Only
            UINT32 messageWord[4];      // This is CAN RX message organized as a set of 32 bit words
        }CANRxMessageBuffer;

        typedef struct
        {
            unsigned SID:11;    // SID of the Received CAN Message
            unsigned FILHIT:5;  // Filter which accepted this message
            unsigned CMSGTS:16; // Time stamp of the received message. Timestamping must be enabled
        }CAN_RX_MSG_SID;

        typedef struct
        {
            unsigned DLC:4;         // Data Length Control. Valid range: 0x0 - 0x8
            unsigned RB0:1;         // Reserved bit. Should be always 0
            unsigned :3;            // UNIMPLEMENTED. Bits set to 0
            unsigned RB1:1;         // Reserved bit. Should be always 0
            unsigned RTR:1;         // Remote Transmit Request bit
            unsigned EID:18;        // CAN TX and RX Extended ID field. Valid range: 0x0 - 0x3FFFF
            unsigned IDE:1;         // Identifier bit. 1 = EID, 0 = SID
            unsigned SRR:1;         // Susbtitute Remote request bit
            unsigned :2;            // UNIMPLEMENTED. Bits set to 0
        }CAN_MSG_EID;
*/
void readCANmsg(void)
{
    CANRxMessageBuffer *message;

    if(CANmsgReceived == FALSE)
    {
        // CAN1 did not receive any messages so exit the function
        // ERROR: Code should NOT enter this IF statement
        printf("ERROR: Interrupt was caused by CAN receive but FIFO Receive Buffer is empty.");
        return;
    }

    // Step 1: CAN1 ISR has triggered and message was received
    // Step 2: Read the CiFIFOUAn register and process one message (16 bytes) from this address.
    message = CANGetRxMessage(CAN1,CAN_CHANNEL1);

    rxCANmsg = *message;    // Store received message inside global CAN message structure

    // Step 3: Set the UINC bit in (CiFIFOCONn<13>) to update the CiFIFOUAn register.
    CANUpdateChannel(CAN1,CAN_CHANNEL1);

} // end readCANmsg

/*=============================================================================================*
    sendCANmsg() loads messages into CAN1 FIFO buffers and initiates transmission

    Step 1: Read the CiFIFOUAn register and store one message (16 bytes) at this address.
    Step 2: Set the UINC bit in (CiFIFOCONn<13>) to update the CiFIFOUAn register.
    Step 3: Set the TXREQ bit in (CiFIFOCONn<3>) to transmit the message.

    References:
        Section 34. CAN - S34.6.2: Loading Messages to be Transmitted into the FIFO [p. 53]
        Section 34. CAN - TABLE 34-2: TRANSMIT MESSAGE BUFFER FORMAT [p. 56]
        REGISTER 23-20: CiFIFOCONn: CAN FIFO CONTROL REGISTER 'n' (n = 0 THROUGH 31) [pg. 285]
 *=============================================================================================*/
/* Built-in functions and structures used to load messages into CAN1 FIFO message buffers:
Prototype:  CANTxMessageBuffer *CANGetTxMessageBuffer(CAN_MODULE module, CAN_CHANNEL channel);
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the desired CAN channel.
Returns:    Returns a CANTxMessageBuffer type pointer to an empty message buffer in the TX channel.
```

```
            Returns NULL if the channel is full and there aren't any empty message buffers.

Structure:  CANTxMessageBuffer
Summary:    Defines the structure of the CAN TX Message Buffer.
Definition:
        typedef union
        {
            struct
            {
                CAN_TX_MSG_SID msgSID;  // This is SID portion of the CAN TX message
                CAN_MSG_EID msgEID;     // This is EID portion of the CAN TX message
                BYTE data[8];           // This is the data portion of the CAN TX message
            };
            UINT32 messageWord[4];  // This is CAN TX message organized as a set of 32 bit words
        } CANTxMessageBuffer;

        typedef struct
        {
            unsigned SID:11;        // CAN TX Message Standard ID. Value range: 0x0 - 0x7FF
            unsigned :21;
        }CAN_TX_MSG_SID;

        typedef struct
        {
            unsigned DLC:4;         // Data Length Control. Valid range: 0x0 - 0x8
            unsigned RB0:1;         // Reserved bit. Should be always 0
            unsigned :3;            // UNIMPLEMENTED. Bits set to 0
            unsigned RB1:1;         // Reserved bit. Should be always 0
            unsigned RTR:1;         // Remote Transmit Request bit
            unsigned EID:18;        // CAN TX and RX Extended ID field. Valid range: 0x0 - 0x3FFFF
            unsigned IDE:1;         // Identifier bit. 1 = EID, 0 = SID
            unsigned SRR:1;         // Susbtitute Remote request bit
            unsigned :2;            // UNIMPLEMENTED. Bits set to 0
        }CAN_MSG_EID;

Prototype:  void CANUpdateChannel(CAN_MODULE module, CAN_CHANNEL channel);
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the CAN channel to be updated.
Descrptn.:  This function updates the CAN Channel internal pointers, letting the CAN module know
            that the message processing is done and message is ready to be transmitted.

Prototype:  void CANFlushTxChannel(CAN_MODULE module, CAN_CHANNEL channel)
Arguments:  module      Identifies the desired CAN module.
            channel     Identifies the desired CAN channel.
Descrptn.:  This routine causes all messages in the specified TX channel to be transmitted.
*/
int sendCANmsg(int channel, int tx_id, int dataLength, unsigned char *tx_data)
{
    unsigned int ide = 0;           // Message will transmit standard CAN identifier (11-bit)
    unsigned int rtr = 0;           // Remote transmission request = 0 -> Normal message

    // Declare a pointer to the message structure in RAM that will store the message
    CANTxMessageBuffer *message;

    // Step 1a: Get the address of the message buffer to write to from the C1FIFOUA0 register
    message = CANGetTxMessageBuffer(CAN1,CAN_CHANNEL0);

    // Check if the returned value is NULL, which means that the channel is full
    if(message != NULL)
    {
        // Clear the message buffer
        message->messageWord[0] = 0;
        message->messageWord[1] = 0;
        message->messageWord[2] = 0;
        message->messageWord[3] = 0;

        // Step 1b: Store one message (16 bytes) at the address of the message buffer (C1FIFOUA0)
        message->msgSID.SID     = tx_id;
        message->msgEID.IDE     = ide;
        message->msgEID.RTR     = rtr;
        message->msgEID.DLC     = dataLength;
        message->data[0]        = tx_data[0];
        message->data[1]        = tx_data[1];
        message->data[2]        = tx_data[2];
        message->data[3]        = tx_data[3];
        message->data[4]        = tx_data[4];
        message->data[5]        = tx_data[5];

        // Step 2: Set the UINC bit in (CiFIFOCONn<13>) to update the CiFIFOUAn register
        CANUpdateChannel(CAN1,CAN_CHANNEL0);

        // Step 3: Set the TXREQ bit in (CiFIFOCONn<3>) to transmit the message
        CANFlushTxChannel(CAN1,CAN_CHANNEL0);
    }
    return(0);
} // end sendCANmsg
```

F-4: IMU.c

```
/*****************************************************************************************************
Name: IMU.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used for IMU (ADIS16300) operation through SPI4.
*****************************************************************************************************/

#include <p32xxxx.h>
#include "support.h"

#define IMU_MODE    SAMPL_PRD_DEFAULT   // IMU_MODE is passed to config_IMU() and getPropertyIMU()

void config_IMU(void)
{
    LATBbits.LATB13 = 1;        // Raise Master Reset Logic to allow communication w/ all SPI4 slaves

    // IMU starts up after it has a valid power supply voltage; start-up time varies with sample rate
    if(IMU_MODE == SAMPL_PRD_DEFAULT){delay_ms(180);}   // Power-on start-up time in normal mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR){delay_ms(245);}    // Power-on start-up time in low power mode

    // Set internal sample rate to maximum (default) setting of 819.2 SPS
    setPropertyIMU(SMPL_PRD, IMU_MODE);
    delay_ms(50);

    // Perform quick automatic calibration of gyroscope offset bias
    setPropertyIMU(GLOB_CMD, GLOB_CMD_AUTO_NULL);
    delay_ms(100);      // All sensor data resets to zero; flash memory updates automatically (100 ms)

    // Set gyroscope sensitivity to +/- 150 deg/sec
    setPropertyIMU(SENS_AVG, SENS_AVG_MEDIUM);
    delay_ms(50);

    // Enable correction for low frequency acceleration influences on gyroscope bias
    //setPropertyIMU(MSC_CTRL, MSC_CTRL_CORRECTGYRO);
    //delay_ms(50);

    // Configure Bartlett filter to to 9-tap setting
    setPropertyIMU(SENS_AVG, SENS_AVG_FILTER_9);
    delay_ms(50);
} // end config_IMU

/*=====================================================================================================*
    setPropertyIMU() writes 2 bytes to a pair of registers inside the IMU.

    Arguments:
        property    the address of the lower of the two registers. Second register is greater by +1.
        value       data value that will be loaded into IMU address location specified by 'property'.

    Description:
        Each 16-bit register has two 7-bit addresses: one for its upper byte and one for its lower byte.
        The 'Write' command bit is specified by setting the MSB to 1 (1<<15).
 *=====================================================================================================*/
short setPropertyIMU(short property, short value)
{
    short tx_dataL = 0;
    short tx_dataH = 0;

    // Inserting 'Write' command bit, target register address, and data contents into a 16-bit sequence
    tx_dataL = (1<<15) | (property << 8) | (value & 0x00FF);    // Data to lower-byte register address
    tx_dataH = (1<<15) | (property + 1 << 8) | (value >> 8);    // Data to upper-byte register address

    // Add delay to satisfy IMU SPI timing specifications
    if(IMU_MODE == SAMPL_PRD_DEFAULT)
    {delay_us(16);}         // Stall period between data transfer is 9us for Normal/Default Mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR)
    {delay_us(100);}        // Stall period between data transfer is 75us for Low Power Mode

    // Send first SPI message carrying data contents for IMU register specified by 'property'
    sendSPImsg(IMU_ID, tx_dataL);   // Message contains data for lower-byte address of an IMU register

    // Add delay to satisfy IMU SPI timing specifications
    if(IMU_MODE == SAMPL_PRD_DEFAULT)
    {delay_us(16);}         // Stall period between data transfer is 9us for Normal/Default Mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR)
    {delay_us(100);}        // Stall period between data transfer is 75us for Low Power Mode

    // Send second SPI message carrying data contents for IMU register specified by 'property + 1'
    sendSPImsg(IMU_ID, tx_dataH);   // Message contains data for upper-byte address of an IMU register

    return(0);
} // end setPropertyIMU

/*=====================================================================================================*
    getPropertyIMU() reads 2 bytes from a 16-bit register inside the IMU.
```

```
    Arguments:
        property    the address of the lower of the two registers. Second register is greater by +1.
        *reply      a pointer to the unprocessed value of the property received from the IMU.

    Description:
        Individual register reads require two 16-bit sequences. The first 16-bit sequence provides the
        read command bit (R/W = 0) and the target register address (7-bit address). The second sequence
        transmits the register contents (2 data bytes) on the DOUT line. For example, if DIN = 0x0A00,
        then the content of XACCL_OUT shifts out on the DOUT line during the next 16-bit sequence.
 *=============================================================================================================*/
short getPropertyIMU(short property, short *reply)
{
    short tx_dataL = 0;

    // Inserting 'Read' command bit, target register address, and data contents into a 16-bit sequence
    tx_dataL = (0<<15) | (property << 8);              // (0<<15) is the 'Read' command bit

    // Add delay to satisfy IMU SPI timing specifications
    if(IMU_MODE == SAMPL_PRD_DEFAULT)
    {delay_us(16);}         // Stall period between data transfer is 9us for Normal/Default Mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR)
    {delay_us(100);}        // Stall period between data transfer is 75us for Low Power Mode

    // Send SPI message requesting specified 'property' data to IMU
    sendSPImsg(IMU_ID, tx_dataL);

    // Add delay to satisfy IMU SPI timing specifications
    if(IMU_MODE == SAMPL_PRD_DEFAULT)
    {delay_us(16);}         // Stall period between data transfer is 9us for Normal/Default Mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR)
    {delay_us(100);}        // Stall period between data transfer is 75us for Low Power Mode

    // Read SPI message received from IMU
    *reply = readSPImsg(IMU_ID);

    // The contents of the received message must now be extracted and organized into usable form
    parseSPImsgIMU(property, reply);    // This function will return processed data packet inside 'reply'

    return(0);
} // end getPropertyIMU

short parseSPImsgIMU(short property, short *rx_value)
{
    *rx_value &= 0x3FFF;            // Drop the flags bits on the 2 MSBs of IMU message frame

    switch(property)
    {
        case SUPPLY_OUT:
            break;
        case GYRO_OUT:
            *rx_value = convert_14bit_to_16bit_signed(rx_value);    // 14-bit to 16-bit signed
            break;
        case XACCL_OUT:
            *rx_value = convert_14bit_to_16bit_signed(rx_value);    // 14-bit to 16-bit signed
            break;
        case YACCL_OUT:
            *rx_value = convert_14bit_to_16bit_signed(rx_value);    // 14-bit to 16-bit signed
            break;
        case ZACCL_OUT:
            *rx_value = convert_14bit_to_16bit_signed(rx_value);    // 14-bit to 16-bit signed
            break;
        case TEMP_OUT:
            *rx_value = convert_12bit_to_16bit_signed(rx_value);    // 12-bit to 16-bit signed
            break;
        case ROLL_OUT:
            *rx_value = convert_13bit_to_16bit_signed(rx_value);    // 13-bit to 16-bit signed
            break;
        case PITCH_OUT:
            *rx_value = convert_13bit_to_16bit_signed(rx_value);    // 13-bit to 16-bit signed
            break;
        default:
            printString("Invalid IMU property.");
            break;
    }
    return(0);
} // end parseSPImsgIMU

float convertUnits_IMU_to_SI(short property, short value)
{
    float value_scaled = 0;

    switch(property)
    {
        case SUPPLY_OUT:
            value_scaled = (float) (value)*0.00242;     // SUPPLY_OUT scale factor: 2.42 mV
            break;
        case GYRO_OUT:
```

```
            value_scaled = (float) (value)*0.05;        // GYRO_OUT scale factor: 0.05 deg/sec
            break;
        case XACCL_OUT:
            value_scaled = (float) (value)*0.0006;      // XACCL_OUT scale factor: 0.0006 g (m/s^2)
            break;
        case YACCL_OUT:
            value_scaled = (float) (value)*0.0006;      // YACCL_OUT scale factor: 0.0006 g (m/s^2)
            break;
        case ZACCL_OUT:
            value_scaled = (float) (value)*0.0006;      // ZACCL_OUT scale factor: 0.0006 g (m/s^2)
            break;
        case TEMP_OUT:
            value_scaled = (float) (value)*0.14;        // TEMP_OUT scale factor: 0.14 degCelsius
            break;
        case ROLL_OUT:
            value_scaled = (float) (value)*0.044;       // ROLL_OUT scale factor: 0.044 deg
            break;
        case PITCH_OUT:
            value_scaled = (float) (value)*0.044;       // PITCH_OUT scale factor: 0.044 deg
            break;
        default:
            LED6_ON;
            printf("Invalid IMU property.");
            break;
    }

    return(value_scaled);
} // end convertUnits_IMU_to_SI

float get_IMUsupply(IMUmsgID *IMUdataSI)
{
    short supply = 0;
    float supply_scaled = 0;

    // Get power supply measurement
    getPropertyIMU(SUPPLY_OUT, &supply);

    // Scale property value obtained from IMU
    supply_scaled = convertUnits_IMU_to_SI(SUPPLY_OUT, supply);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUsupply = supply_scaled;

    return(supply_scaled);
} // end get_IMUsupply

/*=================================================================================================*
    get_IMUgyro() reads gyroscope's yaw rate and filters measurements using a moving average filter.

    Arguments:
        *IMUdataSI  a pointer to the structure used to store processed IMU measurements in SI units.
        n           value used to define the number of points used in the moving average filter.

    Global Variables:
        IMUinitialized  Boolean value used to determine whether the IMU has been initialized.

    Local Variables:
        k               static counter used to track array element.
        gyroData[n]     static array used to hold the n-points that will be averaged by the filter.
        j               counter used to track iteration number.
        gyro            value used to store gyroscope measurements received from the IMU.
        gyro_sum        value holding the calculated sum of n-points.
        gyro_filtered   value used to store filtered gyroscope measurements in IMU units.
        gyro_scaled     value used to store filtered gyroscope measurements in SI units [deg/s].

    Description:
        This function reads the IMU's gyroscope x-axis output, filters the readings using a n-point
        recursive moving average filter, and converts the average value to SI units.

        To prevent platform from jerking during initialization due to lack of points to average, this
        function will collect n-points first before passing the filtered average to LQR().
 *=================================================================================================*/
float get_IMUgyro(IMUmsgID *IMUdataSI, char n)
{
    static char k = 0;
    static short gyroData[11] = {0};        // Array length must be equal to or greater than 'n'
    char j = 0;
    short gyro = 0;
    short gyro_sum = 0;
    short gyro_filtered = 0;
    float gyro_scaled = 0;

    // Recursive moving average filter
    if (IMUinitialized == TRUE)
    { // Execute statement only if this function has executed previously and gyroData[] contains n-points
        // Get x-axis gyroscope output
        getPropertyIMU(GYRO_OUT, &gyro);
```

```
        // Update n-element array, gyroData[], with the latest gyro measurements
        gyroData[k] = gyro;     // gyroData[k] stores the latest gyro values
        k++;                    // Increment k counter
        if (k == n) {k = 0;}    // Reset k counter when number of points stored equals n
    }

    // Before the moving average filter can be properly implemented, n points are needed
    if (IMUinitialized == FALSE)
    { // If this function is being called for the first time, collect n-points from the IMU
        for (k = 0; k < n; k++)
        {
            // Get x-axis gyroscope output
            getPropertyIMU(GYRO_OUT, &gyro);

            gyroData[k] = gyro;     // Store latest gyro measurement
        }
        k = 0;                      // Reset k counter
        IMUinitialized = TRUE;      // Raise flag indicating IMU has been initialized
    }

    // Calculate the sum of the lastest n-points
    for (j = 0; j < n; j++) { gyro_sum = gyro_sum + gyroData[j]; }

    // Calculate average of latest n-points
    gyro_filtered = gyro_sum/n;

    // Scale property value obtained from IMU
    gyro_scaled = convertUnits_IMU_to_SI(GYRO_OUT, gyro_filtered);  // [deg/s]

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUgyro = gyro_scaled;                               // [deg/s]

    return(gyro_scaled);
} // end get_IMUgyro

float get_IMUxaccl(IMUmsgID *IMUdataSI)
{
    short xaccl = 0;
    float xaccl_scaled = 0;

    // Get x-axis accelerometer output
    getPropertyIMU(XACCL_OUT, &xaccl);

    // Scale property value obtained from IMU
    xaccl_scaled = convertUnits_IMU_to_SI(XACCL_OUT, xaccl);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUxaccl = xaccl_scaled;

    return(xaccl_scaled);
} // end get_IMUxaccl

float get_IMUyaccl(IMUmsgID *IMUdataSI)
{
    short yaccl = 0;
    float yaccl_scaled = 0;

    // Get y-axis accelerometer output
    getPropertyIMU(YACCL_OUT, &yaccl);

    // Scale property value obtained from IMU
    yaccl_scaled = convertUnits_IMU_to_SI(YACCL_OUT, yaccl);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUyaccl = yaccl_scaled;

    return(yaccl_scaled);
} // end get_IMUyaccl

float get_IMUzaccl(IMUmsgID *IMUdataSI)
{
    short zaccl = 0;
    float zaccl_scaled = 0;

    // Get z-axis accelerometer output
    getPropertyIMU(ZACCL_OUT, &zaccl);

    // Scale property value obtained from IMU
    zaccl_scaled = convertUnits_IMU_to_SI(ZACCL_OUT, zaccl);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUzaccl = zaccl_scaled;

    return(zaccl_scaled);
} // end get_IMUzaccl

float get_IMUtemp(IMUmsgID *IMUdataSI)
{
```

```
    short temp = 0;
    float temp_scaled = 0;

    // Get x-axis gyroscope temperature measurement
    getPropertyIMU(TEMP_OUT, &temp);

    // Scale property value obtained from IMU
    temp_scaled = convertUnits_IMU_to_SI(TEMP_OUT, temp);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUtemp = temp_scaled;

    return(temp_scaled);
} // end get_IMUtemp

float get_IMUpitch(IMUmsgID *IMUdataSI)
{
    short pitch = 0;
    float pitch_scaled = 0;

    // Get x-axis inclinometer output measurement
    getPropertyIMU(PITCH_OUT, &pitch);

    // Scale property value obtained from IMU
    pitch_scaled = convertUnits_IMU_to_SI(PITCH_OUT, pitch);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUpitch = pitch_scaled;

    return(pitch_scaled);
} // end get_IMUpitch

float get_IMUroll(IMUmsgID *IMUdataSI)
{
    short roll = 0;
    float roll_scaled = 0;

    // Get y-axis inclinometer output measurement
    getPropertyIMU(ROLL_OUT, &roll);

    // Scale property value obtained from IMU
    roll_scaled = convertUnits_IMU_to_SI(ROLL_OUT, roll);

    // Store scaled value in structure containing processed IMU data
    IMUdataSI->IMUroll = roll_scaled;

    return(roll_scaled);
} // end get_IMUroll

short convert_12bit_to_16bit_signed(short *value)
{
    if(*value & 0x0800)
    { // Check if bit 11 is 1
        *value = *value | 0xF000;       // Set bits 12, 13, 14, and 15 to 1
        return(*value);             // Return signed 16-bit integer equivalent of signed 12-bit input
    }
    else
    { // If bit 11 is 0 then the 12-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_12bit_to_16bit_signed

short convert_13bit_to_16bit_signed(short *value)
{
    if(*value & 0x1000)
    { // Check if bit 12 is 1
        *value = *value | 0xE000;       // Set bits 13, 14, and 15 to 1
        return(*value);             // Return signed 16-bit integer equivalent of signed 13-bit input
    }
    else
    { // If bit 12 is 0 then the 13-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_13bit_to_16bit_signed

short convert_14bit_to_16bit_signed(short *value)
{
    if(*value & 0x2000)
    { // Check if bit 13 is 1
        *value = *value | 0xC000;       // Set bits 14 and 15 to 1
        return(*value);             // Return signed 16-bit integer equivalent of signed 14-bit input
    }
    else
    { // If bit 13 is 0 then the 14-bit input is not negative and nothing needs to change
        return(*value);
    }
} // end convert_14bit_to_16bit_signed
```

```
short getIMUdiagnostics(void)
{
    short diagnostics;

    // Perform a memory test (pass/fail criteria is loaded into DIAG_STAT[6] register)
    setPropertyIMU(MSC_CTRL, MSC_CTRL_MEMORYTEST);

    // Perform internal self-test
    setPropertyIMU(MSC_CTRL, MSC_CTRL_INTSELFTEST);

    // Get IMU diagnostics
    getPropertyIMU(DIAG_STAT, &diagnostics);

    if(diagnostics & DIAG_STAT_ZACCL_FAIL)
        printLine("Z-axis accelerometer self-test failure", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_YACCL_FAIL)
        printLine("Y-axis accelerometer self-test failure", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_XACCL_FAIL)
        printLine("X-axis accelerometer self-test failure", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_XGYRO_FAIL)
        printLine("X-axis gyroscope self-test failure", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_ALARM2)
        printLine("Alarm 2 active", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_ALARM1)
        printLine("Alarm 1 active", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_FLASH_CHK)
        printLine("Flash checksum error", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_SELF_TEST)
        printLine("Self test error", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_OVERFLOW)
        printLine("Sensor overrange", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_SPI_FAIL)
        printLine("SPI failure", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_FLASH_UPT)
        printLine("Flash update failed", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_POWER_HIGH)
        printLine("Power supply above 5.25V", LCD_LINE_LENGTH);
    if(diagnostics & DIAG_STAT_POWER_LOW)
        printLine("Power supply below 4.75V", LCD_LINE_LENGTH);

    return(diagnostics);
} // end getIMUdiagnostics

void resetIMU(void)
{
    // Reset IMU software
    setPropertyIMU(GLOB_CMD, GLOB_CMD_SW_RESET);
    if(IMU_MODE == SAMPL_PRD_DEFAULT){delay_ms(55);}    // Reset recovery time in normal mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR){delay_ms(120);}    // Reset recovery time in low power mode
} // end resetIMU

void restoreFactoryCalibration(void)
{
    // Reset all user calibration register and sensor data to 0x0000, and update flash memory
    setPropertyIMU(GLOB_CMD, GLOB_CMD_FAC_CALIB);
    delay_ms(150);                                      // Restore factory default time in normal mode
} // restoreFactoryCalibration

void executeFlashMemoryTest(void)
{
    // Execute flash memory test (pass/fail result is reported in DIAG_STAT)
    setPropertyIMU(MSC_CTRL, MSC_CTRL_MEMORYTEST);
    if(IMU_MODE == SAMPL_PRD_DEFAULT){delay_ms(17);}    // Memory test time in normal mode
    if(IMU_MODE == SAMPL_PRD_LOWPWR){delay_ms(90);}     // Memory test time in low power mode
} // end executeFlashMemoryTest

void executeAutomaticSelfTest(void)
{
    // Execute automatic self-test (pass/fail result is reported in DIAG_STAT)
    setPropertyIMU(MSC_CTRL, MSC_CTRL_INTSELFTEST);
    delay_ms(12);
} // end executeAutomaticSelfTest

void gyroPrecisionCalibration(void)
{
    // Perform gyroscope precision automatic bias null calibration
    setPropertyIMU(GLOB_CMD, GLOB_CMD_P_AUTO_NULL);
    delay_ms(30000);    // Sensor goes offline for 30 seconds while calibrating GYRO_OUT
    delay_ms(50);       // All sensor data resets to zero; flash memory updates automatically
} // end gyroPrecisionCalibration
```

F-5: interrupt_handlers.c

```
/*****************************************************************************************************************
Name: interrupt_handlers.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Interrupt handlers.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include <plib.h>
#include "support.h"

/*===============================================================================================================*
    Built-in functions and macros used to configure interrupt handlers:
        mXXClearIntFlag() - clears the interrupt flag (-IF bit) of the chosen interrupt source.
          Replace XX with corresponding peripheral from TABLE 8-2 [p.87 of PIC32 Peripheral Libraries]

    Notes:
        _TIMER_1_VECTOR: is defined in the device-specific header file or in TABLE 7-1 [p. 132]
        IPL7: sets the interrupt priority level to 7
 *===============================================================================================================*/

/* T1 - Interrupt Handler for Enforcing Sample Time */
void __ISR(_TIMER_1_VECTOR, IPL7AUTO) T1Interrupt(void)
{
    run_time++;
    wait_flag = 1;          // Signal end of sample time
    mT1ClearIntFlag();      // Clear Timer1 interrupt flag
} // end T1Interrupt

/* AN1249 - EXAMPLE 16: SERVICING C1 ISR AND CLEARING INTERRUPT FLAGS [p. 19] */
void __ISR(_CAN_1_VECTOR, IPL4) C1Interrupt (void)
{
    if(C1INTbits.TBIF)      // Check to see if the interrupt is caused by transmit
    {
        C1INTbits.TBIF = 0;     // Clear transmit buffer interrupt flag
    }
    if(C1INTbits.RBIF)      // Check to see if the interrupt is caused by receive
    {
        if(C1FIFOINT1bits.RXNEMPTYIF == 1)  // Check to see if FIFO1 Receive Buffer is not empty
        {
            CANmsgReceived = TRUE;          // Set the buffer full flag
            C1FIFOINT1bits.RXNEMPTYIF = 0;  // Set the buffer received flag: 0 = buffer is empty
        }
        readCANmsg();           // Copy received message into a temporary buffer inside RAM
        C1INTbits.RBIF = 0;     // Clear receive buffer flag
    }
    mCAN1ClearIntFlag();    // Clear CAN1 interrupt flag
} // end C1Interrupt

void __ISR(_SPI_3_VECTOR, IPL4) SPI3Interrupt(void)
{
    if(SPI3STATbits.SPITBF == 0)    // Check to see if the interrupt was caused by SPI3TX
    {} // No action
    if(SPI3STATbits.SPIRBF == 1)    // Check to see if the receive buffer (SPI3RXB) is full
    {} // No action

    IFS0bits.SPI3TXIF = 0;      // Clear SPI3 TX interrupt flag
    IFS0bits.SPI3RXIF = 0;      // Clear SPI3 RX interrupt flag
    IFS0bits.SPI3EIF = 0;       // Clear SPI3 error interrupt flag
    IEC0bits.SPI3TXIE = 0;      // Disable SPI3 TX interrupt
    IEC0bits.SPI3RXIE = 0;      // Disable SPI3 RX interrupt
    IEC0bits.SPI3EIE = 0;       // Disable SPI3 error interrupt
} // end SPI3Interrupt

void __ISR(_SPI_4_VECTOR, IPL4) SPI4Interrupt(void)
{
    if(SPI4STATbits.SPITBF == 0)    // Check to see if the interrupt was caused by SPI4TX
    {} // No action
    if(SPI4STATbits.SPIRBF == 1)    // Check to see if the receive buffer (SPI4RXB) is full
    {} // No action

    IFS1bits.SPI4TXIF = 0;      // Clear SPI4 TX interrupt flag
    IFS1bits.SPI4RXIF = 0;      // Clear SPI4 RX interrupt flag
    IFS1bits.SPI4EIF = 0;       // Clear SPI4 error interrupt flag
    IEC1bits.SPI4TXIE = 0;      // Disable SPI4 TX interrupt
    IEC1bits.SPI4RXIE = 0;      // Disable SPI4 RX interrupt
    IEC1bits.SPI4EIE = 0;       // Disable SPI4 error interrupt
} // end SPI4Interrupt
```

F-6: LCD.c

```
/*****************************************************************************************************************
Name: LCD.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used for operation of SparkFun's Serial Graphic LCD 128x64 (LCD-09351).
*****************************************************************************************************************/

#include <p32xxxx.h>
#include "support.h"

void config_LCD(void)
{
    setBacklightIntensity(25);      // Adjust LCD backlight intensity
    toggleReverseMode();            // Toggle reverse mode
} // end config_LCD

void printVariable(char *stringVar, float number, char precision, char *stringUnits)
{
    char strLength1 = 0;
    char strLength2 = 0;
    char strLength3 = 0;
    char strLengthT = 0;

    strLength1 = printString(stringVar);            // Print variable name string
    strLength2 = printFloat(number, precision);     // Print floating-point number
    strLength3 = printString(stringUnits);          // Print variable units string

    strLengthT = strLength1 + strLength2 + strLength3;  // Find total printed string length
    terminateLine(strLengthT, LCD_LINE_LENGTH);         // Terminate line
} // end printVariable

char printString(char *string)
{
    char i = 0;             // Reset counter
    char strLength = 0;     // String length

    strLength = strlen(string);     // Find length of input string

    while(i < strLength)
    {
        sendUARTbyte(string[i]);        // Print the next character
        i++;
    }
    return(strLength);
} // end printString

/*=============================================================================================================*
    printLine() prints user defined string and jumps to next line on LCD display.

    Arguments:
        *string     pointer to array that holds every character in user defined string.
        lineLength  maximum number of characters that can be printed per line on LCD display.

    Description:
        Length of one line on the LCD is 21 characters. LCD_LINE_LENGTH is defined as 21 in support.h
 *=============================================================================================================*/
void printLine(char *string, char lineLength)
{
    char strLength;                 // String length

    strLength = printString(string);        // Prints input string and returns string length
    terminateLine(strLength, lineLength);   // Jumps to new line on LCD display
} // end printLine

void terminateLine(char strLength, char lineLength)
{
    char i = 0;                     // Counter

    i = strLength;

    while(i < lineLength)           // Print spaces until LCD display jumps to next line
    {
        sendUARTbyte(' ');
        i++;
    }
} // end terminateLine

char printInteger(int number)
{
    char buf[8 * sizeof(int)];      // Array used to store the values of each digit in the input 'number'
    char strLength;                 // String length
    char digits;                    // Holds the # of digits in the input 'number'
    unsigned char i = 0;
```

```
    digits = countDigits(number);   // Counts the # of digits in the input 'number'

    if(number == 0)     // If input integer is zero then print '0' and exit function
    {
        sendUARTbyte('0');
        digits = 1;
        return(digits);
    }

    while(number > 0)   // Repeat loop until all digits of input integer have been stored in buf[]
    {
        buf[i++] = number % 10;     // Stores the last digit of the input integer into array buf[]
        number /= 10;               // Trims the last digit from input integer
    }

    while(i > 0)
    {
        // Convert the integer buf[i - 1] to ASCII code and print it on LCD display
        sendUARTbyte('0' + buf[i - 1]);     // The '0' character shifts buf[i - 1] integer by +48
        i--;        // Print the MSB first and shift to next integer until the LSB is printed
    }

    return(digits);
} // end printInteger

char countDigits(int number)
{ // countDigits() counts the number of digits in the input 'number'; '-' counts as a digit
    char digits = 0;

    while(number)
    {
        number /= 10;
        digits++;
    }

    return(digits);
} // end countDigits

char printFloat(float number, char precision)
{
    char countChar = 0;         // Used to count the number of characters in 'number'
    char digits = 0;            // Holds the # of digits in the 'int_part'
    char totalChar = 0;         // Reset total # of characters in 'number' counter
    unsigned char i = 0;
    float rounding = 0.5;

    // If input 'number' is negative, add negative sign '-' to LCD display
    if(number < 0.0)
    {
        sendUARTbyte('-');      // Add '-' sign to LCD display
        number = -number;       // Make the input 'number' positive for easier handling
        countChar++;            // Increment countChar
    }

    // Using input 'precision', decide whether last significant figure needs to round up or down
    for(i = 0; i < precision; i++)
    {
        rounding /= 10.0;
    }
    number += rounding;     // If printFloat(1.576, 2), the input 'number' is rounded up to "1.58"

    // Extract the integer part of the input 'number' and print it on LCD display
    unsigned int int_part = (unsigned int) number;
    float remainder = number - (float) int_part;
    digits = printInteger(int_part);        // Print; also, store # of digits in the 'int_part'

    // Print the decimal point, but only if the specified decimal 'precision' is greater than 0
    if(precision > 0)
    {
        sendUARTbyte('.');
        countChar++;            // Increment countChar
    }

    // Extract decimal digits from the remainder one at a time
    while(precision-- > 0)
    {
        remainder *= 10.0;                  // Shift decimal digits to the left by one place
        int decDigit = (int) remainder;     // Isolate integer and trim all decimal digits
        printInteger(decDigit);             // Print the isolated integer on LCD display
        remainder -= decDigit;              // Trim integer and store decimal digits
        countChar++;                        // Increment countChar
    }

    totalChar = digits + countChar;         // Calculate total # of char in floating 'number'

    return(totalChar);
} // end printFloat
```

```
void clearScreen(void)
{ // Clears the screen of all written pixels
    sendUARTbyte(0x7C);
    sendUARTbyte(0x00);
} // end clearScreen

void runDemo(void)
{ // Runs demonstration code
    sendUARTbyte(0x7C);
    sendUARTbyte(0x04);
} // end runDemo

void toggleReverseMode(void)
{ // Toggles between white on blue display and blue on white display
    sendUARTbyte(0x7C);
    sendUARTbyte(0x12);
} // end toggleReverseMode

void toggleSplash(void)
{ // Allows or disallows the SparkFun logo to be displayed at power up but the delay remains active
    sendUARTbyte(0x7C);
    sendUARTbyte(0x13);
} // end toggleSplash

void setBacklightIntensity(UINT8 duty)
{ // Changes the backlight intensity (and therefore current draw)
    sendUARTbyte(0x7C);
    sendUARTbyte(0x02);
    sendUARTbyte(duty);     // Enter value between 0 - 100
} // end setBacklightIntensity

void setBaudRate(UINT8 baud)
{ // Changes the baud rate between 6 predefined settings; Default baud rate is 115,200 bps
/* Baud Rate Communication Speeds:
    "1" = 4800 bps      0x31 = 49
    "2" = 9600 bps      0x32 = 50
    "3" = 19,200 bps    0x33 = 51
    "4" = 38,400 bps    0x34 = 52
    "5" = 57,600 bps    0x35 = 53
    "6" = 115,200 bps   0x36 = 54
*/
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(baud);     // Enter value between 49 - 54
    delay_ms(100);

    // If LCD baud rate changes then the baud rate of UART peripheral also needs to change
    if(baud == 49)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 1041;           // Set UART3 baud rate to approximately 4,800 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
    if(baud == 50)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 520;            // Set UART3 baud rate to approximately 9,600 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
    if(baud == 51)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 259;            // Set UART3 baud rate to approximately 19,200 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
    if(baud == 52)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 129;            // Set UART3 baud rate to approximately 38,400 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
    if(baud == 53)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 86;             // Set UART3 baud rate to approximately 57,600 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
    if(baud == 54)
    {
        U3MODEbits.ON = 0;      // Disable UART3 peripheral
        U3BRG = 42;             // Set UART3 baud rate to approximately 115,200 bps
        U3MODEbits.ON = 1;      // Enable UART3 peripheral
    }
} // end setBaudRate

void restoreDefaultBaud(void)
{ // Restore the default baud rate in case the LCD's operating baud rate is unknown
    U3MODEbits.ON = 0;      // Disable UART3 peripheral
```

```
    U3BRG = 1041;           // Set UART3 baud rate to approximately 4,800 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(54);       // Set baud rate to 115,200 bps

    U3MODEbits.ON = 0;      // Disable UART3 peripheral
    U3BRG = 520;            // Set UART3 baud rate to approximately 9,600 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(54);       // Set baud rate to 115,200 bps

    U3MODEbits.ON = 0;      // Disable UART3 peripheral
    U3BRG = 259;            // Set UART3 baud rate to approximately 19,200 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(54);       // Set baud rate to 115,200 bps

    U3MODEbits.ON = 0;      // Disable UART3 peripheral
    U3BRG = 129;            // Set UART3 baud rate to approximately 38,400 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(54);       // Set baud rate to 115,200 bps

    U3MODEbits.ON = 0;      // Disable UART3 peripheral
    U3BRG = 86;             // Set UART3 baud rate to approximately 57,600 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral
    sendUARTbyte(0x7C);
    sendUARTbyte(0x07);
    sendUARTbyte(54);       // Set baud rate to 115,200 bps

    U3MODEbits.ON = 0;      // Disable UART3 peripheral
    U3BRG = 42;             // Set UART3 baud rate to approximately 115,200 bps
    U3MODEbits.ON = 1;      // Enable UART3 peripheral

    delay_ms(10);

    clearScreen();
    printLine("Baud rate restored to 115,200", LCD_LINE_LENGTH);
    delay_ms(5000);
} // end restoreDefaultBaud

void setXref(UINT8 x)
{ // Changes the X-coordinate of the starting point where text is printed on the display
    sendUARTbyte(0x7C);
    sendUARTbyte(0x18);
    sendUARTbyte(x);        // Enter value between 0 - 127 pixels
} // end setXref

void setYref(UINT8 y)
{ // Changes the Y-coordinate of the starting point where text is printed on the display
    sendUARTbyte(0x7C);
    sendUARTbyte(0x19);
    sendUARTbyte(y);        // Enter value between 0 - 63 pixels
} // end setYref

void resetXYref(void)
{ // Reset the starting point of all text printedon the display back to zero
    // X-coordinate
    sendUARTbyte(0x7C);
    sendUARTbyte(0x18);
    sendUARTbyte(0x00);     // Set X reference point to 0
    // Y-coordinate
    sendUARTbyte(0x7C);
    sendUARTbyte(0x19);
    sendUARTbyte(0x00);     // Set Y reference point to 0
} // end resetXYref

void setPixel(UINT8 x, UINT8 y)
{ // Independently sets or resets any pixel on the display
    sendUARTbyte(0x7C);
    sendUARTbyte(0x10);
    sendUARTbyte(x);
    sendUARTbyte(y);
    sendUARTbyte(0x01);     // Enter a 0 or 1 to determine setting or resetting of the pixel
    delay_ms(10);
} // end setPixel

void drawLine(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2)
{ // Draws or erases a line
    sendUARTbyte(0x7C);
    sendUARTbyte(0x0C);
    sendUARTbyte(x1);
    sendUARTbyte(y1);
    sendUARTbyte(x2);
```

```
    sendUARTbyte(y2);
    sendUARTbyte(0x01);     // Enter a 0 or 1 to determine whether to draw or erase the line
    delay_ms(10);
} // end drawLine

void drawCircle(UINT8 x, UINT8 y, UINT8 radius)
{ // Draws a circle
    sendUARTbyte(0x7C);
    sendUARTbyte(0x03);
    sendUARTbyte(x);
    sendUARTbyte(y);
    sendUARTbyte(radius);
    sendUARTbyte(0x01);     // Enter a 0 or 1 to determine whether to draw or erase the circle
    delay_ms(10);
} // end drawCircle

void drawBox(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2)
{ // Draws a box
    sendUARTbyte(0x7C);
    sendUARTbyte(0x0F);
    sendUARTbyte(x1);
    sendUARTbyte(y1);
    sendUARTbyte(x2);
    sendUARTbyte(y2);
    sendUARTbyte(0x01);     // Enter a 0 or 1 to determine whether to draw or erase the box
    delay_ms(10);
} // end drawBox

void eraseBlock(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2)
{ // Erases all contents within the region covered by the perimeter of the block
    sendUARTbyte(0x7C);
    sendUARTbyte(0x05);
    sendUARTbyte(x1);
    sendUARTbyte(y1);
    sendUARTbyte(x2);
    sendUARTbyte(y2);
    delay_ms(10);
} // end eraseBlock
```

F-7: LQR.c

```
/*****************************************************************************************************************
Name: LQR.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Decoupled Linear Quadratic Regulator (LQR) algorithm.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include "support.h"

/*================================================================================================================*
    LQR() computes the left and right wheel motor torques required to perform decoupled LQR control.

    Local Variables:
        k_x,k_dphi,k_psi,k_Tdphi        array element identifier used in RMAfilter().
        loopCount       counter used to update data logger at 1/10th the sampling rate.
        attFactor       attenuation factor for dealing with gyroscope noise due to platform vibration.
        phiOffset       offset used to compensate for misalignment between phi and the equilibrium point.
        xArr,dphiArr,psiArr,TdphiArr    array used to store the last n variables for use in RMAfilter().
        n_x,n_dphi,n_psi,n_Tdphi        number of measurements to be averaged inside the RMAfilter().
        theta_LWrad     angular displacement of left wheel [rad].
        theta_RWrad     angular displacement of right wheel [rad].
        Gyaw    x-axis gyroscope output [deg/s].
        Ax      x-axis accelerometer output [g].
        Ay      y-axis accelerometer output [g].
        x       vehicle linear displacement along the x-axis [m].
        dx      vehicle linear velocity along the x-axis [m/s].
        phi     pendulum angular displacement or pitch angle [rad].
        dphi    pendulum angular velocity or pitch velocity [rad/s].
        psi     vehicle heading angle or yaw angle [rad].
        dpsi    vehicle heading angular velocity or yaw velocity [rad/s].
        T_LW    left wheel motor torque [Nm].
        T_RW    right wheel motor torque [Nm].
        Td_phi  disturbance torque acting on the vehicle's handlebar about the pitch axis [Nm].
        Td_psi  disturbance torque acting on the vehicle's handlebar about the yaw axis [Nm].
        x_cmd   vehicle linear displacement command calculated by the displacement controller [m].
        phi_cmd vehicle pitch angle command calculated by the pitch controller [rad].
        psi_cmd vehicle heading angle command calculated by the heading angle controller [rad].

    Header File Definitions:
        G11, G12, G21, G22      decoupling controller matrix elements.
        Kp11, Kp12, Kp13, Kp14  decoupled LQR pitch gain matrix elements.
        Ky11, Ky12              decoupled LQR yaw gain matrix elements.

    Description:
        This function collects all six state variables of the TWIP system from on-board sensors and
        utilizes a decoupled version of the standard LQR control law to compute the left and right wheel
        motor torques necessary to stabilize the platform about its equilibrium point.

        Complete execution of LQR() without data logging (i.e. no logLQRdata()) requires roughly 0.9 ms
        (measured on the oscilloscope). This time requirement constricts the maximum software sampling
        rate to roughly 1.1 kHz [1/(0.9 ms) = 1.1 kHz]. Thus, do not exceed 1.1 kHz while executing LQR().

        Maximum sampling rates (measured on the oscilloscope):
            - Executing LQR() without data logging: 1.1 kHz.
            - Executing LQR() while data logging 2 variables in logLQRdata(): 750 Hz.
            - Executing LQR() while data logging 8 variables in logLQRdata(): 200 Hz.
            - Executing LQR() while data logging all 11 variables in logLQRdata(): 160 Hz.
            - Read description in logLQRdata() for further information.

        Note: the encoder count of each motor will increment in opposite directions whenever the platform
        moves forward or backwards. To simplify future calculations, the value of 'theta_RWrad' is
        multiplied by -1 to ensure that a forward/backward motion results in equal angular displacements
        in both left and right wheels.

        Note: delay_us(70) is the tested minimum delay required to prevent the Pucks from choking. Choking
        is the term selected to describe a Puck's behavior when it receives a new request from the PIC32
        while it is still processing the previous request. Whenever the Puck chokes, the software enters
        an infinite loop where it awaits for the Puck to respond.

        Note: the attenuation factor, attFactor, is applied to Gyaw in order to deal with gyroscope noise
        instigated by platform vibration during upright stabilization. Changing the software's sampling
        rate or adding/removing the handlebar will increase/decrease platform vibration and require the
        attenuation factor to be retuned. Stabilization tests at 200 Hz revealed that the attFactor can be
        safely set to 1.0 when the handlebar is mounted on the platform.
            Recommended attFactor without handlebar: attFactor = 0.6
            Recommended attFactor with handlebar: attFactor = 1.0

        Note: LQR gain matrix calculations were performed in MATLAB to reduce computational demand.
 *================================================================================================================*/
void LQR(void)
{
    static char k_x = 0, k_dphi = 0, k_psi = 0, k_Tdphi = 0;
    static int loopCount = 0;           // Keep track of LQR() function execution count
```

```
    static float attFactor = 1.0;       // Attenuation factor for gyroscope noise due to platform vibration
    static float phiOffset = -0.06;     // Used to compensate for misalignment in equilibrium point
    static float xArr[11] = {0}, dphiArr[11] = {0}, psiArr[11] = {0}, TdphiArr[11] = {0};
    char n_x = 5;                       // Number of linear displ. measurements averaged inside RMAfilter()
    char n_dphi = 3;                    // Number of pitch rate measurements averaged inside RMAfilter()
    char n_psi = 5;                     // Number of yaw angle measurements averaged inside RMAfilter()
    char n_Tdphi = 5;                   // Number of pitch torque measurements averaged inside RMAfilter()
    float theta_LWrad = 0, theta_RWrad = 0, Gyaw = 0, Ax = 0, Ay = 0;
    float x = 0, dx = 0, phi = 0, dphi = 0, psi = 0, dpsi = 0, T_LW = 0, T_RW = 0;
    float Td_phi = 0, Td_psi = 0, x_cmd = 0, phi_cmd = 0, psi_cmd = 0;

    // Get 32-bit cumulative motor angular positions (accounts for encoder zero crossing) in radians
    theta_LWrad = getPositionPuck(0, L_WHEEL_ID, 1);        // 32-bit left motor angle [radians]
    theta_RWrad = -getPositionPuck(0, R_WHEEL_ID, 1);       // 32-bit right motor angle [radians]
    delay_us(70);                       // Tested minimum delay required to prevent Puck from choking

    // Get IMU 3-axis accelerometer outputs and store returned values inside IMUdataSI structure
    Gyaw = attFactor*get_IMUgyro(&IMUdataSI, 1);    // X-axis gyro output [deg/s] with n-point avg. filter
    Ax = get_IMUxaccl(&IMUdataSI);                  // X-axis accelerometer output  [g or *9.81 m/s^2]
    Ay = get_IMUyaccl(&IMUdataSI);                  // Y-axis accelerometer output  [g or *9.81 m/s^2]

    // Calculate system state variables using data collected from on-board sensors
    x = -getStateVariable_x(theta_LWrad, theta_RWrad);              // x [m]
    x = RMAfilter(x, n_x, &k_x, xArr);                              // Passing x through RMA filter
    dx = getStateVariable_dx(x);                                    // dx [m/s]
    phi = getStateVariable_phi(Gyaw, Ax, Ay) + phiOffset;           // phi [rad]
    dphi = Gyaw * PI/180;                                           // dphi [rad/s]
    dphi = RMAfilter(dphi, n_dphi, &k_dphi, dphiArr);               // Passing dphi through RMA filter
    psi = -getStateVariable_psi(theta_LWrad, theta_RWrad);          // psi [rad]
    psi = RMAfilter(psi, n_psi, &k_psi, psiArr);                    // Passing psi through RMA filter
    dpsi = getStateVariable_dpsi(psi);                              // dpsi [rad/s]

    // Get filtered/normalized torque data collected from pitch and yaw load cells [in Nm]
    Td_phi = getLoadCellData_pitch();                               // Td_phi [Nm]
    Td_phi = RMAfilter(Td_phi, n_Tdphi, &k_Tdphi, TdphiArr);        // Passing Td_phi through RMA filter
    Td_psi = getLoadCellData_yaw();                                 // Td_psi [Nm]

    // Implement user intent control: pitch, displacement, and heading angle controllers
    //x_cmd = DisplacementControl(0.1, 1, 0, 0.1, Td_phi);
    x_cmd = DisplacementControl_v2(0.02, 0.1, 0, 0.8, -x);
    //phi_cmd = PitchControl(0.1, 0.4, 0, 0.2618, Td_phi);
    psi_cmd = -HeadingAngleControl(0.05, 1, 0, 1.0, Td_psi);

    // Update platform displacement, pitch angle, and heading angle using user intent controller data
    x += x_cmd;
    //phi += phi_cmd;
    psi += psi_cmd;

    // Left and right wheel motor torque [Nm] equations derived using decoupled LQR control
    T_LW = (G11*Kp11*x + G11*Kp12*dx + G11*Kp13*phi + G11*Kp14*dphi + G12*Ky11*psi + G12*Ky12*dpsi)/4;
    T_RW = (G21*Kp11*x + G21*Kp12*dx + G21*Kp13*phi + G21*Kp14*dphi + G22*Ky11*psi + G22*Ky12*dpsi)/4;
    T_RW = -T_RW;                       // Multiply T_RW by -1 to account for mirrored motor placement

    // Convert torque values from SI units to puck units and send CAN message to update puck torque
    setTorquePuck(0, L_WHEEL_ID, T_LW);
    setTorquePuck(0, R_WHEEL_ID, T_RW);
    delay_us(70);                       // Tested minimum delay required to prevent Puck from choking

    // Print variables on LCD display
    //printLQRdataLCD(theta_LWrad,theta_RWrad,x,dx,phi,dphi,0,0,0,0);

    loopCount++;
    // Submit a complete set of data to MATLAB at 1/Nth the sampling rate set in init_samptime()
    if(loopCount == 1)
    {
        // Send MATLAB comma-delimited data
        printFloat(Td_phi, 5); sendUARTbyte(','); //sendUARTbyte('$');
        printFloat(Td_psi, 5); sendUARTbyte(',');
        printFloat(x_cmd, 5); sendUARTbyte(',');
        printFloat(psi_cmd, 5); sendUARTbyte(',');

        logLQRdata(theta_LWrad,theta_RWrad,x,dx,phi,dphi,psi,dpsi,T_LW,T_RW);

        loopCount = 0;                      // Reset LQR() execution counter
    }

    TMR4 = 0;                           // Reset elapsed time counter (Timer4)
} // end LQR()

float getStateVariable_x(float theta_LWrad, float theta_RWrad)
{
    float x;

    // Convert angular displacement of left and right wheels to linear displacement along the x-axis
    x = (theta_LWrad + theta_RWrad)*WHEEL_RADIUS/2;

    PUCKdataSI[0].x = x;            // Store x in structure for use in encoderZeroCrossing()
```

```
    return(x);
} // end getStateVariable_x

float getStateVariable_dx(float x)
{
    static float x_old = 0;
    float x_new, dx, dt;

    x_new = x;
    dt = SAMP_RATE;                // Total elapsed time [sec]

    // Approximation of vehicle velocity along the x-axis
    dx = (x_new - x_old)/dt;

    x_old = x_new;

    return(dx);
} // end getStateVariable_dx

/*=============================================================================================================*
    getStateVariable_phi() computes TWIP pitch angle using a first-order complementary filter.

    Arguments:
        Gyaw    x-axis gyroscope output [deg/s].
        Ax      x-axis accelerometer output [g].
        Ay      y-axis accelerometer output [g].

    Local Variables:
        tau     time constant [s]; time at which gyroscope measurement loses precedence and accelerometer
                reading starts to be averaged heavily.
        dt      Sample period; the amount of time that passes between each program loop [s].
        a       filter coefficient that determines the cutoff time for trusting the gyroscope and
                filtering in the accelerometer.

    Description:
        This function uses a complementary filter which combines gyroscope and accelerometer measurements
        in order to provide a responsive, drift-free, and more accurate estimate of the TWIP's pitch angle.

        The filter's time constant, tau, defines how much time the integrated gyroscope angle estimate is
        given precedence before the accelerometer average is given more weighting.
            Time constant:  tau = (a * dt)/(1 - a)

        Tips regarding tuning the filter's time constant, tau:
            tau > 0.7: gives gyro measurements greater influence. Setting tau to 0.7 or higher allows the
              pitch angle estimations to be more accurate when the IMU is in motion. Since the platform
              will be in frequent motion, it is best to keep tau higher than 0.7.
            tau < 0.3: gives accelerometer measurements greater influence. Setting tau to 0.3 or lower
              allows the pitch angle estimations to be more accurate when the IMU is NOT in motion.
              Setting tau to 0.3 or lower will cause the platform to vibrate relentlessly due to inaccurate
              pitch angle estimations.

        Note: changing software sampling rate may require adjusting the time constant, tau.
 *=============================================================================================================*/
float getStateVariable_phi(float Gyaw, float Ax, float Ay)
{
    static float phi_deg = 0;
    static float tau = 0.8;             // Time constant for both the low-pass and high-pass filters
    float phi_rad, dphi, dt, a, num, den, phi_accl;

    dphi = Gyaw;                            // x-axis gyroscope output [deg/s]
    dt = SAMP_RATE;                         // Total elapsed time [sec]

    // Calculating filter coefficient using time constant, tau, and sample period, dt
    a = tau/(tau + dt);

    // Angle of rotation about the IMU's z-axis (or TWIP's pitch axis)
    num = -Ay;      // Multiply by -1 to make TWIPpitch read zero degrees when the TWIP is upright
    den = Ax;
    phi_accl = atanf(num/den) * 180/PI;     // TWIP pitch [deg] calculated using accelerometers

    // Calculating TWIP pitch angle [deg] using first-order complementary filter
    phi_deg = a*(phi_deg + dphi*dt) + (1 - a)*phi_accl;             // phi_deg [deg]

    // Convert phi from degrees to radians
    phi_rad = phi_deg * PI/180;                                     // phi_rad [rad]

    return(phi_rad);
} // end getStateVariable_phi

float getStateVariable_psi(float theta_LWrad, float theta_RWrad)
{
    float psi;

    // Convert angular displacement of left and right wheels to yaw angle
    psi = WHEEL_RADIUS*(theta_LWrad - theta_RWrad)/DIST_LWtoRW;     // psi [rad]

    return(psi);
} // end getStateVariable_psi
```

```
float getStateVariable_dpsi(float psi)
{
    static float psi_old = 0;
    float psi_new, dpsi, dt;

    psi_new = psi;
    dt = SAMP_RATE;                    // Total elapsed time [sec]

    // Approximation of vehicle yaw angular velocity
    dpsi = (psi_new - psi_old)/dt;     // dpsi [rad/s]

    psi_old = psi_new;

    return(dpsi);
} // end getStateVariable_dpsi

float RMAfilter(float var, char n, char *k_ptr, float *varArr)
{ // Implementing n-point recursive moving average (RMA) filter
    char j = 0;
    float var_sum = 0;

    varArr[*k_ptr] = var;              // Store the latest input variable to be filtered
    *k_ptr = *k_ptr + 1;               // Increment pointer k_ptr holding array element identifier
    if (*k_ptr == n) {*k_ptr = 0;}     // Reset k_ptr counter when number of points stored equals n
    for (j = 0; j < n; j++)            // Calculate the sum of the latest n-points
    {var_sum = var_sum + varArr[j];}
    var = var_sum/n;                   // Calculate the average of latest n input variables

    return(var);
} // end RMAfilter

/*=============================================================================================================*
    DisplacementControl() controls vehicle linear displ. based on measured torque from pitch load cell.

    Arguments:
        MinT_x  defines the min torque [Nm] required before linear displacement control will begin.
        MaxT_x  defines the max torque [Nm] at which the platform will reach maximum velocity.
        MinV_dx defines the min linear velocity [m/s] for the platform (always zero).
        MaxV_dx defines the max linear velocity [m/s] for the platform (occurs when Td_phi = +/- MaxT_x).
        Td_phi  disturbance torque [Nm] measured by the pitch load cell.

    Description:
        This function controls how fast the platform travels forward/backward and calculates the new
        stability point along the x-axis based on measured torque from pitch load cell.

        Note: x_cmdold accumulates total distance traveled. At some point, this 32-bit floating point is
        going to overflow and cause an arithmetic error. There are other variables in the entire firmware
        that are vulnerable to such errors. These fixes are outside the scope of my current research but
        will need to be fixed in the future.
 *=============================================================================================================*/
float DisplacementControl(float MinT_x, float MaxT_x, float MinV_dx, float MaxV_dx, float Td_phi)
{
    static float x_cmdold = 0, dx_cmdold = 0, dx_cmdPIold = 0, error_old = 0;
    float x_cmd = 0, dx_cmd = 0, dx_cmdMJT = 0, dx_cmdPI = 0, error = 0;
    float Kp = 0.1;
    float Ki = 10;
    float Ts = SAMP_RATE;

    // Min torque requirement to prevent controller from reacting to noise or non-user disturbances
    if(Td_phi > -MinT_x && Td_phi < MinT_x)
    {dx_cmd = 0;}               // Set displacement velocity to 0 [m/s]

    // Displ. velocity set to absolute maximum (positive) when measured pitch LC torque reaches upper limit
    if(Td_phi > MaxT_x)
    {dx_cmd = MaxV_dx;}         // Set displacement velocity to MaxV_dx [m/s]

    // Displ. velocity set to absolute minimum (negative) when measured pitch LC torque reaches lower limit
    if(Td_phi < -MaxT_x)
    {dx_cmd = -MaxV_dx;}        // Set displacement velocity to -MaxV_dx [m/s]

    // Use MinJerkTraj() to calculate displ. vel. when torque is between positive lower and upper limits
    if(Td_phi >= MinT_x && Td_phi <= MaxT_x)
    {
        // Use minimum jerk trajectory function to calculate displacement velocity command
        dx_cmdMJT = MinJerkTraj(MinT_x, MaxT_x, MinV_dx, MaxV_dx, Td_phi);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dx_cmdMJT - dx_cmdPIold;        // Error = Expected Output - Actual Output
        dx_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dx_cmdPIold);
        dx_cmd = dx_cmdPI;
    }

    // Use MinJerkTraj() to calculate displ. vel. when torque is between negative lower and upper limits
    if(Td_phi <= -MinT_x && Td_phi >= -MaxT_x)
    {
        // Use minimum jerk trajectory function to calculate displacement velocity command
        dx_cmdMJT = MinJerkTraj(-MinT_x, -MaxT_x, -MinV_dx, -MaxV_dx, Td_phi);
```

```
        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dx_cmdMJT - dx_cmdPIold;        // Error = Expected Output - Actual Output
        dx_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dx_cmdPIold);
        dx_cmd = dx_cmdPI;
    }

    // Integrate dx_cmd, using Tustin's approximation, to obtain linear displacement command, x_cmd
    x_cmd = x_cmdold + Ts*(dx_cmd + dx_cmdold)/2;

    error_old = error;
    dx_cmdPIold = dx_cmdPI;
    x_cmdold = x_cmd;
    dx_cmdold = dx_cmd;

    return(x_cmd);
} // end DisplacementControl

/*=============================================================================================================*
    DisplacementControl_v2() controls vehicle linear displ. based on measured linear displ. from encoders.

    Description:
        This function is very similar to DisplacementControl() but instead of using measured torque from
        pitch load cell as the input variable, it uses the vehicle's linear displacement along the x-axis,
        x, as the input variable.
 *=============================================================================================================*/
float DisplacementControl_v2(float MinD_x, float MaxD_x, float MinV_dx, float MaxV_dx, float x)
{
    static float x_cmdold = 0, dx_cmdold = 0, dx_cmdPIold = 0, error_old = 0;
    float x_cmd = 0, dx_cmd = 0, dx_cmdMJT = 0, dx_cmdPI = 0, error = 0;
    float Kp = 0.1;
    float Ki = 10;
    float Ts = SAMP_RATE;

    // Subtract x_cmdold from x to ensure that x measurements take into account displ. commands, x_cmd
    x = x - x_cmdold;

    // Min displacement requirement to prevent controller from reacting to noise or non-user disturbances
    if(x > -MinD_x && x < MinD_x)
    {dx_cmd = 0;}               // Set displacement velocity to 0 [m/s]

    // Displ. velocity set to absolute maximum (positive) when measured displacement reaches upper limit
    if(x > MaxD_x)
    {dx_cmd = MaxV_dx;}         // Set displacement velocity to MaxV_dx [m/s]

    // Displ. velocity set to absolute minimum (negative) when measured displacement reaches lower limit
    if(x < -MaxD_x)
    {dx_cmd = -MaxV_dx;}        // Set displacement velocity to -MaxV_dx [m/s]

    // Use MinJerkTraj() to calculate displ. vel. when x is between positive lower and upper limits
    if(x >= MinD_x && x <= MaxD_x)
    {
        // Use minimum jerk trajectory function to calculate displacement velocity command
        dx_cmdMJT = MinJerkTraj(MinD_x, MaxD_x, MinV_dx, MaxV_dx, x);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dx_cmdMJT - dx_cmdPIold;        // Error = Expected Output - Actual Output
        dx_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dx_cmdPIold);
        dx_cmd = dx_cmdPI;
    }

    // Use MinJerkTraj() to calculate displ. vel. when x is between negative lower and upper limits
    if(x <= -MinD_x && x >= -MaxD_x)
    {
        // Use minimum jerk trajectory function to calculate displacement velocity command
        dx_cmdMJT = MinJerkTraj(-MinD_x, -MaxD_x, -MinV_dx, -MaxV_dx, x);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dx_cmdMJT - dx_cmdPIold;        // Error = Expected Output - Actual Output
        dx_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dx_cmdPIold);
        dx_cmd = dx_cmdPI;
    }

    // Integrate dx_cmd, using Tustin's approximation, to obtain linear displacement command, x_cmd
    x_cmd = x_cmdold + Ts*(dx_cmd + dx_cmdold)/2;

    error_old = error;
    dx_cmdPIold = dx_cmdPI;
    x_cmdold = x_cmd;
    dx_cmdold = dx_cmd;

    return(x_cmd);
} // end DisplacementControl_v2

float PitchControl(float MinT_phi, float MaxT_phi, float MinA_phi, float MaxA_phi, float Td_phi)
{
    static float phi_cmdPIold = 0, error_old = 0;
    float phi_cmd = 0, phi_cmdMJT = 0, phi_cmdPI = 0, error = 0;
```

```
    float Kp = 0.1;
    float Ki = 10;
    float Ts = SAMP_RATE;

    // Min torque requirement to prevent controller from reacting to noise or non-user disturbances
    if(Td_phi > -MinT_phi && Td_phi < MinT_phi)
    {phi_cmd = 0;}              // Set pitch angle to 0 [rad]

    // Pitch angle set to absolute maximum (positive) when measured pitch LC torque reaches upper limit
    if(Td_phi > MaxT_phi)
    {phi_cmd = MaxA_phi;}       // Set pitch angle to MaxA_phi [rad]

    // Pitch angle set to absolute minimum (negative) when measured pitch LC torque reaches lower limit
    if(Td_phi < -MaxT_phi)
    {phi_cmd = -MaxA_phi;}      // Set pitch angle to -MaxA_phi [rad]

    // Use MinJerkTraj() to calculate pitch angle when torque is between positive lower and upper limits
    if(Td_phi >= MinT_phi && Td_phi <= MaxT_phi)
    {
        // Use minimum jerk trajectory function to calculate pitch angle command
        phi_cmdMJT = MinJerkTraj(MinT_phi, MaxT_phi, MinA_phi, MaxA_phi, Td_phi);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = phi_cmdMJT - phi_cmdPIold;      // Error = Expected Output - Actual Output
        phi_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, phi_cmdPIold);
        phi_cmd = phi_cmdPI;
    }

    // Use MinJerkTraj() to calculate pitch angle when torque is between negative lower and upper limits
    if(Td_phi <= -MinT_phi && Td_phi >= -MaxT_phi)
    {
        // Use minimum jerk trajectory function to calculate pitch angle command
        phi_cmdMJT = MinJerkTraj(-MinT_phi, -MaxT_phi, -MinA_phi, -MaxA_phi, Td_phi);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = phi_cmdMJT - phi_cmdPIold;      // Error = Expected Output - Actual Output
        phi_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, phi_cmdPIold);
        phi_cmd = phi_cmdPI;
    }

    error_old = error;
    phi_cmdPIold = phi_cmdPI;

    return(phi_cmd);
} // end PitchControl

float HeadingAngleControl(float MinT_psi, float MaxT_psi, float MinV_dpsi, float MaxV_dpsi, float Td_psi)
{
    static float psi_cmdold = 0, dpsi_cmdold = 0, dpsi_cmdPIold = 0, error_old = 0;
    float psi_cmd = 0, dpsi_cmd = 0, dpsi_cmdMJT = 0, dpsi_cmdPI = 0, error = 0;
    float Kp = 0.1;
    float Ki = 10;
    float Ts = SAMP_RATE;

    // Min torque requirement to prevent controller from reacting to noise or non-user disturbances
    if(Td_psi > -MinT_psi && Td_psi < MinT_psi)
    {dpsi_cmd = 0;}             // Set heading velocity to 0 [rad/s]

    // Heading velocity set to absolute maximum (positive) when measured yaw LC torque reaches upper limit
    if(Td_psi > MaxT_psi)
    {dpsi_cmd = MaxV_dpsi;}     // Set heading velocity to MaxV_dpsi [rad/s]

    // Heading velocity set to absolute minimum (negative) when measured yaw LC torque reaches lower limit
    if(Td_psi < -MaxT_psi)
    {dpsi_cmd = -MaxV_dpsi;}    // Set heading velocity to -MaxV_dpsi [rad/s]

    // Use MinJerkTraj() to calculate yaw velocity when torque is between positive lower and upper limits
    if(Td_psi >= MinT_psi && Td_psi <= MaxT_psi)
    {
        // Use minimum jerk trajectory function to calculate heading velocity command
        dpsi_cmdMJT = MinJerkTraj(MinT_psi, MaxT_psi, MinV_dpsi, MaxV_dpsi, Td_psi);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dpsi_cmdMJT - dpsi_cmdPIold;        // Error = Expected Output - Actual Output
        dpsi_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dpsi_cmdPIold);
        dpsi_cmd = dpsi_cmdPI;
    }

    // Use MinJerkTraj() to calculate yaw velocity when torque is between negative lower and upper limits
    if(Td_psi <= -MinT_psi && Td_psi >= -MaxT_psi)
    {
        // Use minimum jerk trajectory function to calculate heading velocity command
        dpsi_cmdMJT = MinJerkTraj(-MinT_psi, -MaxT_psi, -MinV_dpsi, -MaxV_dpsi, Td_psi);

        // Use PI controller to smoothen and dampen jerking in the values calculated by MinJerkTraj()
        error = dpsi_cmdMJT - dpsi_cmdPIold;        // Error = Expected Output - Actual Output
        dpsi_cmdPI = PIcontroller(Kp, Ki, Ts, error, error_old, dpsi_cmdPIold);
        dpsi_cmd = dpsi_cmdPI;
```

```
    }

    // Integrate dpsi_cmd, using Tustin's approximation, to obtain heading angle command, psi_cmd
    psi_cmd = psi_cmdold + Ts*(dpsi_cmd + dpsi_cmdold)/2;

    error_old = error;
    dpsi_cmdPIold = dpsi_cmdPI;
    psi_cmdold = psi_cmd;
    dpsi_cmdold = dpsi_cmd;

    return(psi_cmd);
} // end HeadingAngleControl

/*=============================================================================================================*
    MinJerkTraj() calculates a smooth trajectory path given a set of lower and upper limits.

    Input Arguments:
        xi      lower limit value (e.g. min torque) along the x-axis.
        xf      upper limit value (e.g. max torque) along the x-axis.
        yi      lower limit value (e.g. min angle) along the y-axis.
        yf      upper limit value (e.g. max angle) along the y-axis.
        t       independent variable (e.g. measured load cell torque).

    Description:
        This is a modified version of the standard minimum jerk trajectory function. This function is used
        allow the user intent functions to follow a smooth trajectory path with minimized jerk within the
        defined set of lower and upper limits.
 *=============================================================================================================*/
float MinJerkTraj(float xi, float xf, float yi, float yf, float t)
{
    float cmd = 0;

    cmd = yi + (yf - yi)*(10*pow((t-xi)/(xf-xi),3) - 15*pow((t-xi)/(xf-xi),4) + 6*pow((t-xi)/(xf-xi),5));

    return(cmd);
} // end MinJerkTraj

/*=============================================================================================================*
    PIcontroller() minimizes the error between expected and actual output based on Kp and Ki values.

    Input Arguments:
        Kp          proportional constant.
        Ki          integral constant.
        Ts          loop execution time (i.e. sampling rate).
        error       error obtained by subtracting Actual (PI) Output from Expected (MinJerkTraj) Output.
        error_old   old error calculated during previous loop execution.
        output_old  old output calculated during previous loop execution.

    Description:
        The purpose of this PI controller is to smoothen and dampen the series of values calculated by the
        minimum jerk trajectory function.
 *=============================================================================================================*/
float PIcontroller(float Kp, float Ki, float Ts, float error, float error_old, float output_old)
{
    float K1 = 0, K2 = 0, output = 0;

    K1 = Kp + Ki*(Ts/2);
    K2 = -Kp + Ki*(Ts/2);

    output = output_old + K1*error + K2*error_old;

    return(output);
} // end PIcontroller

void printLQRdataLCD(float theta_LWrad, float theta_RWrad, float x, float dx, float phi, float dphi,
                     float psi, float dpsi, float T_LW, float T_RW)
{
    // Print variables on LCD display
    printVariable("thLW = ", theta_LWrad, 3, " rad");   // Print value on LCD with 4 decimal places
    printVariable("thRW = ", theta_RWrad, 3, " rad");
    printVariable("   x = ", x, 3, " m");
    printVariable("  dx = ", dx, 3, " m/s");
    printVariable(" phi = ", phi, 3, " deg");
    printVariable("Gyaw = ", dphi, 3, " deg/s");
    printVariable(" psi = ", psi, 3, " rad");
    printVariable("dpsi = ", dpsi, 3, " rad/s");
} // end printLQRdataLCD

/*=============================================================================================================*
    logLQRdata() delimits input values and forwards them to MATLAB for processing and data logging.

    Arguments:
        theta_LWrad     angular displacement of left wheel [rad].
        theta_RWrad     angular displacement of right wheel [rad].
        x       vehicle linear displacement along the x-axis [m].
        dx      vehicle linear velocity along the x-axis [m/s].
        phi     pendulum angular displacement or pitch angle [rad].
        dphi    pendulum angular velocity or pitch velocity [rad/s].
```

```
        psi     vehicle heading angle or yaw angle [rad].
        dpsi    vehicle heading angular velocity or yaw velocity [rad/s].
        T_LW    motor torque applied to the left wheel [Nm].
        T_RW    motor torque applied to the right wheel [Nm].

    Local Variables:
        TXcount     counter used to tag transmitted data sets and check for data loss.

    Description:
        This function is used to send MATLAB comma-delimited data sets for data logging and storage.
        Every data set transmission begins with a static counter that is used in MATLAB to ensure no data
        loss during transfer.

        Complete execution of logLQRdata() requires roughly 5.15 ms (measured on the oscilloscope). This
        time requirement constricts the maximum software sampling rate while both LQR() and logLQRdata()
        functions are executing to roughly 160 Hz [1/(0.9 ms + 5.15 ms) = 165.29].

        To increase that maximum software sampling rate while data logging, consider transmitting only the
        variables that will be observed in MATLAB and removing/commenting the irrelevant variables.

        Time required to execute logLQRdata():
            - Executing logLQRdata() with 2 variables (3 decimals) transmitted to MATLAB: 0.4 ms
            - Executing logLQRdata() with 8 variables (3 decimals) transmitted to MATLAB: 3.7 ms
            - Executing logLQRdata() with all variables (3 decimals) transmitted to MATLAB: 5.15 ms

        This function is designed to interact with PICtoPC_datalogger.m file.
 *=================================================================================================*/
void logLQRdata(float theta_LWrad, float theta_RWrad, float x, float dx, float phi, float dphi,
                float psi, float dpsi, float T_LW, float T_RW)
{
    static int TXcount = 1;     // Counter used to tag transmitted data sets and check for data loss

    //printInteger(TXcount);    sendUARTbyte(',');          // Send MATLAB data set counter

    // Send MATLAB left and right wheel angular displacements (comma-delimited)
    //printFloat(theta_LWrad, 3);       sendUARTbyte(',');  // Add comma to separate data
    //printFloat(theta_RWrad, 3);       sendUARTbyte(',');

    // Send MATLAB all six system state variables (comma-delimited) in ASCII code (0 to 255)
    printFloat(x, 5);       sendUARTbyte(',');          // Add comma to separate data
    printFloat(dx, 5);      sendUARTbyte(',');
    printFloat(phi, 5);     sendUARTbyte(',');
    printFloat(dphi, 5);    sendUARTbyte(',');
    printFloat(psi, 5);     sendUARTbyte(',');
    printFloat(dpsi, 5);    sendUARTbyte(',');

    // Send MATLAB left and right wheel motor torques
    printFloat(T_LW, 5);            sendUARTbyte(',');
    printFloat(T_RW, 5);

    // End MATLAB's fscanf() with terminator character '$' to mark the end of a new set of data
    sendUARTbyte('$');

    TXcount++;                  // Increment data set counter
} // end logLQRdata
```

F-8: Puck.c

```
/*****************************************************************************************************************
Name: Puck.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used for Puck operation.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include "support.h"

void config_pucks(char puckMODE)
{
    // Wait for the Puck to start-up before initializing communication
    delay_ms(300);

    // Wake up pucks
    setPropertyPuck(0, L_WHEEL_ID, STAT, STATUS_READY);     // DATA: 8500 0200
    setPropertyPuck(0, R_WHEEL_ID, STAT, STATUS_READY);
    delay_ms(1000); // Wait 1000 ms for pucks to wake up

    // Set Baud rate of each Puck to 9600 bps
    setPropertyPuck(0, L_WHEEL_ID, BAUD, 9600);
    setPropertyPuck(0, R_WHEEL_ID, BAUD, 9600);
    delay_ms(500);

    // Reset encoder count (encoders must be reset before TORQUE mode is set)
    setPropertyPuck(0, L_WHEEL_ID, P, 0);
    setPropertyPuck(0, R_WHEEL_ID, P, 0);
    delay_ms(50);

    // Set pucks to TORQUE mode
    setPropertyPuck(0, L_WHEEL_ID, MODE, puckMODE);     // DATA: 8800 0200
    setPropertyPuck(0, R_WHEEL_ID, MODE, puckMODE);
    delay_ms(50);

    // Set commanded torque to 0
    setPropertyPuck(0, L_WHEEL_ID, T, 0);                   // DATA: AA00 0000
    setPropertyPuck(0, R_WHEEL_ID, T, 0);
    delay_ms(50);

    // Check if desired Puck mode is set to TORQUE
    if(puckMODE == MODE_TORQUE)
    {
        // Set maximum torque (MAX_TORQUE = 4200 mA)
        //setPropertyPuck(0, L_WHEEL_ID, MT, MAX_TORQUE);           // DATA: AB00 6810
        //setPropertyPuck(0, R_WHEEL_ID, MT, MAX_TORQUE);
        setPropertyPuck(0, L_WHEEL_ID, MT, 4096);           // DATA: AB00 6810
        setPropertyPuck(0, R_WHEEL_ID, MT, 4096);
        delay_ms(50);
    }

    // Check if desired Puck mode is set to VELOCITY
    if(puckMODE == MODE_VELOCITY)
    {
        // Set maximum velocity (MAX_VELOCITY = 1500 cts/ms)
        setPropertyPuck(0, L_WHEEL_ID, MV, MAX_VELOCITY);       // DATA: AD00 DC05
        setPropertyPuck(0, R_WHEEL_ID, MV, MAX_VELOCITY);
        delay_ms(50);
    }
} // end config_pucks

/* MSGID() formats Puck CAN message ID */
int MSGID(int GRP, int FROM, int TO)
{
/*
    Puck MSGID
    --------------      G = Group flag. If '1', then interpret 'To' as GroupID
    GRP  FROM   TO      F = 5-bit 'From' address
     G  FFFFF TTTTT     T = 5-bit 'To' adress (or GroupID)
*/
    int msgid;

    msgid = (GRP << 10) | (FROM << 5) | TO;
    return(msgid);
} // end MSGID

int wakePuck(int channel, int who)
{
/* Inputs:
    channel -> Identifies the desired CAN Channel (FIFO Number)
    who -> Puck ID Number (see support.h)
*/
    int tx_id;
```

```
    tx_id = MSGID(0, CPU, who);
    setPropertyPuck(channel, tx_id, STAT, STATUS_READY); // See support.h for STAT and STATUS_READY
    delay_ms(300);  // Wait 300 ms for puck to initialize

    return(0);
} // end wakePuck

int setPropertyPuck(int channel, int tx_id, int property, long dataValue)
{
    int dataLength = 0;
    int i = 0;
    unsigned char tx_data[8] = {0};

    // Inserting 'property' into Data Field
    tx_data[0] =  0x80 | property;  // 0x80 is the 'Set property' bit
    tx_data[1] = 0; // Second byte (almost) always set to zero (see Puck MSG Format PDF for exceptions)

    // Determining length of Data Field in bytes
    if(dataValue <= 0xFFFF) dataLength = 4;
    if(dataValue > 0xFFFF && dataValue <= 0xFFFFFFFF) dataLength = 6;

    // Inserting 'dataValue' into Data Field
    for (i = 2; i < dataLength; i++)
    {
        tx_data[i] = (char)(dataValue & 0x000000FF);
        dataValue >>= 8;
    }

    // Send CAN message
    sendCANmsg(channel, tx_id, dataLength, tx_data);

    return(0);
} // end setProperty

int setPropertySlowPuck(int channel, int tx_id, int property, long dataValue)
{
    setPropertyPuck(channel, tx_id, property, dataValue);
    delay_us(100);

    return(0);
} // end setPropertySlowPuck

int getPropertyPuck(int channel, int tx_id, int property, long *reply)
{
    int dataLength = 1;
    unsigned char tx_data[8] = {0};
    int rx_id;
    int rx_property;

    // Data Field is 1 byte long consisting of requested property
    tx_data[0] = property;  // Request bit set to 'Get property'

    // Send CAN message
    sendCANmsg(channel, tx_id, dataLength, tx_data);

    // Wait until interrupt occurs and CAN message has been received
    while(CANmsgReceived == FALSE) {};

    // Once C1Interrupt() is cleared, received message will be ready for use inside structure 'rxCANmsg'

    // The contents of the received message must now be extracted and organized into usable form
    parseCANmsgPuck(&rx_id, &rx_property, reply);   // This fnctn returns processed data inside 'reply'

    // Check if the values sent matches the values received for Puck ID and property
    if((tx_id == rx_id) && (property == rx_property))
    {
        CANmsgReceived = FALSE; // Buffer is now ready to receive new message
        return(0);
    }
    else
    {
        LED9_ON;
        printString("getPropertyPuck(): returned Puck ID or property do not match");
        return(1);
    }
} // end getProperty

/*=================================================================================================*
    parseCANmsgPuck() processes the CAN messages received from the Puck and extracts the data value.

    Input Arguments:
        rxCANmsg        contains the id, data_length, and data[8] of incoming message (globally defined).

    Output Arguments:
        *rx_id          a pointer to the message ID received from the puck.
        *rx_property    a pointer to the property received from the puck.
        *rx_value       a pointer to the value of the property received from the puck.
        jointPosition[] stores joint encoder positions for motor pucks (globally defined in main.c).
```

```
    Description:
        Sample 22-bit Packed Position Data:
           MSGID     DLC    D0       D1       D2
                          10MMMMMM mmmmmmmm LLLLLLLL
        10000100011 0011 10010010 11010110 10000111
           0x423      3    0x12     0xD6     0x87
        From ID 1, to GroupID 3, Len = 3, Packed position = 1234567

        Note:
        'dataFormat' determines the data format of the incoming message
        dataFormat = 6 if receiving property feedback (non-position) [GroupID 6]
        dataFormat = 3 if receiving a 22-bit packed position data [GroupID 3]
        dataFormat = 0 if receiving firmware request
 *=================================================================================================*/
int parseCANmsgPuck(int *rx_id, int *rx_property, long *rx_value)
{
    int dataFormat;
    int i;

    // Extract Puck ID value from received CAN message SID
    *rx_id = (rxCANmsg.msgSID.SID >> 5) & 0x01F;

    // Extract Puck GroupID value from received CAN message SID
    dataFormat = rxCANmsg.msgSID.SID & 0x01F;

    switch (dataFormat)
    {
    case 6:  // GroupID 6: Property feedback (non-position)
        *rx_property = rxCANmsg.data[0] & 0x7F;

        // Check sign of received data: TRUE = data is negative; FALSE = data is positive
        *rx_value = rxCANmsg.data[rxCANmsg.msgEID.DLC - 1] & 0x80 ? -1 : 0; // Ternary operation

        // Process received data; second byte of message is zero (for DSP word alignment)
        for (i = rxCANmsg.msgEID.DLC - 1; i >= 2; i--)
            *rx_value = *rx_value << 8 | rxCANmsg.data[i];
        break;

    case 3:  // GroupID 3: Position feedback (P); data is a packed 22-bit position
        *rx_property = P;   // Set rx_property to puck Position (P) feedback property value

        // Process 32-bit position (P)
        *rx_value = 0;
        *rx_value |= ( (long)rxCANmsg.data[0] << 16) & 0x003F0000;
        *rx_value |= ( (long)rxCANmsg.data[1] << 8 ) & 0x0000FF00;
        *rx_value |= ( (long)rxCANmsg.data[2] ) & 0x000000FF;

        if (*rx_value & 0x00200000)     // Check if received data value is negative
            *rx_value |= 0xFFC00000;    // Convert signed 22-bit value to signed 32-bit value

        // Process joint encoder position (JP)
        jointPosition[*rx_id] = 0;
        jointPosition[*rx_id] |= ( (long)rxCANmsg.data[3] << 16) & 0x003F0000;
        jointPosition[*rx_id] |= ( (long)rxCANmsg.data[4] << 8 ) & 0x0000FF00;
        jointPosition[*rx_id] |= ( (long)rxCANmsg.data[5] ) & 0x000000FF;

        if (jointPosition[*rx_id] & 0x00200000)     // Check if received data value is negative
            jointPosition[*rx_id] |= 0xFFC00000;    // Convert signed 22-bit value to signed 32-bit value
        break;

    case 0:  // GroupID 0: Firmware request (GET)
        *rx_property = -(rxCANmsg.data[0] & 0x7F);  // A negative (or zero) property means 'get property'
        *rx_value = 0;
        break;

    default:
        LED8_ON;
        printString("ERROR: Invalid message format");
        return(1);
    }

    return (0);
} // end parseCANmsgPuck

int setTorquePuck(int channel, int puckID, float value)
{
    int value_scaled = 0;

    // Convert torque value from SI units to Puck units
    value_scaled = convertUnits_SI_to_Puck(puckID, T, value);

    // Prepare CAN message frame and send it to motor specified by 'puckID'
    setPropertyPuck(channel, puckID, T, value_scaled);

    return(0);
} // end setTorquePuck
```

```
float getPositionPuck(int channel, int puckID, char n)
{
    static char kLW = 0, kRW = 0;
    static long posnDataLW[11] = {0}, posnDataRW[11] = {0};
    char i = 0, j = 0;
    long position = 0;                  // Holds encoder position [counts/rev]
    long position_cumulative = 0;       // Holds cumulative encoder position [CTS/rev]
    long position_sum = 0;
    float position_filtered = 0;
    float position_scaled = 0;          // Holds cumulative encoder position [radians]

    // Recursive moving average filter
    if (puckID == L_WHEEL_ID)
    {
        // Execute this IF statement only after posnDataLW[] already contains n-points
        if (PuckLWinitialized == TRUE)
        {
            // Get 32-bit motor encoder angular position in encoder counts/rev
            getPropertyPuck(channel, puckID, P, &position);

            // Detect and correct for zero crossing (i.e. motor encoder completes full revolution)
            position_cumulative = encoderZeroCrossing(puckID, position);

            // Update n-element array, posnDataLW[], with the latest LW encoder measurements
            posnDataLW[kLW] = position_cumulative;  // posnDataLW[kLW] stores the latest encoder values
            kLW++;                      // Increment kLW counter
            if (kLW == n)   {kLW = 0;}  // Reset kLW counter when number of points stored equals n
        }

        // Before the moving average filter can be properly implemented, n points are needed
        if (PuckLWinitialized == FALSE)
        { // If this function is being called for the first time, collect n-points from LW Puck
            for (kLW = 0; kLW < n; kLW++)
            {
                // Get 32-bit motor encoder angular position in encoder counts/rev
                getPropertyPuck(channel, puckID, P, &position);

                // Detect and correct for zero crossing (i.e. motor encoder completes full revolution)
                position_cumulative = encoderZeroCrossing(puckID, position);

                posnDataLW[kLW] = position_cumulative;  // Store latest encoder measurements
            }
            kLW = 0;                        // Reset kLW counter
            PuckLWinitialized = TRUE;       // Raise flag indicating LW Puck has been initialized
        }
        // Calculate the sum of the lastest n-points
        for (j = 0; j < n; j++) { position_sum = position_sum + posnDataLW[j]; }
    }

    if (puckID == R_WHEEL_ID)
    {
        // Execute this IF statement only after posnDataRW[] already contains n-points
        if (PuckRWinitialized == TRUE)
        {
            // Get 32-bit motor encoder angular position in encoder counts/rev
            getPropertyPuck(channel, puckID, P, &position);

            // Detect and correct for zero crossing (i.e. motor encoder completes full revolution)
            position_cumulative = encoderZeroCrossing(puckID, position);

            // Update n-element array, posnDataRW[], with the latest RW encoder measurements
            posnDataRW[kRW] = position_cumulative;  // posnDataRW[kRW] stores the latest encoder values
            kRW++;                      // Increment kRW counter
            if (kRW == n)   {kRW = 0;}  // Reset kRW counter when number of points stored equals n
        }

        // Before the moving average filter can be properly implemented, n points are needed
        if (PuckRWinitialized == FALSE)
        { // If this function is being called for the first time, collect n-points from RW Puck
            for (kRW = 0; kRW < n; kRW++)
            {
                // Get 32-bit motor encoder angular position in encoder counts/rev
                getPropertyPuck(channel, puckID, P, &position);

                // Detect and correct for zero crossing (i.e. motor encoder completes full revolution)
                position_cumulative = encoderZeroCrossing(puckID, position);

                posnDataRW[kRW] = position_cumulative;  // Store latest encoder measurements
            }
            kRW = 0;                        // Reset kRW counter
            PuckRWinitialized = TRUE;       // Raise flag indicating RW Puck has been initialized
        }
        // Calculate the sum of the lastest n-points
        for (j = 0; j < n; j++) { position_sum = position_sum + posnDataRW[j]; }
    }

    // Calculate average of latest n-points
    position_filtered = position_sum/n;
```

```
    // Convert motor position value from puck units [CTS/rev] to SI units [radians]
    position_scaled = convertUnits_Puck_to_SI(puckID, P, position_filtered);

    // Store motor position in structure containing processed Puck data in SI units [radians]
    PUCKdataSI[puckID-1].theta_XXrad = position_scaled;

    return(position_scaled);
} // end getPositionPuck

/*=============================================================================================================*
    encoderZeroCrossing() detects and corrects for zero crossing (i.e. motor encoder completes full
        revolution).

    Arguments:
        puckID      Puck ID number of the left or right wheel motor (see support.h).
        position    current encoder count received from Puck specified by 'puckID'.

    Static Arguments (globally defined in main.c):
        posnXX_countold     stores old encoder count; 'XX' denotes Left Wheel (LW) or Right Wheel (RW).
        posnXX_old          stores old cumulative encoder count; 'XX' denotes (LW) or (RW).

    Returns:
        posnXX      the cumulative encoder count; 'XX' denotes (LW) or (RW).

    Description:
        This function uses the latest and previous Puck encoder counts to determine whether the encoder
        count has made a full revolution and whether it requires correction.

        Zero crossing can be tracked by observing when the encoder count makes the transition from the
        predefined counts per revolution (Default = 4096) to zero and vice versa. A 'posnXX_delta' less
        than -512 implies that the wheel has completed a full forward revolution while a 'posnXX_delta'
        greater than 512 implies that the wheel has completed a full backward revolution.

        Note that if the sampling rate is not high enough to sample the encoder readings more than 16
        times per revolution (Nyquist frequency fs/2 > f compensated), the difference between the latest
        and previous encoder count would be larger than 512 counts which could result in a false zero
        crossing (i.e. the software thinks that the wheel has completed a full revolution which may not
        be true).

        To prevent another false zero crossing (i.e. encoder count crosses zero at the initial starting
        position without first making a full revolution), the condition that the platform must have
        traveled a distance of at least 0.2 m has been applied. Note that 0.2 m is a value that was
        selected to be less than the total distance traveled if only one wheel had completed a full
        revolution while the other remained stationary.

        IMPORTANT!
        The structure PUCKconvFactors[puckID-1].counts_per_rev holds the current configured encoder count
        per revolution Puck property. If the Puck property CTS is configured to something other than the
        factory default (CTS = 4096) then the value 512 must be changed to conform to the new CTS property
        value.
 *=============================================================================================================*/
long encoderZeroCrossing(int puckID, long position)
{
    static long posnLW_countold = 0, posnLW_old = 0, posnRW_countold = 0, posnRW_old = 0;
    long posnLW = 0;                // Stores the cumulative encoder count
    long posnLW_count = 0;          // Stores the current encoder count
    long posnLW_delta = 0;          // Stores the difference between current and old encoder count
    long posnRW = 0;
    long posnRW_count = 0;
    long posnRW_delta = 0;

    // If Button 2 is pressed, reset Pucks and clear cumulative encoder counters
    if (resetEncoderCount == TRUE)
    {
        // Clear encoder counters
        posnLW_countold = 0; posnLW_old = 0; posnRW_countold = 0; posnRW_old = 0;
        resetEncoderCount = FALSE;      // Clear reset encoder counters flag
    }

    if(puckID == L_WHEEL_ID)
    { // Detecting and correcting for zero crossing in the Left Wheel (LW) motor encoder
        posnLW_count = position;                        // Store current encoder position in count/rev
        posnLW_delta = posnLW_count - posnLW_countold;  // Calculate the difference between samples

        // Detect and correct for zero crossing, zero crossings will be apparent with large deltas
        if(posnLW_delta < -512 && PUCKdataSI[0].x > 0.2)
        { // If the difference is less than -512 (bigger than expected), then correct
            posnLW = posnLW_old + posnLW_delta + ONE_REV_LW;
        }
        else if(posnLW_delta > 512 && PUCKdataSI[0].x < 0.2)
        { // If the difference is greater than 512 (bigger than expected), then correct
            posnLW = posnLW_old + posnLW_delta - ONE_REV_LW;
        }
        else
        { // If delta is small, then just update position normally with delta
            posnLW = posnLW_old + posnLW_delta;
        }
```

```
        posnLW_countold = posnLW_count;     // Update old encoder count
        posnLW_old = posnLW;                // Update old cumulative encoder count

        return(posnLW);                     // Return current cumulative encoder count
    }

    else if(puckID == R_WHEEL_ID)
    { // Detecting and correcting for zero crossing in the Right Wheel (RW) motor encoder
        posnRW_count = position;                        // Store current encoder position in count/rev
        posnRW_delta = posnRW_count - posnRW_countold;  // Calculate the difference between samples

        // Detect and correct for zero crossing, zero crossings will be apparent with large deltas
        if(posnRW_delta < -512 && PUCKdataSI[0].x > 0.2)
        { // If the difference is less than -512 (bigger than expected), then correct
            posnRW = posnRW_old + posnRW_delta + ONE_REV_RW;
        }
        else if(posnRW_delta > 512 && PUCKdataSI[0].x < 0.2)
        { // If the difference is greater than 512 (bigger than expected), then correct
            posnRW = posnRW_old + posnRW_delta - ONE_REV_RW;
        }
        else
        { // If delta is small, then just update position normally with delta
            posnRW = posnRW_old + posnRW_delta;
        }
        posnRW_countold = posnRW_count;     // Update old encoder count
        posnRW_old = posnRW;                // Update old cumulative encoder count

        return(posnRW);                     // Return current cumulative encoder count
    }
    else
    {
        printString("ERROR: Invalid puckID");
        return(0);
    }
} // encoderZeroCrossing

float convertUnits_Puck_to_SI(int puckID, int property, long value)
{
    float value_scaled = 0;

    switch(property)
    {
        case TEMP:  // Puck temperature
            value_scaled = (float) (value);
            break;
        case THERM: // Motor temperature
            value_scaled = (float) (value);
            break;
        case T:     // Motor torque [milliamps to Nm]
            value_scaled = (float) (value)/PUCKconvFactors[puckID-1].milliAmp_per_Nm;
            break;
        case V:     // Motor velocity [counts/milliseconds to m/s]
            value_scaled = (float) (value);
            break;
        case P:     // 32-bit motor position [encoder counts to radians]
            value_scaled = (float) (value)*((2*PI)/PUCKconvFactors[puckID-1].counts_per_rev);
            break;
        default:
            LED7_ON;
            printf("Invalid Puck property.");
            break;
    }

    return(value_scaled);
} // end convertUnits_Puck_to_SI

int convertUnits_SI_to_Puck(int puckID, int property, float value)
{
    int value_scaled = 0;

    switch(property)
    {
        case T:     // Motor torque [Nm to milliamps]
            value_scaled = (float) (value)*PUCKconvFactors[puckID-1].milliAmp_per_Nm;
            value_scaled = Border(value_scaled, -8191, 8191);   // Bound the torque commands
            break;
        case V:     // Motor velocity [m/s to counts/milliseconds]
            value_scaled = (int) (value);
            break;
        case P:     // 32-bit motor position [radians to encoder counts]
            value_scaled = (int) (value)/((2*PI)/PUCKconvFactors[puckID-1].counts_per_rev);
            break;
        default:
            LED7_ON;
            printString("Invalid Puck property.");
            break;
    }
```

```
    return(value_scaled);
} // end convertUnits_SI_to_Puck

void checkTemp(int puckID)
{
    long temperature;

    wakePuck(0, puckID);
    getPropertyPuck(0, puckID, TEMP, &temperature);
    printf("\n TEMP = %ld", temperature);

    // Check if temperature is less than 15 degC or greater than 60 degC
    if(temperature < 15 || temperature > 60)
    {
        LED5_ON;
        printString("Puck temperature is outside safe limits. Stopping wheels.");
        stopWheels();
    }
} // end checkTemp

void stopWheels(void)
{
    // Set commanded torque to 0
    setPropertyPuck(0, L_WHEEL_ID, T, 0);
    setPropertyPuck(0, R_WHEEL_ID, T, 0);
} // end stopWheels

void setDefaultPropertiesPuck(int puckID)
{
    setPropertySlowPuck(0, puckID, IKCOR, 1638);    // Factory default IKCOR = 1638
    setPropertySlowPuck(0, puckID, IKP, 8192);      // Factory default IKP = 8192
    setPropertySlowPuck(0, puckID, IKI, 3276);      // Factory default IKI = 3276
    setPropertySlowPuck(0, puckID, IPNM, 2755);     // Factory default IPNM = 2755
    setPropertySlowPuck(0, puckID, POLES, 12);      // Factory default POLES = 12
    //setPropertySlowPuck(0, puckID, GRPA, 0);      // Factory default GRPA = NONE
    //setPropertySlowPuck(0, puckID, GRPB, 1);      // Factory default GRPB = NONE
    //setPropertySlowPuck(0, puckID, GRPC, 4);      // Factory default GRPC = NONE
    //setPropertySlow(0,newID,JIDX,0,targID-3);
    //setPropertySlow(0,newID,PIDX,0,((targID-1)%4)+1);

    //setPropertySlowPuck(0,newID,SAVE,0,-1);       // Save new default properties
} // end setDefaultPropertiesPuck

void getConversionFactorsPuck(int puckID)
{
    long reply;

    // Get conversion factors from specified puck ID
    getPropertyPuck(0, puckID, CTS, &reply);
    PUCKconvFactors[puckID-1].counts_per_rev = reply;
    getPropertyPuck(0, puckID, IPNM, &reply);
    PUCKconvFactors[puckID-1].milliAmp_per_Nm = reply;
} // end getConversionFactorsPuck

void resetPucks(void)
{
    // Raise flag to clear encoder count in encoderZeroCrossing()
    resetEncoderCount = TRUE;   // Boolean value globally defined in main.c

    // Clear IMUinitialized flag to allow software to collect a new set of n-points for gyro filter
    IMUinitialized = FALSE;     // See get_IMUgyro() in IMU.c

    // Clear PuckXXinitialized flag to allow software to collect a new set of n-points for encoder filter
    PuckLWinitialized = FALSE;      // See getPositionPuck() in Puck.c
    PuckRWinitialized = FALSE;      // See getPositionPuck() in Puck.c

    // Reset Pucks
    setPropertyPuck(0, L_WHEEL_ID, STAT, STATUS_RESET);
    setPropertyPuck(0, R_WHEEL_ID, STAT, STATUS_RESET);
    delay_ms(500);              // Wait for the Pucks' reset routine to complete

    // Wake up pucks and configure them to TORQUE mode
    config_pucks(MODE_TORQUE);
} // end resetPuck
```

F-9: SPI.c

```
/*****************************************************************************************************
Name: SPI.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used to configure and initialize:
    SPI3 master for transmission/reception with two 6-channel delta sigma A/D converters (MCP3903),
    SPI4 master for transmission/reception with a 4-DOF inertial measurement unit (ADIS16300).

    Also contains driver functions used to send and read SPI messages.
*****************************************************************************************************/

#include <p32xxxx.h>
#include <plib.h>
#include "support.h"

void init_SPI3master(void)
{
/*===================================================================================================*
    Step 1: Disable the SPI interrupts in the respective IECx register.
    Step 2: Stop and reset the SPI module by clearing the ON bit.
    Step 3: Clear the receive buffer.
    Step 4: Clear the ENHBUF bit (SPIxCON<16>) if using Standard Buffer mode or set the bit if using
            Enhanced Buffer mode.
    Step 5: If SPI interrupts are not going to be used, skip this step and continue to step 5.
            Otherwise the following additional steps are performed:
             a) Clear the SPIx interrupt flags/events in the respective IFSx register.
             b) Write the SPIx interrupt priority and subpriority bits in the respective IPCx register.
             c) Set the SPIx interrupt enable bits in the respective IECx register.
    Step 6: Write the Baud Rate register, SPIxBRG.
    Step 7: Clear the SPIROV bit (SPIxSTAT<6>).
    Step 8: Write the desired settings to the SPIxCON register with MSTEN (SPIxCON<5>) = 1.
    Step 9: Enable SPI operation by setting the ON bit (SPIxCON<15>).

    References:
        Section 23. SPI - S23.3.3.1: Master Mode Operation [p. 18]
        Section 23. SPI - S23.3.3.7: SPI Master Mode Clock Frequency [p. 29]
 *===================================================================================================*/
    int temp;

    IEC0bits.SPI3TXIE = 0;      // Step 1: Disable SPI3 TX interrupt
    IEC0bits.SPI3RXIE = 0;      // Step 1: Disable SPI3 RX interrupt
    IEC0bits.SPI3EIE = 0;       // Step 1: Disable SPI3 error interrupt
    SPI3CONbits.ON = 0;         // Step 2: Stop and reset the SPI module by clearing the ON bit
    temp = SPI3BUF;             // Step 3: Clear the receive buffer
    SPI3CONbits.ENHBUF = 0;     // Step 4: Clear the Enhanced Buffer Enable bit

    IFS0bits.SPI3TXIF = 0;      // Step 5a: Clear SPI3 TX interrupt flag
    IFS0bits.SPI3RXIF = 0;      // Step 5a: Clear SPI3 RX interrupt flag
    IFS0bits.SPI3EIF = 0;       // Step 5a: Clear SPI3 error interrupt flag
    IPC6bits.SPI3IP = 4;        // Step 5b: Set SPI3 interrupt priority level to 4
    IPC6bits.SPI3IS = 1;        // Step 5b: Set SPI3 interrupt sub-priority level to 1
    IEC0bits.SPI3TXIE = 1;      // Step 5c: Enable SPI3 TX interrupt
    IEC0bits.SPI3RXIE = 1;      // Step 5c: Enable SPI3 RX interrupt
    IEC0bits.SPI3EIE = 1;       // Step 5c: Enable SPI3 error interrupt

/* SPI Master Mode Clock Frequency (SCKx):
    Peripheral Bus Clock,           Fpb = 80 MHz
    Desired SPI Clock Frequency,    Fsck = 1 MHz

    Equation 23-1: [p. 29]
    Fsck = Fpb/[2*(SPIxBRG+1)]
    SPIxBRG = [Fpb/(2*Fsck)]-1 = [80MHz/(2*1MHz)]-1 = 39
*/
    SPI3BRG = 39;               // Step 6: Set SPI clock frequency to 1 MHz

/* REGISTER 17-2: SPIxSTAT: SPIx STATUS AND CONTROL REGISTER [p. 268] */
    SPI3STATbits.SPIROV = 0;    // Step 7: Clear the Receive Overflow Flag bit

/* REGISTER 17-1: SPIxCON: SPI CONTROL REGISTER [p. 212] */
    SPI3CONbits.SIDL = 0;       // Step 8: Continue module operation in Idle mode
    SPI3CONbits.DISSDO = 0;     // Step 8: SDOx pin is controlled by the module
    SPI3CONbits.MODE16 = 0;     // Step 8: Communication is 8-bit wide
    SPI3CONbits.SMP = 1;        // Step 8: Input data is sampled at the end of data output time
    SPI3CONbits.CKE = 0;        // Step 8: Serial output data changes on transition from
                                        // idle clock state to active clock state
    SPI3CONbits.SSEN = 0;       // Step 8: SSx pin is not used by the module
    SPI3CONbits.CKP = 1;        // Step 8: Idle state for clock is a high-level;
                                        // active state is a low-level
    SPI3CONbits.MSTEN = 1;      // Step 8: Master mode enabled
    SPI3CONbits.FRMEN = 0;      // Step 8: Framed SPIx support is disabled

    SPI3CONbits.ON = 1;         // Step 9: Enable SPI peripheral

    //LATCbits.LATC14 = 1;      // Set initial state of IMU slave select line to high
```

```
    LATDbits.LATD11 = 1;        // Set initial state of ADC1 slave select line to high
    LATDbits.LATD9 = 1;         // Set initial state of ADC2 slave select line to high
} // end init_SPI3master

void init_SPI4master(void)
{
    int temp;

    IEC1bits.SPI4TXIE = 0;      // Step 1: Disable SPI4 TX interrupt
    IEC1bits.SPI4RXIE = 0;      // Step 1: Disable SPI4 RX interrupt
    IEC1bits.SPI4EIE = 0;       // Step 1: Disable SPI4 error interrupt
    SPI4CONbits.ON = 0;         // Step 2: Stop and reset the SPI module by clearing the ON bit
    temp = SPI4BUF;             // Step 3: Clear the receive buffer
    SPI4CONbits.ENHBUF = 0;     // Step 4: Clear the Enhanced Buffer Enable bit

    IFS1bits.SPI4TXIF = 0;      // Step 5a: Clear SPI4 TX interrupt flag
    IFS1bits.SPI4RXIF = 0;      // Step 5a: Clear SPI4 RX interrupt flag
    IFS1bits.SPI4EIF = 0;       // Step 5a: Clear SPI4 error interrupt flag
    IPC8bits.SPI4IP = 4;        // Step 5b: Set SPI4 interrupt priority level to 4
    IPC8bits.SPI4IS = 1;        // Step 5b: Set SPI4 interrupt sub-priority level to 1
    IEC1bits.SPI4TXIE = 1;      // Step 5c: Enable SPI4 TX interrupt
    IEC1bits.SPI4RXIE = 1;      // Step 5c: Enable SPI4 RX interrupt
    IEC1bits.SPI4EIE = 1;       // Step 5c: Enable SPI4 error interrupt

/* SPI Master Mode Clock Frequency (SCKx):
    Peripheral Bus Clock,           Fpb = 80 MHz
    Desired SPI Clock Frequency,    Fsck = 1 MHz

    Equation 23-1: [p. 29]
    Fsck = Fpb/[2*(SPIxBRG+1)]
    SPIxBRG = [Fpb/(2*Fsck)]-1 = [80MHz/(2*1MHz)]-1 = 39
*/
    SPI4BRG = 39;               // Step 6: Set SPI clock frequency to 1 MHz

/* REGISTER 17-2: SPIxSTAT: SPIx STATUS AND CONTROL REGISTER [p. 268] */
    SPI4STATbits.SPIROV = 0;    // Step 7: Clear the Receive Overflow Flag bit

/* REGISTER 17-1: SPIxCON: SPI CONTROL REGISTER [p. 212] */
    SPI4CONbits.SIDL = 0;       // Step 8: Continue module operation in Idle mode
    SPI4CONbits.DISSDO = 0;     // Step 8: SDOx pin is controlled by the module
    SPI4CONbits.MODE16 = 1;     // Step 8: Communication is word-wide (16 bits)
    SPI4CONbits.SMP = 1;        // Step 8: Input data is sampled at the end of data output time
    SPI4CONbits.CKE = 0;        // Step 8: Serial output data changes on transition from
                                        // idle clock state to active clock state
    SPI4CONbits.SSEN = 0;       // Step 8: SSx pin is not used by the module
    SPI4CONbits.CKP = 1;        // Step 8: Idle state for clock is a high-level;
                                        // active state is a low-level
    SPI4CONbits.MSTEN = 1;      // Step 8: Master mode enabled
    SPI4CONbits.FRMEN = 0;      // Step 8: Framed SPIx support is disabled

    SPI4CONbits.ON = 1;         // Step 9: Enable SPI peripheral

    LATBbits.LATB15 = 1;        // Set initial state of IMU slave select line to high
} // end init_SPI4master

short sendSPImsg(char device, short SPItx_data)
{
    short temp;

    switch(device)
    {
    case IMU_ID:
        IEC1bits.SPI4TXIE = 1;              // Enable SPI4 TX interrupt
        IEC1bits.SPI4RXIE = 1;              // Enable SPI4 RX interrupt
        IEC1bits.SPI4EIE = 1;               // Enable SPI4 error interrupt

        LATBbits.LATB15 = 0;                // Lower IMU slave select line to initiate data exchange
        SPI4BUF = SPItx_data;               // Write data to TX buffer
        while(SPI4STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI4RXB) is full
        temp = SPI4BUF;                     // Dummy read of the SPI3BUF register to clear the SPIRBF flag
        LATBbits.LATB15 = 1;                // Raise IMU slave select line to terminate data exchange
        break;

    case ADC1_ID:
        IEC0bits.SPI3TXIE = 1;              // Enable SPI3 TX interrupt
        IEC0bits.SPI3RXIE = 1;              // Enable SPI3 RX interrupt
        IEC0bits.SPI3EIE = 1;               // Enable SPI3 error interrupt

        SPI3BUF = SPItx_data;               // Write data to TX buffer
        while(SPI3STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI3RXB) is full
        temp = SPI3BUF;                     // Dummy read of the SPI3BUF register to clear the SPIRBF flag
        break;

    case ADC2_ID:
        IEC0bits.SPI3TXIE = 1;              // Enable SPI3 TX interrupt
        IEC0bits.SPI3RXIE = 1;              // Enable SPI3 RX interrupt
        IEC0bits.SPI3EIE = 1;               // Enable SPI3 error interrupt
```

```
        SPI3BUF = SPItx_data;               // Write data to TX buffer
        while(SPI3STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI3RXB) is full
        temp = SPI3BUF;                     // Dummy read of the SPI3BUF register to clear the SPIRBF flag
        break;

    default:
        printString("Specified 'device' value does not match any existing device IDs.");
        break;
    }

    return(0);
} // end sendSPImsg

short readSPImsg(char device)
{
    short rx_data;

    switch(device)
    {
    case IMU_ID:
        IEC1bits.SPI4TXIE = 1;              // Enable SPI4 TX interrupt
        IEC1bits.SPI4RXIE = 1;              // Enable SPI4 RX interrupt
        IEC1bits.SPI4EIE = 1;               // Enable SPI4 error interrupt

        LATBbits.LATB15 = 0;                // Lower IMU slave select line to initiate data exchange
        SPI4BUF = 0;                        // Initiate data transfer by writing dummy data to TX buffer
        while(SPI4STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI4RXB) is full
        rx_data = SPI4BUF;                  // Read RX buffer
        LATBbits.LATB15 = 1;                // Raise IMU slave select line to terminate data exchange
        break;

    case ADC1_ID:
        IEC0bits.SPI3TXIE = 1;              // Enable SPI3 TX interrupt
        IEC0bits.SPI3RXIE = 1;              // Enable SPI3 RX interrupt
        IEC0bits.SPI3EIE = 1;               // Enable SPI3 error interrupt

        SPI3BUF = 0;                        // Initiate data transfer by writing dummy data to TX buffer
        while(SPI3STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI3RXB) is full
        rx_data = SPI3BUF;                  // Read RX buffer
        break;

    case ADC2_ID:
        IEC0bits.SPI3TXIE = 1;              // Enable SPI3 TX interrupt
        IEC0bits.SPI3RXIE = 1;              // Enable SPI3 RX interrupt
        IEC0bits.SPI3EIE = 1;               // Enable SPI3 error interrupt

        SPI3BUF = 0;                        // Initiate data transfer by writing dummy data to TX buffer
        while(SPI3STATbits.SPIRBF == 0);    // Wait until the receive buffer (SPI3RXB) is full
        rx_data = SPI3BUF;                  // Read RX buffer
        break;

    default:
        printString("Specified 'device' value does not match any existing device IDs.");
        break;
    }

    return(rx_data);
} // end readSPImsg
```

F-10: support.c

```
/*********************************************************************************************************
Name: support.c
Creation Date: 09/12/2013
Authors: Airton R. da Silva Jr.
Comments: Functions used to configure & initialize internal clock rate, I/O pins, sampling rate,
    16/32-bit counters, delay_ms, delay_us, and hardware operation checkpoints.
*********************************************************************************************************/

#include <p32xxxx.h>
#include <plib.h>
#include "math.h"
#include "support.h"

/*=================================================================================================*
    config_clock() configures and initializes internal clock rate.

    Description:
        PIC32MX Family Ref. Manual v1: Section 06. Oscillators [p. 19]
        Given: Desired clock rate is 80 MHz from an 8 MHz internal oscillator.
        The input frequency to the PLL must be >= 4 MHz and <= 5 MHz.

        From the FPLLIDIV field in the DEVCFG2 register description,
        an input divisor of 2 is available: 8/2 = 4 MHz.
        This fulfills the PLL input requirement.

        The desired net multiplier is 80/4 = 20 MHz.
        Locate the row in Table 6-5 that corresponds to a net multiplier value of 20.
        From the table, the PLL multiplier value is 20 and the output divider value is 1.

        See FIGURE 8-1: OSCILLATOR BLOCK DIAGRAM [p. 141]
        Fin = 8 MHz internal oscillator
        FPLLMUL = 20
        FPLLIDIV = 2
        FPLLODIV = 1
        Fosc = Fin/FPLLIDIV*FPLLMUL/FPLLODIV = (8MHz Crystal)/2*20/1 = 80 MHz

        REGISTER 28-3: DEVCFG2: DEVICE CONFIGURATION WORD 2 [p. 344]
 *=================================================================================================*/
void config_clock(void)
{
    // Configure PLL prescaler, PLL postscaler, PLL divisor
    #pragma config FPLLIDIV = DIV_2     // PLL Input Divider: 8/2 = 4 MHz
    #pragma config FPLLMUL = MUL_20     // PLL Multiplier: 4*20 = 80 MHz
    #pragma config FPLLODIV = DIV_1     // PLL Output Divider 80/1 = 80 MHz
    OSCTUNbits.TUN = 0;     // Tune FRC osc to center freq.; Osc runs at minimal freq (8 MHz)

    // Wait for Clock switch to occur
    while (OSCCONbits.COSC != 0b001);

    // Wait for PLL to lock
    while(OSCCONbits.SLOCK != 1) {};
} // end config_clock

/*=================================================================================================*
    config_pins() configures and initializes I/O pins.

    Pin Overview Map:
        P1 :RE5 - (O|D) LED1
        P2 :RE6 - (O|D) LED2
        P3 :RE7 - (O|D) LED3
        P4 :RG6 - (O|D) LED4
        P5 :RG7 - (I|D) UART3 Receive (U3RX)
        P6 :RG8 - (O|D) UART3 Transmit (U3TX)
        P7 :MCLR
        P8 :RG9 - (O|D) LED5
        P9 :V_SS1
        P10:V_DD1
        P11:RB5 - (O|D) SD5V
        P12:AN4 - (I|A) BM
        P13:FREE
        P14:FREE
        P15:RB1 - (O|D) PICkit 3 (PGEC1)
        P16:RB0 - (O|D) PICkit 3 (PGED1)
        P17:FREE
        P18:FREE
        P19:AV_DD
        P20:AV_SS
        P21:RB8 - (O|D) LED6
        P22:RB9 - (O|D) LED7
        P23:RB10 - (I|D) Button 1 (S1)
        P24:RB11 - (I|D) Button 2 (S2)
        P25:V_SS2
        P26:V_DD2
        P27:FREE
```

```
        P28:RB13 - (O|D) ADIS16300: Master Reset Logic Output (RESET)
        P29:RB14 - (O|D) ADIS16300: SPI4 Serial Clock Output (SCK4)
        P30:RB15 - (O|D) ADIS16300: Chip Select (CS) Active-Low bit
        P31:RF4 - (I|D) ADIS16300: SPI4 Serial Data Input (SDI4)
        P32:RF5 - (O|D) ADIS16300: SPI4 Serial Data Output (SDO4)
        P33:FLOATING
        P34:FLOATING
        P35:FLOATING
        P36:FLOATING
        P37:FLOATING
        P38:V_DD3
        P39:RC12 - (O|D) LED8
        P40:RC15 - (O|D) Output Clock Enabled
        P41:V_SS3
        P42:RD8 - (O|D) LED9
        P43:RD9 - (O|D) MCP3903 #2: Chip Select (CS) Active-Low bit
        P44:RD10 - (I|D) MCP3903 #2: Data Ready Signal Input for channels pair A (DRA)
        P45:RD11 - (O|D) MCP3903 #1: Chip Select (CS) Active-Low bit
        P46:RD0 - (O|D) MCP3903 #1 & #2: OSC1
        P47:RC13 - (I|D) MCP3903 #1: Data Ready Signal Input for channels pair A (DRA)
        P48:RC14 - (O|D) ADIS16300: Chip Select (CS) Active-Low bit
        P49:RD1 - (O|D) ADIS16300 + MCP3903 #1 & #2: SPI3 Serial Clock Output (SCK3)
        P50:RD2 - (I|D) ADIS16300 + MCP3903 #1 & #2: SPI3 Serial Data Input (SDI3)
        P51:RD3 - (O|D) ADIS16300 + MCP3903 #1 & #2: SPI3 Serial Data Output (SDO3)
        P52:RD4 - (O|D) ADIS16300 + MCP3903 #1 & #2: Master Reset Logic Output (RESET)
        P53:RD5 - (O|D) LED10
        P54:RD6 - (O|D) LED11
        P55:RD7 - (O|D) LED12
        P56:V_CAP
        P57:V_DD4
        P58:RF0 - (I|D) CAN1 Receive (C1RX)
        P59:RF1 - (O|D) CAN1 Transmit (C1TX)
        P60:FREE
        P61:FREE
        P62:FREE
        P63:FREE
        P64:FREE

        Legend: (I)nput | (O)utput | (A)nalog | (D)igital

    References:
        Built-in functions and macros used to configure I/O ports referenced from:
        PIC32 Peripheral Libraries for MPLAB C32 Compiler [Ch. 10, p. 103]
 *=============================================================================================================*/
void config_pins(void)
{
    // TABLE 4-25: PORTB REGISTER MAP [p. 96]
    TRISB = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    PORTSetPinsDigitalIn(IOPORT_B,BIT_10|BIT_11);   // Set RB10,11 to digital inputs
    LATB = 0x00000000;                              // All pins set to zero
    ODCB = 0x00000000;                              // All pins set NOT open drain
    // TABLE 4-26: PORTC REGISTER MAP [p. 97]
    TRISC = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    PORTSetPinsDigitalIn(IOPORT_C,BIT_13);          // Set RC13 to digital input
    LATC = 0x00000000;                              // All pins set to zero
    ODCC = 0x00000000;                              // All pins set NOT open drain
    // TABLE 4-28: PORTD REGISTER MAP [p. 98]
    TRISD = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    PORTSetPinsDigitalIn(IOPORT_D,BIT_2|BIT_10);    // Set RD2,10 to digital input
    LATD = 0x00000000;                              // All pins set to zero
    ODCD = 0x00000000;                              // All pins set NOT open drain
    // TABLE 4-30: PORTE REGISTER MAP [p. 99]
    TRISE = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    LATE = 0x00000000;                              // All pins set to zero
    ODCE = 0x00000000;                              // All pins set NOT open drain
    // TABLE 4-32: PORTF REGISTER MAP [p. 100]
    TRISF = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    PORTSetPinsDigitalIn(IOPORT_F,BIT_0|BIT_4);     // Set RF0,4 to digital input
    LATF = 0x00000000;                              // All pins set to zero
    ODCF = 0x00000000;                              // All pins set NOT open drain
    // TABLE 4-34: PORTG REGISTER MAP [p. 101]
    TRISG = 0x00000000;                             // Data direction: 1 = input | 0 = output bit
    PORTSetPinsDigitalIn(IOPORT_G,BIT_7);           // Set RG7 to digital input
    LATG = 0x00000000;                              // All pins set to zero
    ODCG = 0x00000000;                              // All pins set NOT open drain

    // TABLE 4-16: ADC REGISTER MAP [p. 85]
    AD1PCFG = 0xFFFFFFFF;                           // A/D register: 1 = digital | 0 = analog
    PORTSetPinsAnalogIn(IOPORT_B,BIT_4);            // Set AN4 to analog input
} // end config_pins

/*=============================================================================================================*
    init_samptime() initializes Timer1 to enforce desired sampling rate of code iteration in main.c.

    Description:
        This function is used to initialize and define the sampling rate of software execution.

        Timer1 configuration:
```

```
            Timer Input Clock Prescaler Value,  1/PV = 1:64 = 1/64
            Oscillator Frequency,               F_OSC = 80 MHz
            Elapsed time per TMR1 count,        P = 1/F_OSC = 12.5 ns per count
            Sampling Time,                      t_s = 1 ms or 1000 Hz
                                                t_s = (PR1 + 1) * PV * P
            Period Register,
            PR1 = t_s/(PV * P) - 1 = (1 ms)/(64 * 12.5 ns) - 1 = 1250 - 1
            PR1 = 1250 - 1

            Proofcheck: t_s = (1249 + 1) * 64 * (12.5 ns) = 1 ms (1000 Hz)

        Sampling Times:
            Set PR1 = 1250 - 1  for 1000  Hz    (This sampling rate causes moderate rattling in motors)
                PR1 = 2500 - 1  for 500   Hz    (This sampling rate causes minor rattling in motors)
                PR1 = 6250 - 1  for 200   Hz
                PR1 = 8333 - 1  for 150   Hz    (Tested max for MATLAB's 'static' data logging function)
                PR1 = 12500 - 1 for 100   Hz
                PR1 = 62500 - 1 for 20    Hz    (Tested max for MATLAB's 'dynamic' data logging function)
    References:
        Section 14. Timers: S14.3.4.2 16-Bit Synchronous Counter Initialization Steps [p. 14]
        Programming 32-bit MCUs in C - Exploring the PIC32 [p. 104]

        IMPORTANT!
        The sampling rate specified in init_samptime() MUST MATCH the sampling rate defined by SAMP_RATE
        variable which can be located inside support.h at or around line 52. Currently, SAMP_RATE = 0.01.
        Thus, if the firmware sampling rate is modified then SAMP_RATE must be changed accordingly.
 *=====================================================================================================*/
void init_samptime(void)
{
    T1CONbits.ON = 0;           // Disable timer
    T1CONbits.TCS = 0;          // Timer1 Clock Source set to be the Internal PBCLK Source
    T1CONbits.TCKPS = 0b10;     // Timer1 input clock prescaler set to 1:64

    TMR1 = 0;                   // Clear timer register
    PR1 = 12500 - 1;            // Set the period register

    // Initialize Timer1 interrupt control bits
    mT1ClearIntFlag();          // Clear Timer1 interrupt flag
    mT1SetIntPriority(7);       // Set Timer1 interrupt priority level to 7
    mT1SetIntSubPriority(1);    // Set Timer1 sub-priority level to 1
    mT1IntEnable(1);            // Enable Timer1 interrupt source

    T1CONbits.ON = 1;           // Enable timer
} //end init_samptime

/*=====================================================================================================*
    delay_ms() delays program execution for t milliseconds (tolerance = +/- 0.25%).

    Description:
        Timer2 configuration:
            Timer Input Clock Prescaler Value,  1/PV = 1:2 = 1/2
            Oscillator Frequency,       sysCLK = F_OSC = 80 MHz

            TMR2 ticks 80 million times per second with 1:1 prescale value
            A minimum prescale value of 1:2 is required since TMR2 is capped at 16-bits (65,536)
            TMR2 ticks 40 million times per second with 1:2 prescale value
            (80,000,000 tick/sec)*(1000ms/1s)*(1/2) = 40,000 ticks per ms
 *=====================================================================================================*/
void delay_ms(unsigned int t)
{
    T2CONbits.ON = 0;           // Disable timer
    T2CONbits.TCKPS = 0b001;    // Timer2 input clock prescaler set to 1:2
    T2CONbits.T32 = 0;          // Timer2 and Timer3 form separate 16-bit timers
    T2CONbits.ON = 1;           // Enable 16-bit Timer2

    while (t--)
    {
        TMR2 = 0;                           // Clear timer register
        while(TMR2 < sysCLK / 1000 / 2);    // Wait until TMR2 counter reaches 40,0000
    }
} // end delay_ms

/*=====================================================================================================*
    delay_us() delays program execution for t microseconds (tolerance = +/- 25%).

    Description:
        Timer2 configuration:
            Oscillator Frequency,       sysCLK = F_OSC = 80 MHz
            TMR2 ticks 80 million times per second with 1:1 prescale value
            (80,000,000 tick/sec)*(1,000,000us/1s) = 80 ticks per us
 *=====================================================================================================*/
void delay_us(unsigned int t)
{
    T2CONbits.ON = 0;           // Disable timer
    T2CONbits.TCKPS = 0;        // Timer2 input clock prescaler set to 1:1
    T2CONbits.T32 = 0;          // Timer2 and Timer3 form separate 16-bit timers
    T2CONbits.ON = 1;           // Enable 16-bit Timer2
```

```
    while (t--)
    {
        TMR2 = 0;                           // Clear timer register
        while(TMR2 < sysCLK / 1000000);     // Wait until TMR2 counter reaches 80
    }
} // end delay_us

/*=============================================================================================================*
    delay_ns() delays program execution for t nanoseconds [t must be divisible by 25] (tol. = +/- 25%).

    Description:
        Timer2 configuration:
            Oscillator Frequency,       sysCLK = F_OSC = 80 MHz
            TMR2 ticks 80 million times per second with 1:1 prescale value
            P = 1/(80 MHz) = 12.5 ns per TMR count (1:1 prescale)

        Note that Timer 2 is re-initialized everytime delay_ns() is called. This means that delay_ns() will
        always have minimum delay offset to allow Timer2 to initialize.
 *=============================================================================================================*/
void delay_ns(unsigned int t)
{
    // Input argument t must be divisible by 25 to avoid decimal values
    if(t%25 != 0) {t = t - (t%25);}
    t = 2*t / 25;       // Use t = 2*t / 25 instead of t = t / 12.5 to avoid using floating numbers

    T2CONbits.ON = 0;           // Disable timer
    T2CONbits.TCKPS = 0;        // Timer2 input clock prescaler set to 1:1
    T2CONbits.T32 = 0;          // Timer2 and Timer3 form separate 16-bit timers
    T2CONbits.ON = 1;           // Enable 16-bit Timer2

    while (t--)
    {
        TMR2 = 0;                           // Clear timer register
        while(TMR2 < sysCLK / 80000000);    // Wait until TMR2 counter reaches 1
    }
} // end delay_ns

/*=============================================================================================================*
    init_Timer4() initializes Timer4 and Timer5 to be used as a 32-bit time counter.

    Description:
        This function is used to initialize and define a counter for keeping track of elapsed time between
        state variable sampling in LQR.c.

        Timer4/5 configuration:
            Timer Input Clock Prescaler Value,  1/PV = 1:256 = 1/256
            Oscillator Frequency,               F_OSC = 80 MHz
            Device Operating Frequency,         F_CY = F_OSC/2 = 40 MHz
            Internal Peripheral Bus Clock,      PBCLK = F_OSC / FPBDIV = 40 MHz

            Elapsed time per TMR4/5 count       P = 1/(80 MHz) = 12.5 ns per count (1:1 prescale)
                w/ a prescaler value of 1:256   Pscaled = (12.5 ns/count)*256 = 3.2 us per count
                With a prescaler value of 1:256, each TMR4/5 count corresponds to a 3.2 us time increment.

            Set Period Register (PR4) to max    PR4 = 0xFFFFFFFF = 2^16

        Maximum time lapse interval that TMR4/5 can track before reaching the 32-bit counter ceiling is:
            = Pscaled * PR4 = (3.2 us/count) * (2^32) = 13743.90 seconds
 *=============================================================================================================*/
void init_Timer4(void)
{
    T4CONbits.ON = 0;           // Stop any 16/32-bit Timer4 operation
    T5CONbits.ON = 0;           // Stop any 16-bit Timer5 operation
    T4CONSET = (0 << 1);        // Timer4 Clock Source set to be the Internal PBCLK Source
    T4CONbits.TCKPS = 0b111;    // Timer4 input clock prescaler value set to 1:256
    T4CONbits.T32 = 1;          // Timer4 and Timer5 form a 32-bit timer

    TMR4 = 0;                   // Clear the contents of TMR4 and TMR5
    PR4 = 0xFFFFFFFF;           // Load PR4 and PR5 registers with a 32-bit value

    T4CONbits.ON = 1;           // Enable Timer4/5
} // end init_Timer4

// Checks if Button 1 has been pressed
/* NOTE: RB10 is always HIGH, Button 1 is either damaged or soldered incorrectly
char readButton1(void)
{
    if (_RB10 == 1) {return(1);}
    if (_RB10 == 0) {return(0);}
} // end readButton1
*/

// Checks if Button 2 has been pressed
char readButton2(void)
{
    if (_RB11 == 1) {return(1);}
    if (_RB11 == 0) {return(0);}
} // end readButton2
```

```
void improperConfigCheck(void)
{
    // Ensure that the counts/rev on both LW and RW Pucks are configured to the same values!
    if(ONE_REV_LW != ONE_REV_RW)
    { // ONE_REV_LW and ONE_REV_RW are defined in support.h
        printString("ERROR: LW/RW Puck CTS do not match!");
        delay_ms(10000);
    }
} // end improperConfigCheck
```

F-11: UART.c

```
/*****************************************************************************************************************
Name: UART.c
Creation Date: 09/12/2013
Author: Airton R. da Silva Jr.
Comments: Functions used to configure and initialize:
    UART3 for transmission/reception with Serial Graphic LCD and MATLAB.

    Also contains driver functions used to send and read UART messages.
*****************************************************************************************************************/

#include <p32xxxx.h>
#include <plib.h>
#include "support.h"

/*================================================================================================================
    init_UART3() configures and initializes UART3 module.

    Step 1: Initialize the UxBRG register for the appropriate baud rate.
    Step 2: Set the number of data and Stop bits, and parity selection by writing to the PDSEL<1:0> bits
            (UxMODE<2:1>) and STSEL bit (UxMODE<0>).
    Step 3: If interrupts are desired, set the UxTXIE or UxRXIE control bits in the corresponding
            Interrupt Enable Control register (IEC). Specify the interrupt priority and subpriority for
            the transmit interrupt using the UxIP<2:0> and UxIS<1:0> control bits in the corresponding
            Interrupt Priority Control register (IPC). Also, select the Transmit Interrupt mode by writing
            to the UTXISEL bits (UxSTA<15:14>) or the Receive Interrupt mode by writing to the
            URXISEL<1:0> bits (UxSTA<7:6>).
    Step 4: Enable the transmission by setting the UTXEN bit (UxSTA<10>), which also sets the
            UxTXIF bit. The UxTXIF bit should be cleared in the software routine that services the
            UART transmit interrupt. The operation of the UxTXIF bit is controlled by the UTXISEL
            control bits. Enable the UART receiver by setting the URXEN bit (UxSTA<12>).
    Step 5: Enable the UART module by setting the ON bit (UxMODE<15>).
    Step 6: Load data to the UxTXREG register (starts transmission) or read data from the receive buffer.
            If 9-bit transmission is selected, read a word; otherwise,read a byte. The URXDA bit is set
            whenever data is available in the buffer.

    Serial Graphic LCD UART Communication Requirements:
        Baud Rate          = 115,200 bps (adjustable)
        Number of Data Bits = 8 data bits
        Number of Stop Bits = 1 stop bit
        Parity Selection    = No parity

    UART3 Baud Rate Generator (U3BRG) Calculation with Standard Speed Mode (BRGH = 0):
        Peripheral Bus Clock,           Fpb = 80 MHz
        Desired UART3 Baud Rate,        Baud Rate = 115,200 bps

        UxBRG = [Fpb/(16*(Baud Rate))] - 1 = [(80 MHz)/(16*115,200 bps)] - 1
              = 42.403 = 42

        Calculated Baud Rate:
        Baud Rate = Fpb/[16*(UxBRG + 1)] = (80,000,000 Hz)/[16*(42+1)]
                  = 116,279 bps

        Baud Rate Error:
        Error = (Calculated Baud Rate - Desired Baud Rate)
              = (116,279 - 115,200)/115,200
              = 0.94%

    References:
        Section 21. UART - S21.5: UART Transmitter [p. 17]
        Section 21. UART - S21.7: UART Receiver [p. 21]
 *================================================================================================================*/
void init_UART3 (void)
{
    U3BRG = 42;             // Step 1: Set Baud rate to approximately 115,200 bps
    U3MODEbits.STSEL = 0;   // Step 2: 1 Stop bit
    U3MODEbits.PDSEL = 0;   // Step 2: 8-bit data, no parity
    U3MODEbits.BRGH = 0;    // Step 2: Standard Speed mode - 16x baud clock enabled
    U3MODEbits.UEN = 0;     // Step 2: U3TX and U3RX pins are enabled and used
    U3STA = 0;              // Step 3: Clear all bits in UART3 Status and Control Register
    U3STAbits.URXEN = 1;    // Step 4: UART3 receiver enabled. U3RX pin controlled by UART3
    U3STAbits.UTXEN = 1;    // Step 4: UART3 transmitter enabled. U3TX pin controlled by UART3

    U3MODEbits.ON = 1;      // Step 5: Enable UART3 peripheral
} // end init_UART3

void sendUARTbyte(char UARTtx_byte)
{
    while(U3STAbits.UTXBF == 1);    // Wait until message is transmitted and buffer is empty
    U3TXREG = UARTtx_byte;          // Write data to TX buffer
} // end sendUARTmsg

short readUARTmsg(void)
{
    short rx_data;
```

```
    if(U3STAbits.URXDA == 1)
    {
        rx_data = U3RXREG;              // Read RX buffer
        while(U3STAbits.URXDA == 1);    // Wait until receive buffer is empty
    }
    else
    {
        printf("Receive buffer empty.");
    }
    return(rx_data);
} // end readUARTmsg
```

F-12: support.h

```
/*****************************************************************************************************************
Name: support.h
Creation Date: 09/12/2013
Authors: Airton R. da Silva Jr.
Comments: Support file for variable/structure declarations, function prototypes, and puck components.
          This file also supports CAN1 and SPI3 configuration/driver files.
*****************************************************************************************************************/

#ifndef SUPPORT_H
#define SUPPORT_H

#include <plib.h>
#include <math.h>

/*===============================================================================================================*
                                        GLOBAL VARIABLE DECLARATIONS
 *===============================================================================================================*/
extern unsigned char CANmsgReceived;
extern unsigned char resetEncoderCount;
extern unsigned char IMUinitialized;
extern unsigned char PuckLWinitialized;
extern unsigned char PuckRWinitialized;
extern unsigned int i;
extern unsigned int j;
extern unsigned int run_time;
extern unsigned int sysCLK;
extern unsigned int wait_flag;
extern long jointPosition[2];
extern unsigned int CANmsgBuffersFIFO[16];
extern CANRxMessageBuffer rxCANmsg;

/*===============================================================================================================*
                                        MACRO AND VARIABLE DEFINITIONS
 *===============================================================================================================*/
/*=============================================*
    CAN Baud Rate Prescalar Configuration
 *=============================================*/
// References:
//   AN1249 - EXAMPLE 2: CODE FOR BRP CONFIGURATION [pp. 7]
//   Section 34 - CAN - S34.10: Bit Timing [pp. 83]
#define FCAN    80000000    // Equivalent to PIC Internal Clock Frequency, Fosc
#define BITRATE 1000000
#define NTQ     8           // 8 Time Quanta in a Bit Time
#define BRP_VAL ((FCAN/(2*NTQ*BITRATE))-1)

/*=============================================*
    Extra Variable Definitions
 *=============================================*/
#define FALSE       0
#define TRUE        1
#define PI          3.141592653589
#define SAMP_RATE   0.01                    // Firmware sampling rate

/*=============================================*
    LED Logic Definitions
 *=============================================*/
#define LED1_ON     LATEbits.LATE5 = 1
#define LED1_OFF    LATEbits.LATE5 = 0
#define LED2_ON     LATEbits.LATE6 = 1
#define LED2_OFF    LATEbits.LATE6 = 0
#define LED3_ON     LATEbits.LATE7 = 1
#define LED3_OFF    LATEbits.LATE7 = 0
#define LED4_ON     LATGbits.LATG6 = 1
#define LED4_OFF    LATGbits.LATG6 = 0
#define LED5_ON     LATGbits.LATG9 = 1
#define LED5_OFF    LATGbits.LATG9 = 0
#define LED6_ON     LATBbits.LATB8 = 1
#define LED6_OFF    LATBbits.LATB8 = 0
#define LED7_ON     LATBbits.LATB9 = 1
#define LED7_OFF    LATBbits.LATB9 = 0
#define LED8_ON     LATCbits.LATC12 = 1
#define LED8_OFF    LATCbits.LATC12 = 0
#define LED9_ON     LATDbits.LATD8 = 1
#define LED9_OFF    LATDbits.LATD8 = 0

/*=============================================*
    Puck and Peripherals IDs
 *=============================================*/
#define CPU         0       // Central processing unit ID
#define L_WHEEL_ID  1       // Left wheel ID
#define R_WHEEL_ID  2       // Right wheel ID
#define IMU_ID      3       // ADIS16300: inertial measurement unit ID
#define ADC1_ID     4       // MCP3903: A/D converter #1 ID
```

```
#define ADC2_ID     5       // MCP3903: A/D converter #2 ID

/*============================================*
    IMU (ADIS16300) Output Data Registers
 *============================================*/
// Table 8: User Register Memory Map [p. 10]
#define FLASH_CNT    0x00    // Flash memory write count
#define SUPPLY_OUT   0x02    // Power supply measurement
#define GYRO_OUT     0x04    // X-axis gyroscope output
#define XACCL_OUT    0x0A    // X-axis accelerometer output
#define YACCL_OUT    0x0C    // Y-axis accelerometer output
#define ZACCL_OUT    0x0E    // Z-axis accelerometer output
#define TEMP_OUT     0x10    // X-axis gyroscope temperature measurement
#define PITCH_OUT    0x12    // X-axis inclinometer output measurement
#define ROLL_OUT     0x14    // Y-axis inclinometer output measurement
#define AUX_ADC      0x16    // Auxiliary ADC output
#define GYRO_OFF     0x1A    // X-axis gyroscope bias offset factor
#define XACCL_OFF    0x20    // X-axis acceleration bias offset factor
#define YACCL_OFF    0x22    // Y-axis acceleration bias offset factor
#define ZACCL_OFF    0x24    // Z-axis acceleration bias offset factor
#define ALM_MAG1     0x26    // Alarm 1 amplitude threshold
#define ALM_MAG2     0x28    // Alarm 2 amplitude threshold
#define ALM_SMPL1    0x2A    // Alarm 1 sample size
#define ALM_SMPL2    0x2C    // Alarm 2 sample size
#define ALM_CTRL     0x2E    // Alarm control
#define AUX_DAC      0x30    // Auxiliary DAC data
#define GPIO_CTRL    0x32    // Auxiliary digital input/output control
#define MSC_CTRL     0x34    // Miscellaneous control
#define SMPL_PRD     0x36    // Internal sample period (rate) control
#define SENS_AVG     0x38    // Dynamic range/digital filter control
#define SLP_CNT      0x3A    // Sleep mode control
#define DIAG_STAT    0x3C    // System status
#define GLOB_CMD     0x3E    // System command

// Table 12: GLOB_CMD [p. 12]
#define GLOB_CMD_SW_RESET       (1<<7)  // Software reset command
#define GLOB_CMD_P_AUTO_NULL    (1<<4)  // Precision autonull command
#define GLOB_CMD_FLASH_UPD      (1<<3)  // Flash update command
#define GLOB_CMD_DAC_LATCH      (1<<2)  // Auxiliary DAC data latch
#define GLOB_CMD_FAC_CALIB      (1<<1)  // Factory calibration restore command
#define GLOB_CMD_AUTO_NULL      (1<<0)  // Autonull command; Automatic calibration of sensors offset

// Table 13: SAMPL_PRD [p. 12]
#define SAMPL_PRD_TIME_BASE     (1<<7)  // Time base (t_B); 0 = 0.61035 ms, 1 = 18.921 ms
#define SAMPL_PRD_DEFAULT       0x01    // Sets IMU sampling rate to default, 819.2 SPS
#define SAMPL_PRD_LOWPWR        0x0A    // Sets IMU sampling rate to low power, 149 SPS

// Table 14: SLP_CNT [p. 12]
#define SLP_CNT_POWER_OFF       (1<<8)  // Indefinite sleep mode, set to 1

// Table 15: SENS_AVG [p. 13]
#define SENS_AVG_DEFAULT        (1<<10) // Measurement range (sensitivity): +/- 300 deg/sec (default)
#define SENS_AVG_MEDIUM         (1<<9)  // Measurement range (sensitivity): +/- 150 deg/sec
#define SENS_AVG_LOW            (1<<8)  // Measurement range (sensitivity): +/- 75 deg/sec
#define SENS_AVG_FILTER_129     0x07    // Bartlett filter tap setting: N = 129 taps
#define SENS_AVG_FILTER_65      0x06    // Bartlett filter tap setting: N = 65 taps
#define SENS_AVG_FILTER_33      0x05    // Bartlett filter tap setting: N = 33 taps
#define SENS_AVG_FILTER_17      0x04    // Bartlett filter tap setting: N = 17 taps
#define SENS_AVG_FILTER_9       0x03    // Bartlett filter tap setting: N = 9 taps
#define SENS_AVG_FILTER_5       0x02    // Bartlett filter tap setting: N = 5 taps
#define SENS_AVG_FILTER_3       0x01    // Bartlett filter tap setting: N = 3 taps
#define SENS_AVG_FILTER_1       0x00    // Bartlett filter tap setting: N = 1 tap

// Table 17: MSC_CTRL [p. 13]
#define MSC_CTRL_MEMORYTEST     (1<<11) // Memory test; 1 = enabled, 0 = disabled
#define MSC_CTRL_INTSELFTEST    (1<<10) // Internal self-test; 1 = enabled, 0 = disabled
#define MSC_CTRL_NEGSELFTEST    (1<<9)  // Manual self-test, - stimulus; 1 = enabled, 0 = disabled
#define MSC_CTRL_POSSELFTEST    (1<<8)  // Manual self-test, + stimulus; 1 = enabled, 0 = disabled
#define MSC_CTRL_CORRECTGYRO    (1<<7)  // Gyroscope bias compensation; 1 = enabled, 0 = disabled
#define MSC_CTRL_ALIGNACCL      (1<<6)  // Accelerometer origin alignment; 1 = enabled, 0 = disabled
#define MSC_CTRL_DATARDYEN      (1<<2)  // Data ready enable; 1 = enabled, 0 = disabled
#define MSC_CTRL_DATARDYPOL     (1<<1)  // Data ready polarity; 1 = active high, 0 = active low
#define MSC_CTRL_DATARDYSEL     (1<<0)  // Data ready line select; 1 = DIO2, 0 = DIO1

// Table 21: DIAG_STAT [p. 14]
#define DIAG_STAT_ZACCL_FAIL    (1<<15) // Z-axis accelerometer self-test failure; 1 = error, 0 = normal
#define DIAG_STAT_YACCL_FAIL    (1<<14) // Y-axis accelerometer self-test failure; 1 = error, 0 = normal
#define DIAG_STAT_XACCL_FAIL    (1<<13) // X-axis accelerometer self-test failure; 1 = error, 0 = normal
#define DIAG_STAT_XGYRO_FAIL    (1<<10) // Gyroscope self-test failure; 1 = error, 0 = normal
#define DIAG_STAT_ALARM2        (1<<9)  // Alarm 2 status; 1 = active, 0 = inactive
#define DIAG_STAT_ALARM1        (1<<8)  // Alarm 1 status; 1 = active, 0 = inactive
#define DIAG_STAT_FLASH_CHK     (1<<6)  // Flash test, check-sum flag; 1 = failure, 0 = normal
#define DIAG_STAT_SELF_TEST     (1<<5)  // Self-test diagnostic error flag; 1 = error, 0 = normal
#define DIAG_STAT_OVERFLOW      (1<<4)  // Sensor overrange; 1 = error, 0 = normal
#define DIAG_STAT_SPI_FAIL      (1<<3)  // SPI communications failure; 1 = error, 0 = normal
#define DIAG_STAT_FLASH_UPT     (1<<2)  // Flash update failed; 1 = error, 0 = normal
#define DIAG_STAT_POWER_HIGH    (1<<1)  // Power supply above 5.25 V; 1 = true, 0 = false
#define DIAG_STAT_POWER_LOW     (1<<0)  // Power supply below 4.75 V; 1 = true, 0 = false
```

```
/*============================================*
    Puck Macros and Variable Definitions
 *============================================*/
// Border(): This macro is used to bound the torque command to a value between Min and Max
// MAX_TORQUE: Value between 0 and 8000, anything over 4200 causes rattling in motor when commanded
#define Border(Value,Min,Max)  (Value<Min)?Min:((Value>Max)?Max:Value)
#define MAX_TORQUE      4200        // Max allowed motor torque in Puck units
#define MAX_VELOCITY    1500        // Factory default MV = 1500 cts/ms
#define ONE_REV_LW      PUCKconvFactors[0].counts_per_rev   // Factory default CTS = 4096
#define ONE_REV_RW      PUCKconvFactors[1].counts_per_rev   // Factory default CTS = 4096

/*============================================*
    Puck Control Mode States
 *============================================*/
enum
{ // Property #5: STAT
   STATUS_RESET = 0,
   STATUS_READY = 2
};

enum
{ // Property #8: MODE
   MODE_IDLE = 0,
   MODE_DUTY = 1,
   MODE_TORQUE = 2,
   MODE_PID = 3,
   MODE_VELOCITY = 4,
   MODE_TRAPEZOIDAL = 5
};

/*============================================*
    Puck Properties List
 *============================================*/
#define VERS    0       // Firmware version
#define ROLE    1
#define SN      2
#define ID      3
#define ERROR   4
#define STAT    5       // Status: 0 = Reset/Monitor, 2 = Ready/Main
#define ADDR    6
#define VALUE   7
#define MODE    8
#define TEMP    9
#define PTEMP   10
#define OTEMP   11
#define BAUD    12
#define LOCK    13
#define DIG0    14
#define DIG1    15
#define FET0    16
#define FET1    17
#define ANA0    18
#define ANA1    19
#define THERM   20
#define VBUS    21
#define IMOTOR  22
#define VLOGIC  23
#define ILOGIC  24
#define SG      25
#define GRPA    26
#define GRPB    27
#define GRPC    28
#define CMD     29
#define SAVE    30
#define LOAD    31
#define DEF     32
#define FIND    33
#define X0      34
#define X1      35
#define X2      36
#define X3      37
#define X4      38
#define X5      39
#define X6      40
#define X7      41
#define T       42      // Motor torque command (in milliamps)
#define MT      43      // Maximum torque a Puck will apply (in milliamps); Default = 4700 mA
#define V       44      // Motor velocity (in counts/millisecond)
#define MV      45      // Maximum velocity (in counts/millisecond); Default = 1500 cts/ms
#define MCV     46
#define MOV     47
#define P       48      // Motor position (in encoder counts)
#define P2      49
#define DP      50
#define DP2     51
#define E       52
#define E2      53
```

```
#define OT      54
#define OT2     55
#define CT      56
#define CT2     57
#define M       58
#define M2      59
#define DS      60
#define MOFST   61
#define IOFST   62
#define UPSECS  63
#define OD      64
#define MDS     65
#define MECH    66
#define MECH2   67
#define CTS     68
#define CTS2    69
#define PIDX    70
#define HSG     71
#define LSG     72
#define IVEL    73
#define IOFF    74
#define IOFF2   75
#define MPE     76
#define EN      77
#define TSTOP   78
#define KP      79
#define KD      80
#define KI      81
#define ACCEL   82
#define TENST   83
#define TENSO   84
#define JIDX    85
#define IPNM    86      // The puck torque unit to Nm conversion is stored here; Default = 2755 mA/Nm
#define HALLS   87
#define HALLH   88
#define HALLH2  89
#define POLES   90
#define IKP     91
#define IKI     92
#define IKCOR   93
#define HOLD    94
#define TIE     95
#define ECMAX   96
#define ECMIN   97
#define LFLAGS  98
#define LCTC    99
#define LCVC    100

/*=============================================*
    ADC (MCP3903) Internal Registers
 *=============================================*/
// TABLE 7-1: INTERNAL REGISTER SUMMARY [p. 37]
#define ADC_CH0         0x00        // Channel 0 ADC Data <23:0>, MSB first, left justified
#define ADC_CH1         0x01        // Channel 1 ADC Data <23:0>, MSB first, left justified
#define ADC_CH2         0x02        // Channel 2 ADC Data <23:0>, MSB first, left justified
#define ADC_CH3         0x03        // Channel 3 ADC Data <23:0>, MSB first, left justified
#define ADC_CH4         0x04        // Channel 4 ADC Data <23:0>, MSB first, left justified
#define ADC_CH5         0x05        // Channel 5 ADC Data <23:0>, MSB first, left justified
#define ADC_MOD         0x06        // Delta Sigma Modulators Output Value
#define ADC_PHASE       0x07        // Phase Delay Configuration Register
#define ADC_GAIN        0x08        // Gain Configuration Register
#define ADC_COM         0x09        // Status/Communication Register
#define ADC_CONFIG      0x0A        // Configuration Register

// REGISTER 7-3: PHASE REGISTER [p. 40]
#define ADC_PHASEC(val)     (val<<16)   // CH4 relative to CH5 phase delay (two's complement)
#define ADC_PHASEB(val)     (val<<8)    // CH2 relative to CH3 phase delay (two's complement)
#define ADC_PHASEA(val)     (val<<0)    // CH0 relative to CH1 phase delay (two's complement)

// REGISTER 7-4: PROGRAMMABLE GAIN AMPLIFIER (PGA) GAIN REGISTER [p. 41]
#define PGA_CHN(gain,chn)   (gain << (chn*4 + chn%2))               // Channel PGA gain
#define BOOST_CH(boost,chn) (boost << ((chn*4) + 3*((chn+1)%2)))    // Current scaling
#define PGA_GAIN_32         0b101       // PGA gain is 32
#define PGA_GAIN_16         0b100       // PGA gain is 16
#define PGA_GAIN_8          0b011       // PGA gain is 8
#define PGA_GAIN_4          0b010       // PGA gain is 4
#define PGA_GAIN_2          0b001       // PGA gain is 2
#define PGA_GAIN_1          0b000       // PGA gain is 1
#define BOOST_2X            0b1         // Channel has current x 2
#define BOOST_NORM          0b0         // Channel has normal current

// REGISTER 7-5: STATUS/COMMUNICATION REGISTER [p. 42]
#define COM_READ(val)           (val<<22)   // Address loop setting
#define COM_WMODE(val)          (val<<21)   // Write mode bit (internal use only)
#define COM_WIDTH_CHn(val)      (val<<15)   // ADC channels output data word width control
#define COM_DR_LTY(val)         (val<<14)   // Data Ready Latency Control for DRA, DRB, DRC pins
#define COM_DR_HIZ(val)         (val<<13)   // Data Ready Pin Inactive State Control for DRA, DRB, DRC pins
#define COM_DR_LINK(val)        (val<<12)   // Data Ready Link Control
```

```
#define COM_DRC_MODE(val)       (val<<10)   // Data Ready C Mode
#define COM_DRB_MODE(val)       (val<<8)    // Data Ready B Mode
#define COM_DRA_MODE(val)       (val<<6)    // Data Ready A Mode
#define COM_DRSTATUS_CHn(val)   (val<<0)    // Data Ready Status (READ ONLY)
#define WIDTH_24bit             0b111111    // Set CH0-CH5 output data word width to 24-bit mode
#define WIDTH_16bit             0b000000    // Set CH0-CH5 output data word width to 16-bit mode (DEFAULT)

// REGISTER 7-6: CONFIGURATION REGISTER [p. 44]
#define CONFIG_RESET_CHn(val)   (val<<18)   // Reset mode setting for ADCs
#define CONFIG_SHUTDOWN_CHn(val)    (val<<12)   // Shutdown mode setting for ADCs
#define CONFIG_DITHER_CHn(val)  (val<<6)    // Control for dithering circuit for idle tones cancellation
#define CONFIG_OSR(val)         (val<<4)    // Oversampling Ratio for Delta Sigma A/D Conversion
#define CONFIG_PRESCALE(val)    (val<<2)    // Internal Master Clock Prescaler Value
#define CONFIG_EXTVREF(val)     (val<<1)    // Internal Voltage Reference Shutdown Control
#define CONFIG_EXTCLK(val)      (val<<0)    // Clock Mode
#define OSR_256                 0b11        // Set Delta Sigma ADC oversampling ratio to 256
#define OSR_128                 0b10        // Set Delta Sigma ADC oversampling ratio to 128
#define OSR_64                  0b01        // Set Delta Sigma ADC oversampling ratio to 64 (DEFAULT)
#define OSR_32                  0b00        // Set Delta Sigma ADC oversampling ratio to 32

/*=============================================*
    LCD Variable Definitions
 *=============================================*/
#define LCD_LINE_LENGTH     21      // Max # of characters that can be printed per line on LCD display

/*=============================================*
    LQR Variable Definitions
 *=============================================*/
#define WHEEL_RADIUS    0.114       // Radius of left and right wheels [m]
#define DIST_LWtoRW     0.346       // Distance between the L & R wheels along the y-axis
#define G11             0.5         // Decoupling controller matrix element G(1,1)
#define G12             0.5         // Decoupling controller matrix element G(1,2)
#define G21             0.5         // Decoupling controller matrix element G(2,1)
#define G22             -0.5        // Decoupling controller matrix element G(2,2)
#define Kp11            -100.0000   // Decoupled LQR pitch gain matrix element Kp(1,1)
#define Kp12            -60.7486    // Decoupled LQR pitch gain matrix element Kp(1,2)
#define Kp13            370.5861    // Decoupled LQR pitch gain matrix element Kp(1,3)
#define Kp14            80.6936     // Decoupled LQR pitch gain matrix element Kp(1,4)
#define Ky11            31.6228     // Decoupled LQR yaw gain matrix element Ky(1,1)
#define Ky12            3.6995      // Decoupled LQR yaw gain matrix element Ky(1,2)

/*=============================================================================================================*
                                            MESSAGE STRUCTURES
 *=============================================================================================================*/
/*=============================================*
    Puck Message Structures
 *=============================================*/
// This structure stores received Puck messages that have been processed and converted to SI units
typedef struct PUCKstructSI
{ // LW = Left Wheel (ID 1), RW = Right Wheel (ID 2)
    float theta_XXrad;          // Wheel angular displacement [radians]; 'XX' denotes LW or RW
    float velocity;
    float torque;
    float temperature;
    float x;                    // Linear displacement along the x-axis
}PUCKmsgID;
extern PUCKmsgID PUCKdataSI[];

/* Note regarding milliAmp_per_Nm factor:
    IPNM is never actually used in the pucks themselves. It is simply a non-volatile place to store the
    conversion constant so that the controlling PC can read it upon startup and use it to convert between
    puck units and SI units of torque for the end user.

    Calculating IPNM:
    1 Amp = 1024 puck torque units
    Puck torques are saturated at 8191 by the communication layer

    Motor torque constant:  Kt = 0.457 (value published in motor specs sheet)
    Motor torque constant:  Kt = 0.379 (actual value)

    IPNM = [1024 PuckUnits/A] / [0.379 Nm/A] = 2701.85 ~= 2700 PuckUnits/Nm
*/
// This structure stores received Puck conversion factors set by the user (or factory default)
typedef struct PUCKconvFactorsStruct
{ // LW = Left Wheel (ID 1), RW = Right Wheel (ID 2)
    long counts_per_rev;        // Set to CTS = 4096 (counts per revolution)
    long milliAmp_per_Nm;       // Set to IPNM = 2700 (milli-Amps per Newton-meter)
}PUCKconvFactorsID;
extern PUCKconvFactorsID PUCKconvFactors[];

/*=============================================*
    IMU Message Structures
 *=============================================*/
// This structure stores received IMU messages that have been processed and converted to SI units
typedef struct IMUstructSI
{
    float IMUsupply;
    float IMUgyro;
```

```
    float IMUxaccl;
    float IMUyaccl;
    float IMUzaccl;
    float IMUtemp;
    float IMUpitch;
    float IMUroll;
}IMUmsgID;
extern IMUmsgID IMUdataSI;

/*=========================================================================================================*
                                        FUNCTION PROTOTYPES
 *=========================================================================================================*/
// support.c function prototypes
void config_clock(void);
void config_pins(void);
void delay_ms(unsigned int t);
void delay_us(unsigned int t);
void delay_ns(unsigned int t);
void improperConfigCheck(void);
void init_samptime(void);
void init_Timer4(void);
char readButton1(void);
char readButton2(void);

// CAN function prototypes
void init_CAN1(void);
void readCANmsg(void);
int sendCANmsg(int channel, int tx_id, int dataLength, unsigned char *tx_data);

// Puck function prototypes
void config_pucks(char puckMODE);
void checkTemp(int puckID);
void getConversionFactorsPuck(int puckID);
void resetPucks(void);
void setDefaultPropertiesPuck(int puckID);
void stopWheels(void);
int convertUnits_SI_to_Puck(int puckID, int property, float value);
int getPropertyPuck(int channel, int tx_id, int property, long *reply);
int MSGID(int GRP, int FROM, int TO);
int parseCANmsgPuck(int *rx_id, int *rx_property, long *rx_value);
int setPropertyPuck(int channel, int tx_id, int property, long dataValue);
int setPropertySlowPuck(int channel, int tx_id, int property, long dataValue);
int setTorquePuck(int channel, int puckID, float value);
int wakePuck(int channel, int who);
long encoderZeroCrossing(int puckID, long position);
float convertUnits_Puck_to_SI(int puckID, int property, long value);
float getPositionPuck(int channel, int puckID, char n);

// SPI function prototypes
void init_SPI3master(void);
void init_SPI4master(void);
short readSPImsg(char device);
short sendSPImsg(char device, short SPItx_data);

// IMU function prototypes
void config_IMU(void);
void executeAutomaticSelfTest(void);
void executeFlashMemoryTest(void);
void gyroPrecisionCalibration(void);
void resetIMU(void);
void restoreFactoryCalibration(void);
short convert_12bit_to_16bit_signed(short *value);
short convert_13bit_to_16bit_signed(short *value);
short convert_14bit_to_16bit_signed(short *value);
short getIMUdiagnostics(void);
short getPropertyIMU(short property, short *reply);
short parseSPImsgIMU(short property, short *rx_value);
short setPropertyIMU(short property, short value);
float convertUnits_IMU_to_SI(short property, short value);
float get_IMUgyro(IMUmsgID *IMUdataSI, char n);
float get_IMUpitch(IMUmsgID *IMUdataSI);
float get_IMUroll(IMUmsgID *IMUdataSI);
float get_IMUsupply(IMUmsgID *IMUdataSI);
float get_IMUtemp(IMUmsgID *IMUdataSI);
float get_IMUxaccl(IMUmsgID *IMUdataSI);
float get_IMUyaccl(IMUmsgID *IMUdataSI);
float get_IMUzaccl(IMUmsgID *IMUdataSI);

// UART function prototypes
void init_UART3 (void);

// LCD function prototypes
void clearScreen(void);
void config_LCD(void);
void drawBox(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2);
void drawCircle(UINT8 x, UINT8 y, UINT8 radius);
void drawLine(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2);
void eraseBlock(UINT8 x1, UINT8 y1, UINT8 x2, UINT8 y2);
```

```
void printLine(char *string, char lineLength);
void printVariable(char *stringVar, float number, char precision, char *stringUnits);
void resetXYref(void);
void restoreDefaultBaud(void);
void runDemo(void);
void setBacklightIntensity(UINT8 duty);
void setBaudRate(UINT8 baud);
void setPixel(UINT8 x, UINT8 y);
void setXref(UINT8 x);
void setYref(UINT8 y);
void terminateLine(char strLength, char lineLength);
void toggleReverseMode(void);
void toggleSplash(void);
char countDigits(int number);
char printFloat(float number, char precision);
char printInteger(int number);
char printString(char *string);

// LQR function prototypes
void logLQRdata(float theta_LWrad, float theta_RWrad, float x, float dx, float phi, float dphi,
                float psi, float dpsi, float T_LW, float T_RW);
void LQR(void);
void printLQRdataLCD(float theta_LWrad, float theta_RWrad, float x, float dx, float phi, float dphi,
                     float psi, float dpsi, float T_LW, float T_RW);
float getStateVariable_x(float theta_LWrad, float theta_RWrad);
float getStateVariable_dx(float x);
float getStateVariable_phi(float Gyaw, float Ax, float Ay);
float getStateVariable_psi(float theta_LWrad, float theta_RWrad);
float getStateVariable_dpsi(float psi);
float RMAfilter(float var, char n, char *k_ptr, float *varArr);
float DisplacementControl(float MinT_x, float MaxT_x, float MinV_dx, float MaxV_dx, float Td_phi);
float DisplacementControl_v2(float MinD_x, float MaxD_x, float MinV_dx, float MaxV_dx, float x);
float PitchControl(float MinT_phi, float MaxT_phi, float MinA_phi, float MaxA_phi, float Td_phi);
float HeadingAngleControl(float MinT_psi, float MaxT_psi, float MinV_dpsi, float MaxV_dpsi, float Td_psi);
float MinJerkTraj(float xi, float xf, float yi, float yf, float t);
float PIcontroller(float Kp, float Ki, float Ts, float error, float error_old, float output_old);

// ADC function prototypes
void config_ADC(void);
void getPropertyADC(char device, char dev_addr, char property, int *reply);
void init_MasterClockADC(int PWMfrequency, char dutyCycle);
void parseSPImsgADC(char property, int *rx_value);
void setPropertyADC(char device, char dev_addr, char property, int value);
int convert_17bit_to_32bit_signed(int *value);
int convert_20bit_to_32bit_signed(int *value);
int convert_23bit_to_32bit_signed(int *value);
int convert_24bit_to_32bit_signed(int *value);
int getLoadCellOffset_pitch(void);
int getLoadCellOffset_yaw(void);
float getLoadCellData_pitch(void);
float getLoadCellData_yaw(void);

#endif
```

APPENDIX G

DATA LOGGING, MATLAB FILE

```
% Name: PICtoPC_datalogger.m
% Creation Date: 02/19/2014
% Author: Airton R. da Silva Jr.
% Comments: Configuring and initializing MATLAB's serial COM port for transmission/reception
%     with a PIC32. Received data is processed, stored, and plotted in real-time.
%
%        'commaCount' variable counts the number of commas present in data set received from
%     PIC32 but it's primary purpose is to ensure only complete data sets are used in
%     subsequent lines of code. If MATLAB's serial sampling rate is slower than the PIC's
%     data set transmission rate then MATLAB's input buffer will eventually overflow, causing
%     data sets to be separated or resulting in data loss. 'commaCount' is used to prevent
%     MATLAB errors encountered due to incomplete data sets.
%
%        To plot output data in real time, set the PIC32's sampling rate to a low value (~20Hz)
%     in order to give MATLAB enough time to retrieve and process data sets while avoiding
%     buffer overflow.

clear all; close all; clc;

%% User defined variables
stateVar = 2;       % Specify the state variable that will be observed by MATLAB in real-time
      % 1 = ID   | 2 = thLW | 3 = thRW | 4 = x     | 5 = dx    | 6 = phi
      % 7 = dphi | 8 = psi  | 9 = dpsi | 10 = T_LW | 11 = T_RW |
maxCount = 200;      % Specify the max number of points that can be displayed in plot
stopTime = 30;       % Set run time to terminate data logging [in seconds]
maxDataSet = 1500;    % Specify the max number of data sets that will be logged in text file
expectVarN = 10;     % Specify the number of unique variables expected from the PIC32

% Specify if data log will be processed and plotted in real-time or stored in a text file
logType = 'static';  % 'static' data acquisition allows higher sampling rates (<= 150 Hz)
% logType = 'dynamic'; % 'dynamic' data acquisition plots data in real-time (<= 20 Hz)

%% Preallocating function variables
time = zeros(1, maxCount);
data = zeros(1, maxCount);
count = 0;                                  % Counter used track loop iteration
dataArray = zeros(maxDataSet, expectVarN);   % Array used to store serial data

%% Plot parameters
plotTitle = 'Serial Data Log';   % Plot title
xLabel = 'Elapsed Time [s]';     % X-axis label
yLabel = 'Data';                 % Y-axis label
plotGrid = 'on';                 % Turn on grid
min = -1.0;                        % Set y-axis minimum
max =  1.0;                        % Set y-axis maximum
scrollWidth = 12;    % Range of data displayed in plot; plots entire data log if width <= 0
delay = 0.001;       % Specify delay to allow MATLAB plot to update

if (strcmp(logType,'dynamic'))
   plotGraph = plot(time,data,'-mo','LineWidth',1,'MarkerEdgeColor','k',...
      'MarkerFaceColor',[.49 1 .63],'MarkerSize',2);
   title(plotTitle,'FontSize',25);
   xlabel(xLabel,'FontSize',15);
   ylabel(yLabel,'FontSize',15);
   axis([0 10 min max]);
   grid(plotGrid);
end

%% Configuration parameters and initialization for serial COM port (complies with PIC32 config)
delete(instrfindall);                % Release all COM ports used by MATLAB

serPIC = serial('COM3');             % Create serial object
serPIC.BaudRate = 115200;            % Set Baud rate to 115,200
serPIC.Tag = 'PIC32';                % Assign a label to serial port object
serPIC.InputBufferSize = 10000;      % Size of input buffer in bytes
serPIC.FlowControl = 'none';         % Data flow control (handshaking) is not used
serPIC.Parity = 'none';              % Parity checking is not performed
serPIC.DataBits = 8;                 % Number of data bits transmitted
```

```
serPIC.StopBit = 1;                    % Number of bits used to indicate end of byte
serPIC.Timeout = 5;                    % Waiting time to complete read/write operation
serPIC.Terminator = '$';               % Indicator used to terminate a series of bytes
serPIC.ReadAsyncMode = 'continuous';   % Automatically read and store data inside input buffer
fopen(serPIC);       % Open the serial port read/write
tic;                 % Begin stopwatch timer
pause(0.8);          % Wait for data to arrive (adjust according to device sampling rate)
disp('PIC32 to PC communication initialized!')
disp('   (Close plot to terminate session.)')

% Check if device is transmitting and data is available inside input buffer
if (get(serPIC, 'BytesAvailable') == 0)
   error('ERROR: Buffer empty; ensure transmission has been initialized.')
end

% Wait until terminator character arrives to avoid data loss during fscanf()
while (fread(serPIC,1) ~= serPIC.Terminator)
end

%% Read data from PIC32 and store tab-delimited data inside a text document
% Maximum PIC32 sampling rate tested to avoid input buffer overflow is <= 66.6 Hz
if (strcmp(logType,'static'))
   for k = 1:maxDataSet
      % Use this function to check if MATLAB is able to read input buffer before it overflows
      fprintf('%d ', serPIC.BytesAvailable)

      % Check if MATLAB's serial input buffer is full and overflowing with data bytes
      if (get(serPIC, 'BytesAvailable') == serPIC.InputBufferSize)
         error('ERROR: Serial input buffer overflow! Reduce serial device sampling rate.')
      end

      % Read data from serial port and format it as a string
      dataStr = fscanf(serPIC, '%s');        % Read serial data up until '$' terminator

      % Remove terminator character '$' at end of string
      dataStr = strrep(dataStr,'$','');

      % Convert comma-delimited string to a row vector with a numerical value in each column
      dataArray(k,1) = toc;                          % Append elapsed time to each data set
      dataArray(k,2:expectVarN+1) = str2num(dataStr);   % Convert string to number set
   end

   % Save tab-delimited data inside text document
   dlmwrite('datalog4.txt', dataArray, 'delimiter', '\t','precision', '%.5f');
   type datalog4.txt
end

%% Read data from PIC32 and plot results in real-time
% Maximum PIC32 sampling rate tested to avoid input buffer overflow is <= 10 Hz
if (strcmp(logType,'dynamic'))
   % Count the number of commas delimiting the data set received from PIC
   commaCount = length(strfind(fscanf(serPIC, '%s'),','));

   % Terminate loop when graph is closed or spacified max run time is reached
   while (ishandle(plotGraph) && toc <= stopTime)
      % Read data from serial port and format it as a string
      dataASCII = fscanf(serPIC, '%s');      % Read serial data up until '$' terminator

      % Use this function to check if MATLAB is able to read input buffer before it overflows
      fprintf('%d ', serPIC.BytesAvailable)

      % Check if MATLAB's serial input buffer is full and overflowing with data bytes
      if (get(serPIC, 'BytesAvailable') == serPIC.InputBufferSize)
         disp('WARNING: Serial input buffer overflow! Reduce serial device sampling rate.')
      end

      % Check if stored data set is complete, if not, discard it and fetch new data set
      while (length(strfind(dataASCII,',')) ~= commaCount)
         dataASCII = fscanf(serPIC, '%s');      % Read serial data up until '$' terminator
         disp('WARNING: Incomplete data set discarded; a new data set has been retrieved.')
      end

      % Remove terminator character '$' at end of string
      dataASCII = strrep(dataASCII,'$','');

      % Convert comma-delimited string to a row vector with a numerical value in each column
```

```
      dataNum = str2num(dataASCII);
      count = count + 1;
      time(count) = toc;                  % Extract elapsed time
      data(count) = dataNum(stateVar);    % Extract desired state variable

      xData = time(time > time(count)-scrollWidth); % Only plot points within scrollWidth range
      yData = data(time > time(count)-scrollWidth);

      % Plot processed serial data in real-time
      if(scrollWidth > 0)
         set(plotGraph,'XData',xData,'YData',yData);
         axis([time(count)-scrollWidth time(count) min max]);  % Automatically adjust axes
      else
         set(plotGraph,'XData',xData,'YData',yData);
         axis([0 time(count) min max]);
      end

      % To limit resource consumption, maxi points displayed on plot is limited by 'maxCount'
      if (count == maxCount)
         count = 0;                % Reset count to zero; old plot points will be overwritten
      end

      % Allow MATLAB to update plot
      pause(delay);                % Minimum pause value averages to about 0.008 s
   end
end

%% Close serial port
fclose(serPIC);                % Close the serial port
delete(serPIC);                % Delete the serial object
```

APPENDIX H

BILL OF MATERIALS – MECHANICAL COMPONENTS

Table H-1. Bill of materials for the mechanical components and other mechanical accessories used in prototype construction.

Bill of Materials: Mechanical Components and Other Mechanical Accessories						
Reference	**QTY**	**Part Number**	**Description**	**Supplier**	**Unit Pricing**	**Total Cost**
WHEELED CHASSIS	1	N/A	Two-Wheeled Chassis Components	Grupen	Unknown	Unknown
HANDLEBAR	1	N/A	Handlebar Components (See Zucker's BOM)	Zucker	$334.370	$334.37
PEND BASE PLATE	1	8975k445	Multipurpose Aluminum Sheet (Alloy 6061)	McMaster	$17.23	$17.23
PCB BASE PLATE	1	89015k18	Multipurpose Aluminum Sheet (Alloy 6061)	McMaster	$28.34	$28.34
6-32 HEX BOLT	16	90480a007	Zinc-Plated Steel Machine Screw Hex Nut	McMaster	$1.16	$1.16
6-32 WASHER	16	90126a509	Zinc-Plated Steel Type A SAE Flat Washer	McMaster	$1.31	$1.31
1-72 CAPSCREW	2	92196a071	18-8 Stainless Steel Socket Head Cap Screw	McMaster	$5.88	$5.88
1-72 HEX BOLT	2	90480a002	Zinc-Plated Steel Machine Screw Hex Nut	McMaster	$2.96	$2.96
1-72 WASHER	2	95395a102	Brass Thick Flat Washer	McMaster	$3.95	$3.95
EXT RTNG RING	2	91590A250	Stainless Steel Ext. Retaining Ring	McMaster	$2.03	$4.06
CUSHION FOAM	1	87035k46	Antistatic Super-Cushioning Polyure. Foam	McMaster	$12.38	$12.38
CABLE TIE	1	7130K13	Std. Nylon Cable Tie 40 lbs TS; Pkg. Qty: 100	McMaster	$4.31	$4.31
GRIP WRAP	3	32765K211	Blue Thin Wrap-on Grip 1"W; Pkg. Qty: 3	McMaster	$8.18	$24.54
VELCRO	1	9273K15	Velcro General Purpose Hook and Look	McMaster	$11.18	$11.18
				Grand Total:		$451.67

BIBLIOGRAPHY

[1] A. Greenblatt, "Aging Population," *CQ Res.*, vol. 21, no. 25, pp. 577–600, 2011.

[2] A. Greenblatt, "Aging Baby Boomers," *CQ Res.*, vol. 17, no. 37, pp. 865–888, 2007.

[3] D. Rodriguez-Losada, F. Matia, A. Jimenez, R. Galan, and G. Lacey, "Implementing Map Based Navigation in Guido, the Robotic SmartWalker," *Int. Conf. Robot. Autom.*, pp. 3390–3395, 2005.

[4] G. Lee, T. Ohnuma, and N. Y. Chong, "Design and Control of JAIST Active Robotic Walker," *J. Intell. Serv. Robot.*, vol. 3, no. 3, pp. 125–135, Apr. 2010.

[5] C. Barrué, R. Annicchiarico, U. Cortés, A. Martínez-Velasco, E. X. Martín, F. Campana, and C. Caltagirone, "The i-Walker: An Intelligent Pedestrian Mobility Aid," *Comput. Intell. Healthc. 4*, pp. 103–123, 2010.

[6] H.-G. Jun, Y.-Y. Chang, B.-J. Dan, B.-R. Jo, B.-H. Min, H. Yang, W.-K. Song, and J. Kim, "Walking and sit-to-stand support system for elderly and disabled," *2011 IEEE Int. Conf. Rehabil. Robot.*, pp. 1–5, Jan. 2011.

[7] A. F. Neto, R. Ceres, E. Rocon, and J. L. Pons, "Empowering and Assisting Natural Human Mobility: The Simbiosis Walker," *Int. J. Adv. Robot. Syst.*, vol. 8, no. 3, pp. 34–50, 2011.

[8] Y. Hirata, S. Komatsuda, and K. Kosuge, "Fall Prevention Control of Passive Intelligent Walker Based on Human Model," *2008 IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pp. 1222–1228, Sep. 2008.

[9] D. Choi, M. Kim, and J.-H. Oh, "Development of a Rapid Mobile Robot with a Multi-Degree-of-Freedom Inverted Pendulum Using the Model-Based Zero-Moment Point Stabilization Method," *Adv. Robot.*, vol. 26, no. 5–6, pp. 515–535, Jan. 2012.

[10] F. Grasser, A. D'Arrigo, S. Colombi, and A. C. Rufer, "JOE: A Mobile, Inverted Pendulum," *IEEE Trans. Ind. Electron.*, vol. 49, no. 1, pp. 107–114, 2002.

[11] J. Li, X. Gao, Q. Huang, Q. Du, and X. Duan, "Mechanical Design and Dynamic Modeling of a Two-Wheeled Inverted Pendulum Mobile Robot," *2007 IEEE Int. Conf. Autom. Logist.*, pp. 1614–1619, Aug. 2007.

[12] S. Lin, C.-C. Tsai, and H.-C. Huang, "Nonlinear Adaptive Sliding-Mode Control Design for Two-Wheeled Human Transportation Vehicle," *Proc. 2009 IEEE Int. Conf. Syst. Man, Cybern.*, pp. 1965–1970, 2009.

[13] D. R. Jones and K. A. Stol, “Modelling and Stability Control of Two-Wheeled Robots in Low-Traction Environments,” *Australas. Conf. Robot. Autom.*, pp. 1–9, 2010.

[14] S. Kalra, D. Patel, and K. Stol, “Design and Hybrid Control of a Two Wheeled Robotic Platform,” *Australas. Conf. Robot. Autom.*, pp. 1–7, 2007.

[15] K. D. Do and G. Seet, “Motion Control of a Two-Wheeled Mobile Vehicle with an Inverted Pendulum,” *J. Intell. Robot. Syst.*, vol. 60, no. 3–4, pp. 577–605, May 2010.

[16] S. Ahmad and M. O. Tokhi, “Linear Quadratic Regulator (LQR) Approach for Lifting and Stabilizing of Two-Wheeled Wheelchair,” *2011 4th Int. Conf. Mechatronics*, no. May, pp. 1–6, May 2011.

[17] C. Huang, W. Wang, and C. Chiu, “Design and Implementation of Fuzzy Control on a Two-Wheel Inverted Pendulum,” *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 2988–3001, 2011.

[18] T.-J. Ren, T.-C. Chen, and C.-J. Chen, “Motion control for a two-wheeled vehicle using a self-tuning PID controller,” *Control Eng. Pract.*, vol. 16, no. 3, pp. 365–375, Mar. 2008.

[19] S. H. Jeong and T. Takahashi, “Wheeled Inverted Pendulum Type Assistant Robot: Inverted Mobile, Standing, and Sitting Motions,” *2007 IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pp. 1932–1937, Oct. 2007.

[20] C. Xu, M. Li, and F. Pan, “The System Design and LQR control of a Two-wheels Self-balancing Mobile Robot,” *2011 Int. Conf. Electr. Control Eng.*, pp. 2786–2789, Sep. 2011.

[21] K. M. Goher and M. O. Tokhi, “Development, Modeling and Control of a Novel Design of Two-Wheeled Machines,” *Cyber Journals Multidiscip. Journals Sci. Technol. J. Sel. Areas Robot. Control*, pp. 6–16, 2010.

[22] R. Kahani and B. Moaveni, “Control of Two-Wheels Inverted Pendulum Using Parallel Distributed Compensation and Fuzzy Linear Quadratic Regulator,” *2011 3rd Int. Conf. Comput. Model. Simul.*, pp. 312–317, 2011.

[23] P. K. W. Abeygunawardhana and T. Murakami, “Vibration Suppression of Two-Wheel Mobile Manipulator Using Resonance-Ratio-Control-Based Null-Space Control,” *IEEE Trans. Ind. Electron.*, vol. 57, no. 12, pp. 4137–4146, Dec. 2010.

[24] K. M. Goher and M. O. Tokhi, “A New Configuration of Two-Wheeled Inverted Pendulum: A Lagrangian-Based Mathematical Approach,” *J. Sel. Areas Robot. Control*, pp. 1–5, 2010.

[25] Z. Li, Y. Zhu, and T. Mo, “Adaptive Robust Dynamic Balance and Motion Control of Mobile Wheeled Inverted Pendulums,” *Proc. 7th World Congr. Intell. Control Autom.*, no. 1, pp. 933–938, 2008.

[26] Y. Kim, S. H. Kim, and Y. K. Kwak, “Dynamic Analysis of a Nonholonomic Two-Wheeled Inverted Pendulum Robot,” *J. Intell. Robot. Syst.*, vol. 44, no. 1, pp. 25–46, Jan. 2005.

[27] M. Muhammad, S. Buyamin, M. N. Ahmad, and S. W. Nawawi, “Dynamic Modeling and Analysis of a Two-Wheeled Inverted Pendulum Robot,” *2011 Third Int. Conf. Comput. Intell. Model. Simul.*, pp. 159–164, Sep. 2011.

[28] S. W. Nawawi, M. N. Ahmad, and J. H. S. Osman, “Development of a Two-Wheeled Inverted Pendulum Mobile Robot,” *5th Student Conf. Res. Dev.*, pp. 1–5, 2007.

[29] S. J. Lee, Y. G. Bae, and S. Jung, “Object Handling Control between a Balancing Robot and a Human Operator,” *2012 IEEE Int. Symp. Ind. Electron.*, pp. 931–936, May 2012.

[30] S. J. Lee and S. Jung, “Experimental Studies of an Object Handling Task by Force Control between Two Balancing Robots,” *11th Int. Conf. Control. Autom. Syst.*, pp. 197–201, 2011.

[31] X. Ruan and J. Chen, “H∞ Robust Control of Self-Balancing Two-Wheeled Robot,” *Proc. 8th World Congr. Intell. Control Autom.*, pp. 6524–6527, 2010.

[32] N. Hatakeyama and A. Shimada, “Movement Control using Zero Dynamics of Two-Wheeled Inverted Pendulum Robot,” *2008 10th IEEE Int. Work. Adv. Motion Control*, pp. 38–43, Mar. 2008.

[33] A. Shimada and N. Hatakeyama, “Movement control of two-wheeled inverted pendulum robots considering robustness,” *SICE Annu. Conf. 2008*, pp. 3361–3366, 2008.

[34] S.-C. Lin, C.-C. Tsai, and H.-C. Huang, “Adaptive Robust Self-Balancing and Steering of a Two-Wheeled Human Transportation Vehicle,” *J. Intell. Robot. Syst.*, vol. 62, no. 1, pp. 103–123, Aug. 2010.

[35] C. Yang, Z. Li, and J. Li, “Trajectory Planning and Optimized Adaptive Control for a Class of Wheeled Inverted Pendulum Vehicle Models,” *IEEE Trans. Cybern.*, vol. 43, no. 1, pp. 24–36, Jun. 2012.

[36] M. Zefran and J. W. Burdick, “Design of switching controllers for systems with changing dynamics,” *Proc. 37th IEEE Conf. Decis. Control*, vol. 2, pp. 2113–2118, 1998.

[37] W. C. Mann, D. Hurren, M. Tomita, and B. Charvat, “An Analysis of Problems with Walkers Encountered by Elderly Persons,” *Phys. Occup. Ther. Geriatr.*, vol. 13, no. 1–2, pp. 1–23, 1995.

[38] N. Tractinsky, “Aesthetics and Apparent Usability: Empirically Assessing Cultural and Methodological Issues,” *Proc. ACM Conf. Hum. Factors Comput. Syst.*, pp. 115–122, 1997.

[39] J. J. Pirkl, *Transgenerational Design: Products for an Aging Population*. 1994.

[40] T. Hirsch, J. Forlizzi, E. Hyder, J. Goetz, J. Stroback, and C. Kurtz, “The ELDer Project: Social, Emotional, and Environmental Factors in the Design of Eldercare Technologies,” *Proc. 2000 Conf. Univers. Usability*, pp. 72–79, 2000.

[41] O. Chuy Jr, Y. Hirata, Z. Wang, and K. Kosuge, “Approach in Assisting a Sit-to-Stand Movement Using Robotic Walking Support System,” *Proc. 2006 IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pp. 4343–4348, 2006.

[42] A. Frizera, R. Ceres, J. L. Pons, A. Abellanas, and R. Raya, “The Smart Walkers as Geriatric Assistive Device. The SIMBIOSIS Purpose,” *Gerontechnology 7*, no. 2, pp. 1–6, 2008.

[43] G. Lee, E.-J. Jung, T. Ohnuma, N. Y. Chong, and B.-J. Yi, “JAIST Robotic Walker Control Based on a Two-layered Kalman Filter,” *2011 IEEE Int. Conf. Robot. Autom.*, pp. 3682–3687, May 2011.

[44] G. Wasson, J. Gunderson, S. Graves, and R. Felder, “Effective Shared Control in Cooperative Mobility Aids,” *Am. Assoc. Artif. Intell.*, pp. 1–5, 2000.

[45] Y. Hirata, A. Muraki, and K. Kosuge, “Motion Control of Intelligent Passive-type Walker for Fall-prevention Function based on Estimation of User State,” *Proc. 2006 IEEE Int. Conf. Robot. Autom.*, pp. 3498–3503, 2006.

[46] G. J. Lacey and D. Rodriguez-Losada, “The Evolution of Guido,” *IEEE Robot. Autom. Mag.*, no. 4, pp. 75–83, 2008.

[47] S. Dubowsky, F. Genot, S. Godding, H. Kozono, A. Skwersky, H. Yu, and L. S. Yu, “PAMM - A Robotic Aid to the Elderly for Mobility Assistance and Monitoring: A ‘Helping-Hand’ for the Elderly,” *IEEE Int. Conf. Robot. Autom.*, vol. 1, pp. 570–576, 2000.

[48] C. Todd and D. Skelton, “What are the main risk factors for falls amongst older people and what are the most effective interventions to prevent these falls ?,” *WHO Reg. Off. Eur. (Health Evid. Network)*, pp. 1–28, 2004.

[49] L. W. Korba, P. J. Nelson, B. A. M. Turpin, and R. Joly, “Exploring Applications of Advanced Robotics Technology in a Long-Term Care Setting - Part 2: Identifying Applications,” *Int. J. Technol. Aging*, vol. 5, no. 1, pp. 21–38, 1992.

[50] D. Choi and J.-H. Oh, “Human-friendly Motion Control of a Wheeled Inverted Pendulum by Reduced-order Disturbance Observer,” *2008 IEEE Int. Conf. Robot. Autom.*, pp. 2521–2526, May 2008.

[51] N. Hogan, “An Organizing Principle for a Class of Voluntary Movements,” *J. Neurosci.*, vol. 4, no. 11, pp. 2745–2754, 1984.

[52] A. Gocmen, “Design of Two Wheeled Electric Vehicle,” no. July, 2011.

[53] S. R. Kuindersma, E. Hannigan, D. Ruiken, and R. A. Grupen, “Dexterous Mobility with the uBot-5 Mobile Manipulator,” *Proc. 14th Int. Conf. Adv. Robot.*, 2009.

[54] D. Ruiken, M. W. Lanighan, and R. A. Grupen, “Postural Modes and Control for Dexterous Mobile Manipulation: the UMass uBot Concept,” *Proc. 13th IEEE-RAS Int. Conf. Humanoid Robot.*, 2013.

[55] D. Linden and T. B. Reddy, *Handbook of Batteries*, 3rd ed. New York: McGraw-Hill, 2002, p. 1200.

[56] N. Zucker, “Design and Manufacture of Handlebars for a Smart Walker,” no. May, 2013.

[57] R. L. Knoblauch, M. T. Pietrucha, and M. Nitzburg, “Field Studies of Pedestrian Walking Speed and Start-Up Time,” *Transp. Res. Rec. 1538*, pp. 27–38, 1996.