A Compact, Powerful, Easy-to-Use Robotics Platform for Use in Autonomous Mobile Robot Competitions

by Clayton Faber, Bachelor of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Degree of
Master of Science
in the field of Electrical Engineering

Advisory Committee:

Dr. George L. Engel, Chair

Dr. Bradley Noble

Dr. Timothy York

Graduate School
Southern Illinois University Edwardsville

August, 2016

ABSTRACT

A COMPACT, POWERFUL, EASY-TO-USE ROBOTICS PLATFORM
FOR USE IN AUTONOMOUS MOBILE ROBOT COMPETITIONS

by

CLAYTON FABER

Chairperson: Professor George L. Engel

Robotics is a fast growing field with a wide variety of applications. With the advent of new, compact, and cheap computing solutions, the user community and hobbyist markets are expanding rapidly. Robotic competitions are also becoming extremely popular among students in engineering programs across the country. Achieving reliable, accurate movement is one of the most challenging aspects of any robot, and most of the current platforms out on the market do not provide the level of performance or the necessary degree of flexibility that autonomous robot competitions at the university level demand.

Using a popular yet inexpensive single-board computer *i.e.* the Texas Instruments' BeagleBone Black, a custom circuit board (the SIUE Robot Cape), and *Makeblock*® (mechanical parts), a small, easy-to-use, yet powerful robotics platform has been developed. The platform supports either four DC motors with quadrature wheel encoders or two stepper motors running open-loop. The ability to independently control four DC motors with encoder feedback affords the user the option of employing four omni-directional wheels in the design of the robot.

In addition to providing reliable movement, the SIUE (Southern Illinois University Edwardsville) Robot Cape, described herein, supports two I2C (Inter-Integrated Circuit)

buses, a RTC (Real-Time Clock), a 16-channel servo motor controller, and a 9 DOF (Degree of Freedom) accelerometer. The system can be powered from standard RC car NiMH batteries. The overall system strikes the right balance between flexibility and ease-of-use. In the pursuit of this end, a graphical user interface was developed to allow users to quickly configure the system.

A very simple demonstration robot, using a pair of DC motors with optical wheel encoders, is described in the thesis. A cascaded PID (Proportional, Integral, Derivative) controller is employed. A PID loop (slave) for each motor controls wheel velocity while a third (master) loop drives the difference in wheel velocities to zero. The output of the master loop adjusts (differentially) the velocity setpoints of the left and right motors. When tested, the robot performed well on tests used to assess robot movement employing odometry.

ACKNOWLEDGEMENTS

First, I want to deeply thank my committee chair, Dr. George Enge,l who has been a wonderful and supportive teacher and without whose help and support this thesis could not be possible.

I would also like to thank my other committee members, Dr. Brad Noble and Dr. Timothy York, since they have been indispensable to my studies. I also want to extend my thanks to other professors who have helped me achieve my goals: Dr. Andy Lozowski, Mr. Steve Muren, Dr. Ying Shang, Dr. Scott Smith, and Dr. Scott Umbaugh.

I am forever indebted to my parents, Terry and Sara Faber, and my sister, Libby Faber, for their unconditional love and support throughout the years. They has given me strength through the good and bad times. Thank you. Without you, this thesis would not have been possible.

I also thank all of my friends for their support and camaraderie throughout this crazy journey. I especially want to thank Christian Cool, Luke Hunt, Tyler Mustard, and Mark Ostroot who have helped me in their own ways. I am only as tall as the shoulders I stand on. You all have my deepest gratitude.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Growing World-Wide Interest in Robotics

As computing power increases and form factor decreases, increased emphasis has been placed on robotics as a means of problem solving. Robots can be used to automate and simplify processes that would otherwise require large amounts of manpower, or they can make a work environment safer by taking the human element out of the equation. The scale of robotics applications ranges from industrial assembly, using large-scale control systems and powerful motors, to hobbyist, using small low-powered computing systems and simple components.

Hobbyist robotics is a growing field accompanied by an ever increasing number of competitions both for academic and recreational purposes and has spurred a thriving industry to support these endeavors. Moreover, robotic competitions are becoming extremely popular among students in engineering programs across the country, and most electrical and computer engineering programs in this country offer courses in robotics. Class projects in robotic courses often involve competition.

Frequently, teams will buy pre-made boards or kits that provide certain specialized functionality such as sensors for environmental feedback or movement via servos or motors. A combination of these are typically used to complete objectives in a course. During competition, a robot is generally required to *move* through a course and complete a series of specified tasks along the way. There is a large variety of styles and mechanisms for achieving accurate and reliable movement but probably the most popular and easiest method is to use a wheeled-type robot utilizing a motor controller of some sort.

Electrical and Computer Engineering students at SIUE (Southern Illinois University Edwardsville) have, for the last 20 years or so, participated in autonomous robotics

competitions sponsored by Region 5 of the IEEE (Institute of Electrical and Electronic Engineers) professional society. Students have generally built a robot from scratch for each competition using available hobbyist parts or have purchased robot kits.

1.2 Motivation for Thesis Work

The author's personal experiences while working on a a robot, pictured in Figure 1.1,for the aforementioned IEEE competition was the motivation for the work described in this thesis. The author was disappointed by the robot's performance in competition. A large amount of time and energy went into trying to deal with simple mechanical issues. Reliable, accurate robot movement was also a major problem.

The author reached the following conclusions: (1) complete robotic platforms currently available are not versatile enough to meet the demands of university-level, autonomous robot competitions and (2) successfully completing contest-specific tasks is sufficiently challenging without students needing to worry about reliable robot movement. After researching available options, it became clear that recent advances in so-called single-board computers (SBC) offer an opportunity to improve the chances of future SIUE robotic teams performing well in autonomous mobile robot competitions.

Past experience suggests that the overall robotics platform must contain the following elements if a team is to be successful in an autonomous mobile robot competition:

- mechanical components to construct a mechanically-sound robot,
- a high-performance, small form factor computer platform that can be used for high-level decision making,
- an intelligent, flexible, dedicated motor controller highly integrated with the high-level compute platform,
- a method to quickly and easily integrate peripheral devices into the robotic system, and

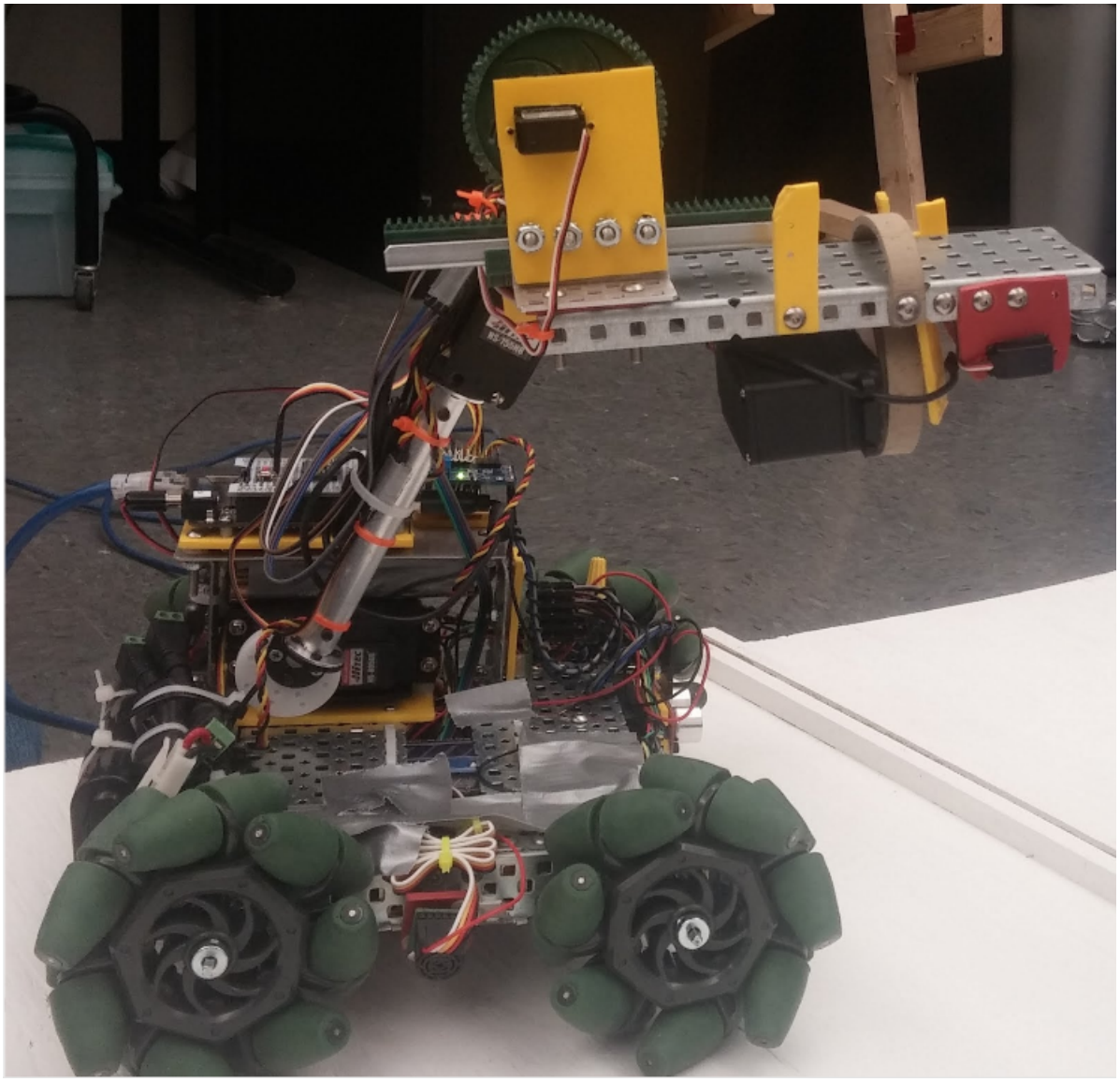

Figure 1.1: Authors's Robot for IEEE Competition

- a quick and convenient way to change system parameters, thus promoting experimentation.

With the requirements above as a guide, the following objectives for the thesis work became clear.

1. Identify a maker of mechanical robot parts which are inexpensive, easy-to-use, and flexible. Students should **not** have to machine their own parts nor should they be constrained to use a prescribed robotics kit.

2. Identify a SBC that would be able to handle the requirements of both movement and higher level decision making.

3. Create a custom motor controller board capable of handling all motor types which could be highly integrated with the processor tasked with higher level decision making. The board should also contain key peripherals and the ability for the user to quickly and easily add additional peripherals.

4. Create a graphical user interface (GUI) that would allow students to quickly configure the key parameters of the robotic system.

In the following section, we briefly describe the various types of motors generally encountered in robotic applications.

1.3 Types of Motors Used in Mobile Robotics

Traditionally, three types of motors are used in robotics: DC, stepper, and servo. In a continuous DC motor, application of power causes the shaft to rotate continually. The shaft stops only when the power is removed, or if the motor is stalled because it can no longer drive the load attached to it. In a stepping motor, applying power causes the shaft to rotate a few degrees, then stop. Continuous rotation of the shaft requires that

the power be pulsed to the motor. A special case of continuous DC motors is the servo motor, which in typical cases combines a continuous DC motor with a feedback loop to ensure accurate positioning. A summary of the advantages and disadvantages of each motor type is presented in Table 1.1.

As is made clear by looking at the table, the three motor types have advantages and disadvantages. The task is for the user to choose the right motor for the application. It is important that all three types of motors be supported. For example, while locomotion is perhaps best handled by a DC or stepper motor, opening and closing a gripper for grabbing objects is best accomplished with a servo motor. All three motors need to be "controlled" and in the following section we review motor controllers.

Type	Advantages	Disadvantages
DC	Wide selection available. Many come with gear boxes so high torque, low-velocity operation is possible.	Poor standards in sizing and mounting arrangements.
Stepper	Does not require gear reduction to power at low speeds. Dynamic braking achieved by leaving coils of stepper motor energized (motor will not turn, but will lock in place).	Poor performance under varying loads. Not great for robot locomotion over uneven surfaces. Consumes high current.
Servo	Inexpensive. Can be used for precise angular control, or for continuous rotation. Available in several standard sizes, with standard mounting holes.	Practical weight limit for powering a robot is about 10 pounds.

Table 1.1: Comparison of Motor Types Used in Robotics.

1.4 Motor Controllers

There are many motor controllers on the market which range from simple H-Bridge drivers to full-blown computing boards that implement some type of control using feedback from the motor. One controller on the market that a past SIUE robotics team used was called the *SSC-32*, manufactured by *Lynxmotion*. This board used an ATMEGA168A to provide PWM (Pulse-Width-Modulation) control to servo motors connected to the board powered with an external power supply.

Commands to this board were received via a serial interface that could be either RS-232 or UART (Universal Asynchronous Receiver Transmitter) compliant based on jumper configuration. One major problem with this board was the lack of feedback. It only supported continuous rotation servos. These motors are adequate for very small, lightweight robots but can produce very little torque, thus rendering them useless for use with larger, heavier robots.

Another controller board used by a past team was the *Serializer* manufactured by *Robotics Connections*. This product was later produced by *cmRobot* under the name *Element* but it is now obsolete. The *Serializer* was sold as a complete robotics board with on-board CPU, memory, I2C (Inter-Integrated Circuit) connections, servo ports, analog inputs, and dual motor H-bridges with encoder input. The board was designed so that users could start working on projects quickly, giving the user immediate access to the functionality of the board.

However, the board existed as a closed system and could only accept a small, fixed set of commands via a serial port. The closed system aspect of the *Serializer* was a major problem for robotic teams. The lack of flexibility frustrated students. When using I2C functions one could use the libraries provided for certain parts but if an unsupported device was used, a read had to be done for each individual byte. The PID functionality of the board was nice but it was difficult to tune the system for different motors and

platforms because it had to be done on every boot of the system.

Furthermore, the serial port, while being ubiquitous in most applications, caused problems for past teams. When using the serial port in a C programming environment, which is typical of most smaller embedded computer systems, data comes back in a character array format followed by a null terminator. If data was being read from the system, the user would have to take the time to first parse the array and then make the conversion from characters to numbers that could be used by the system. This becomes a very tedious process for programmers. Other issues with the board would be that serial communication would often disconnect due to poor connections, causing a massive headache for teams. The board was usable, but it was not as flexible as one would like.

In an attempt to address some of the issues described above, a custom board was created, in the same spirit as that of the *Serializer*, by the author's thesis advisor. This board (see Figure 1.2) used a PSoC 1 microprocessor made by Cypress Semiconductor along with Texas Instrument H-Bridges to control a pair of DC motors with feedback from wheel encoders. The board also supported two servo motors and a single I2C bus. It did **not** support stepper motors. Initially, this PSoC-based board was used for both motor control and high level decision making, but it soon became clear that it lacked sufficient computing power to perform both tasks.

The PSoC 1 is a microprocessor that is highly versatile with programmable digital modules to implement a wide variety of counters, communication protocols and analog devices all controlled by a signal CPU. The PSoC served as the heart of this board, implementing PID functionality for two DC motors with encoder signal feedback for each motor. Along with the motor controller, this board provided line following capability to allow a robot to follow black lines on white surfaces which could be used to keep the robot on course. Other functions of the board included servo motor controllers, I2C interfacing, and GPIO (General-Purpose Input/Output) connections. This board solved some of the

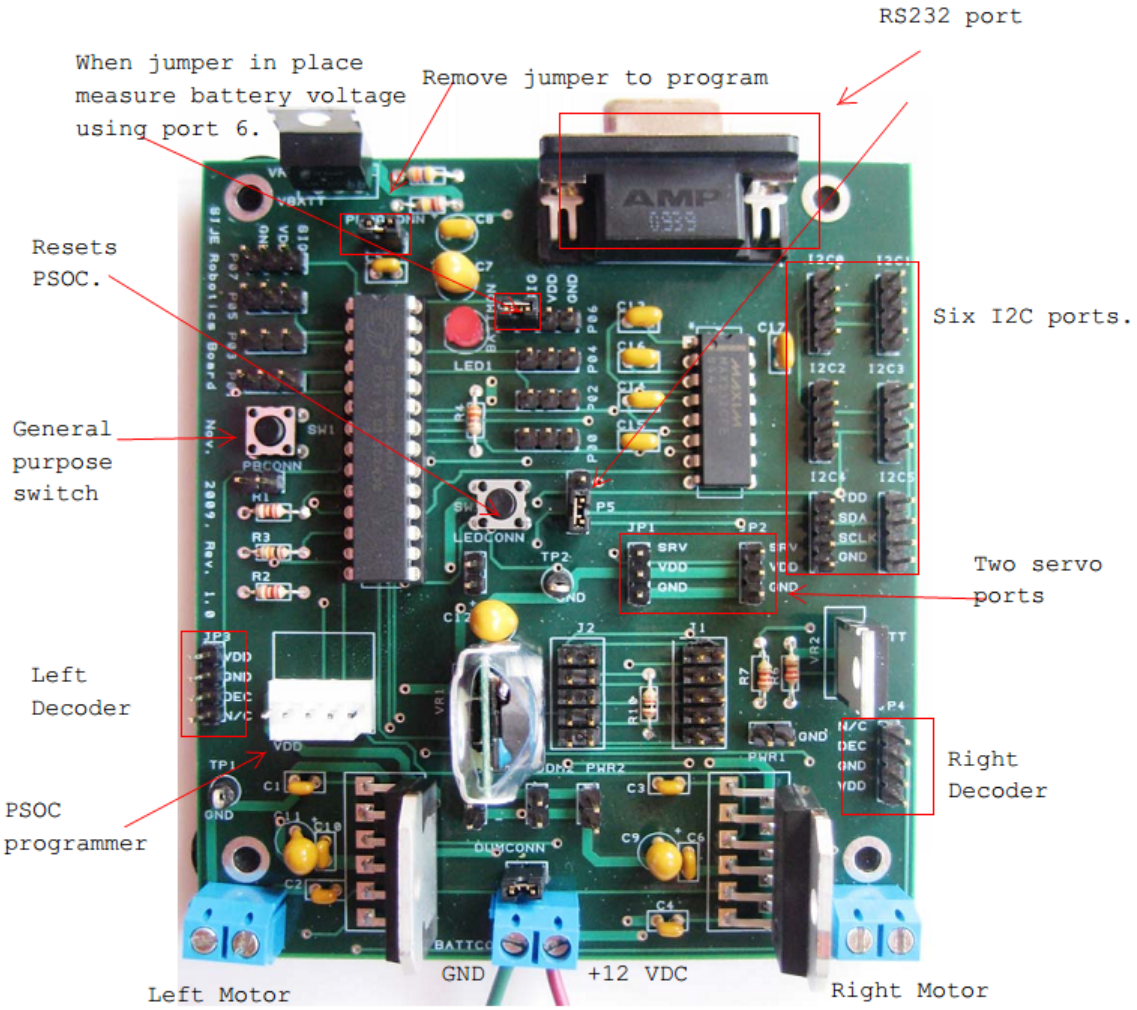

Figure 1.2: SIUE Motor Controller Board Using Cypress PSoC

problems associated with the *Serializer*, but had its own set of issues.

Again, this board used a serial port to communicate with a host and much like the previous board, communication would sometimes be sporadic on the robot due to the potential for connections to break. Also, there was a limited amount of information that was transmitted back to the host system, mostly just acknowledgments that the system had completed the process that it was given. In some situations the host system would hang when waiting for an acknowledgement signal because something had either gone wrong on the PSoC board or the acknowledgement had been lost in transmission. When adjusting and tweaking, it is helpful to view values that were read and calculated, but there was no memory dump for the host system. When values needed to be changed, the program for the PSoC needed to be recompiled in the IDE (Integrated Design Environment) that is only available through Microsoft Windows and then programmed through a USB (Universal Serial Bus) programmer.

This approach did not allow for on-the-fly tweaking of PID (Propotional, Integral, Derivative) parameters which resulted in a slow tuning process for the PID controller. Things like wheel diameter, ticks per rotation, and PID gain all had to be reprogrammed into the PSoC. A more useful approach is **not** to use serial communication but rather have one system that handles both the high level decision making and handling of the motors and other peripheral devices. As mentioned above, the PSoC lacked sufficient power to handle both tasks. A more powerful system such as a SBC is needed if one system is to handle all of these tasks.

One motor controller board that is designed to use a SBC (Single-Board Computer) is the DMCC (Dual Motor Controller Cape). It was designed by *Exadler Technologies Inc* for the Texas Instruments' BeagleBone Black and is described in detail in a later section. This board (see Figure 1.3 is used to control two motors with H-Bridges that can supply up to 7 Amps at 5 to 28 volts [Tan, 2016].

There is no datasheet available for the board, but there are github repositories. After some exploring of the repositories, it appears that this cape is controlled by a dsPIC33FJ32MC304 by Microchip which is a 16-bit micro-controller with 8 PWM channels and two quadrature encoder interfaces. It appears that the micro-controller is accessed by the BeagleBone Black via I2C communication for setup and control.

The software is a collection of functions to access the I2C bus and send data to the micro-controller for setup and operation. PID control is implemented on the micro-controller and can be used to control either the velocity or position of the motors. This is again a closed system, similar to the *Serializer*, because the firmware on the micro-controller is pre-loaded and source code is not provided. It is only the DMCC that can truly be considered a direct competitor to the board designed for this thesis. In the following section, we will review SBCs *i.e* single board computers.

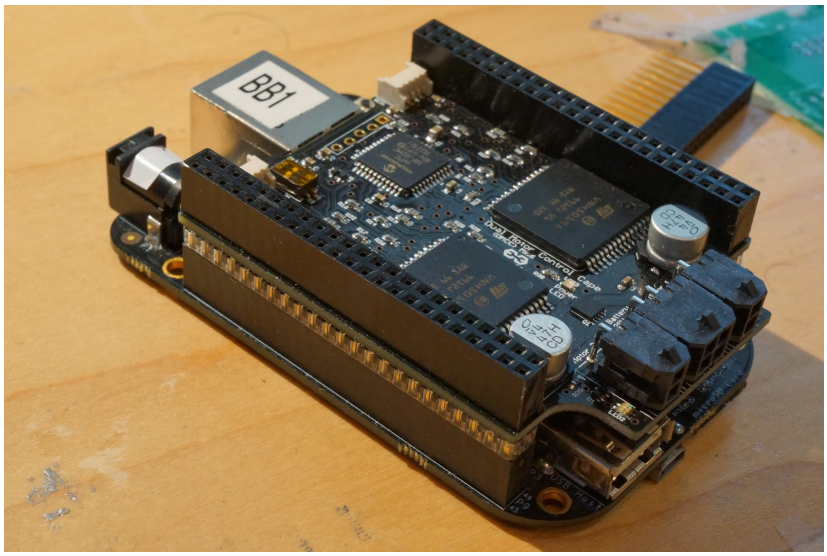

Figure 1.3: Dual Motor Controller Cape (DMCC)

1.5 Single-Board Computers

Single-board computers (SBCs) have been around for quite some time and have recently grown in popularity as integrated circuits decrease in both size and price. One of the most prolific devices currently out on the market is the *Arduino* which is more aptly described as a micro-controller rather than a SBC, but it is often mistaken for one. While the Arduino is powerful, complete with analog and digital I/O, it has limits when it comes to intense calculations or tasks that are better suited for full-blown computers.

The Raspberry Pi is one of the most popular SBCs currently out on the market, and it has taken both the hobbyist and teaching communities by storm. The small size of the system along with the low power consumption, which permit it to operate off a USB connection, make it easy to use and accessible to the general public. The system runs a custom version of Debian called Raspbian which is a widely used Linux distribution and comes with the ability to be hooked up to a monitor with keyboard and mouse and used as a full-fledged computer.

The board also has digital I/O pins which can be used to implement a wide variety of protocols such as I2C or UART as well as implementing general digital I/O functions. The combination of a full-blown operating system along with microprocessor functionality makes for a potent combo in a robotics platform. Over the last couple of years, upgraded models have been released. The Raspberry Pi 2 and Pi 3 have increased performance over the original Pi, and the Pi Zero which is less than half the size of the original Pi. A comparison of the various Raspberry Pi offerings is presented in Figure 1.4 [Benchoff, 2016]. While these devices are very powerful, especially the recently released Raspberry Pi 3, they are actually better suited for multimedia applications such as console/arcade emulation and media box creation than for mobile robotic applications.

A direct competitor to the Raspberry Pi is the BeagleBone Black which will be described in the following section. It is the BeagleBone Black that was selected for use in

	Raspberry Pi 3 Model B	Raspberry Pi Zero	Raspberry Pi 2 Model B	Raspberry Pi Model B+
Introduction Date	2/29/2016	11/25/2015	2/2/2015	7/14/2014
SoC	BCM2837	BCM2835	BCM2836	BCM2835
CPU	Quad Cortex A53 @ 1.2GHz	ARM11 @ 1GHz	Quad Cortex A7 @ 900MHz	ARM11 @ 700MHz
Instruction set	ARMv8-A	ARMv6	ARMv7-A	ARMv6
GPU	400MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV
RAM	1GB SDRAM	512 MB SDRAM	1GB SDRAM	512MB SDRAM
Storage	micro-SD	micro-SD	micro-SD	micro-SD
Ethernet	10/100	none	10/100	10/100
Wireless	802.11n / Bluetooth 4.0	none	none	none
Video Output	HDMI / Composite	HDMI / Composite	HDMI / Composite	HDMI / Composite
Audio Output	HDMI / Headphone	HDMI	HDMI / Headphone	HDMI / Headphone
GPIO	40	40	40	40
Price	$35	$5	$35	$35

Figure 1.4: Raspberry Pi Offerings

this thesis work. The rationale for this decision will also be presented.

1.6 Texas Instruments' BeagleBone Black

One SBC offering from Texas Instruments, through the BeagleBoard Foundation, is a SBC called the BeagleBone Black. The main CPU (Central Processing Unit) is an ARM A8 single core processor running at a 1 GHz clock speed with 512 MB of memory and with a 4 GB eMMC holding the operating system. The device has peripherals for Ethernet, microSD, micro HDMI, a debug serial header, and two separate USB ports. One USB port is a standard type A format used as a host connection and the other is a USB type A Mini format which is configured as a USB network adapter [BeagleBoard.org, 2016].

The Beaglebone Black runs Debian for its operating system which comes with all the free utilities associated with Linux. One of the major draws for this system is the two 46 female pin headers on the sides of the board (referred to as the P8 and P9 connectors) with the ability to support up to 65 digital I/O pins along with other connections to ground, 3.3 volt supply, 5 volt supply, and a 1.8 volt tolerant ADC (Analog-to-Digital

Converter). The digital I/O pins can be multiplexed internally to different digital blocks for different behavior such as I2C, SPI (Serial Peripheral Interface), UART, and GPIO. In addition there are two RISC (Reduces Instruction Set Computer) processors that can run independent from the main processor called PRUs (Programmable Real-time Units).

The PRUs were designed as a way to give users the ability to access real-time functionality within the Linux environment. The PRUs have a 32-bit RISC architecture. THe PRUs operate at a clock speed of 200 MHz with single cycle instructions. Each PRU has access to an 8 KB instruction memory and an 8 KB data mmeory. There is also a 12 KB shared memory that can be accessed by both PRUs **and** the ARM core running Linux.

The PRUs are also equipped with external GPIO pins that can be mutliplexed out to the the P8 and P9 headers for fast acting output and input compared to that availabe from the Linux system. They are also equipped with an interrupt controller (in reality a semaphore system rather that an interrupt in the conventional sense) with the ability for the two PRUs to interrupt each other or to receive and send interrupts to the host Linux system. This allows for communication between the two PRUs.

For example, if one PRU is calculating data and the other is working as a hardware interface, the interface unit can interrupt (*i.e.* notify) the calculating unit when there is new data to process. This combination of dedicated processing inside a main computing core running a full-fledged operating system makes a potent combination for robotics applications. The existence of these two PRUs is why the BeagleBone Black was chosen by the author over the Raspberry Pi 3 with its extra processing power.

1.7 Object and Scope of Thesis

The goal of this thesis is to integrate a single-board computerwith a motor controller/periphearal board to make it easier for future robotics teams and hobbyists to quickly assemble a mobile robot. The hope is that the work presented in the thesis can

be used by student groups at SIUE in the future as a solid foundation for the creation of robots which can be competive in autonomous robotic contests.

In Chapter 1 of this thesis, we have discussed the growing interest in mobile robotics, the various types of motors which are used in robotic applications, and the motivating forces for this work. We then went on to review motor controller boards and single board computers. We concluded with an overview of the Texas Instruments' BeagleBone Black single board computer which will be used throughout this work.

Chapter 2 presents a detailed decription of the design of a daughter board (known as a "cape") for the Beaglebone Black. The SIUE Robot Cape described in Chapter 2 supports either four DC motors with quadrature wheel encoders or two stepper motors running open-loop. The ability to independently control four DC motors with encoder feedback affords the user the option to employ four omni-directional wheels in the design of a robot. In addition to providing reliable movement, the SIUE robot cape provides access to two I2C buses, a real-time clock, a 16-channel servo motor controller, and a 9 DOF (Degree of Freedom) accelerometer.

In Chapter 3 the software developed for the two BeagleBone Black PRUs (Programmable Real-Tine Units) is described. The chapter starts with a description of the contents of the memory which is shared between the two PRUs and the ARM core. This shared memory provides tight integration between the user code running on the ARM and the motor controller code running on the two PRUs eliminating the need for serial communication between compute platform and motor controller! The chapter goes on to describe the PID algorithms used to ensure high quality movement employing odometry provided by optical wheel encoders. The code developed for each of the two PRUs is decribed in detail.

Chapter 4 presents the development of demonstration robot used to evaluate the quality of movement. The chapter begins with a description of the robot frame using

MakeBlock mechanical components. The chapter goes on to describe the routines in a series of libraries which were created to jump start future robotic teams. The chapter continues with a description of the Graphical User Interface (GUI) which was created to allow users to quickly change system parameters. The chapter concludes with a discussion of a demonstration program and present results of tests which were run to quantify the quality of the robot movement. A summary, concluding remarks, and some suggestions for future work are provided in Chapter 5.

CHAPTER 2

SIUE ROBOT CAPE DESIGN

2.1 Hardware Design Objectives

In this chapter the design of a daughter board that can be plugged into the BeagleBone Black, using the P8 and P9 connectors, is described. The daughter board provides the necessary hardware to interface and control the motors and other popular peripherals. Daughter boards that give added functionality to the BeagleBone Black are called *capes* which have their own unique overlay to describethe pin multiplexing that is required for the board to connect correctly to the various peripheral devices.

The cape overlay system has the added benefit of having a plug and play functionality which on boot, if the cape is present in the firmware, can be loaded. First, the system level design of the board will be presented and then a short discussion of how the overlay system for a cape works is given. We conclude with a discussion of the peripherals available on the cape.

The main goal of the SIUE Robot Cape was to provide the user with motor drivers that are capable of handling modest amounts of power while retaining a small form factor. In addition to motor drivers, appropriate level shifting was required by the motor encoder signals to ensure that only a 3.3 volt signal would be presented to the BeagleBone Black input pins. Any voltage higher than 3.3 volts on any digital pin pose a hazard to the Black. Additional peripheral devices were added to the board if they were deemed either extremely useful for robot designers and/or provided infrastructure for commonly used communication protocols.

Physically, we desired a two-layer board with one layer being a ground pour. We used surface mount components on the top layer which helped save on space because of their typical small form factor compared to through hole parts. Finally, the board was designed

in a manner so as to isolate the digital and high power circuits from one another as much as possible by having the high powered components located on an outcropping away from the digital electronics so that there would be a reduced chance of crosstalk which might in turn cause unexpected problems.

2.2 System Overview

The final version of the SIUE robot cape is pictured in Figure 2.1. Onboard chips include H-Bridges for the motor driving circuits, buffers for the inputs and outputs of the PRU pins, a quad XOR gate for use with the quadrature encoder input signals, an I2C level shifter, a pair of LEDs, two momentary switches, and an EEPROM (Electrically Erasable Programmable Read Only Memory) for board loading. In addition there are two additional break out boards, an accelerometer and a real time clock, along with an additional EEPROM that can be accessed through the I2C bus and used by the ARM or PRUs if desired. There is also a place to add a 16-Channel servo driver board. After a discussion of board setup and configuration, the peripherals available on the board will be described in more detail.

2.3 LINUX Device Tree Overlay

As just mentioned, the cape requires that an overlay be loaded into the Linux device tree to let the operating system know how to multiplex various pins in the system and load in the necessary drivers for the devices that are used by the board. The device tree is a structure used by hardware developers to describe hardware available to the system that is un-discoverable by the operating system [eLinux, 2016].

In total the SIUE Robot cape uses 21 different pins on the BeagleBone Black in various pullup and pulldown configurations presented in Figure 2.1. The cape has an installed EEPROM chip on board with the board name and a description of the pins used by the board. The EEPROM memory map is defined in Figure 2.2 which is taken from the

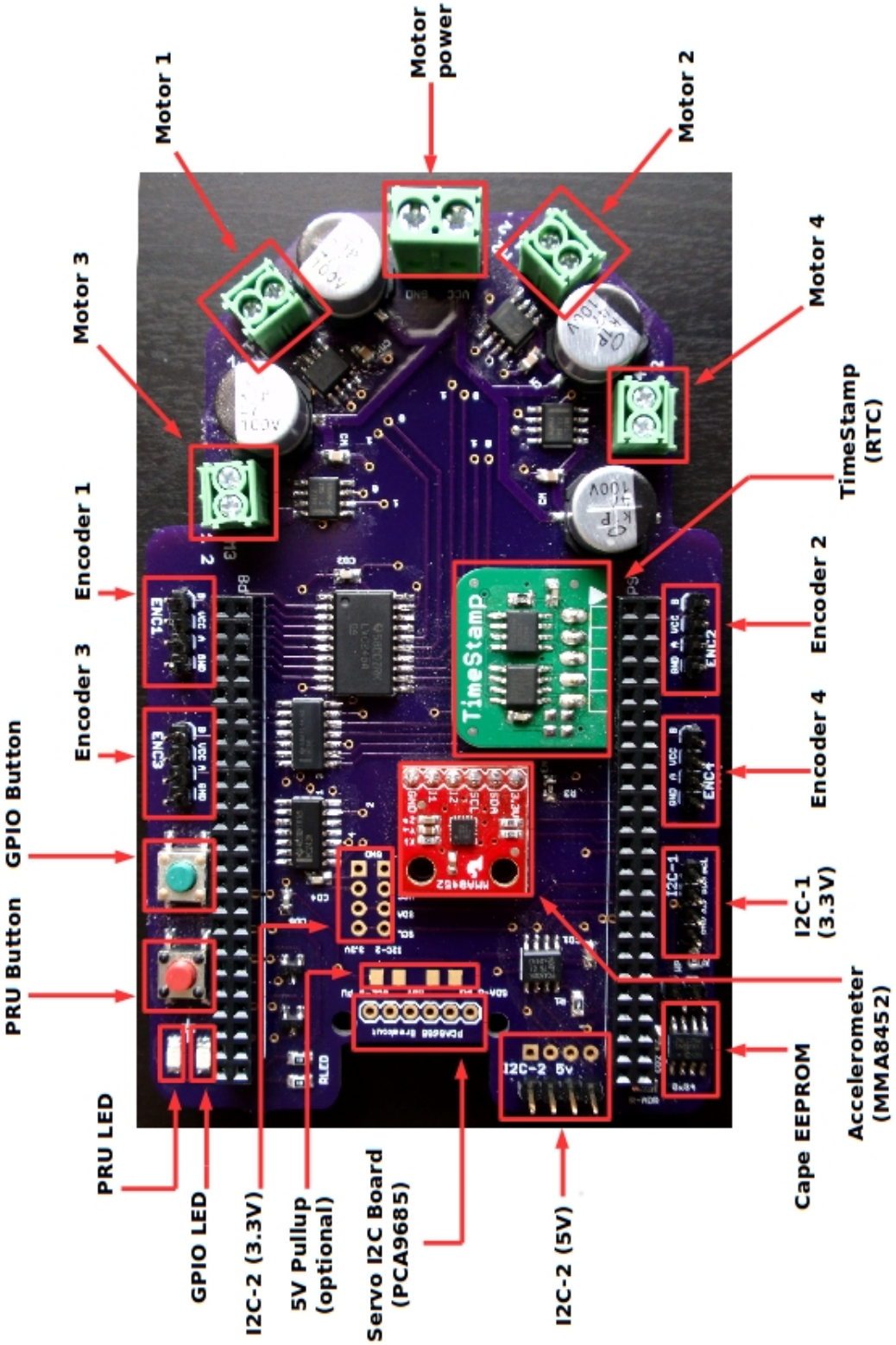

Figure 2.1: SIUE Robot Cape

BeagleBone Black System Reference Manual [Coley, 2014].

Header	Pin #	Signal	Configuration	Pullup/Pulldown	Input/Output	Mux Mode	Address	$PINS	Purpose
P9	17	I2C-2	0x72	Pullup	In/Out	2 - I2C1	0x15C	87	Second I2C bus SCL
P9	18	I2C-2	0x72	Pullup	In/Out	2 - I2C1	0x158	86	Second I2C bus SDA
P9	22	IN-2	0x27	Pulldown	Input	7 - GPIO0[2] (@2)	0x150	84	Accelerometer interrupt (GPIO)
P9	25	IN-1	0x26	Pulldown	Input	6 - PRU0 r31.t7	0x1AC	107	Accelerometer interrupt (PRU0)
P9	27	DRV-EN	0x17	Pullup	Output	7 - GPIO3[19] (@115)	0x1A4	105	Output to enable buffers
P8	11	PRU_LED	0x0E	Disabled	Output	6 - PRU0 r30.t15	0x034	13	PRU controlled LED
P8	12	GPIO_LED	0x0F	Disabled	Output	7 - GPIO1[12] (@44)	0x030	12	GPIO controlled LED
P8	15	GPIO_SW	0x27	Pulldown	Input	7 - GPIO1[15] (@	0x03C	15	GPIO input push button
P8	16	PRU_SW	0x26	Pulldown	Input	6 - PRU0 r31.t14	0x038	14	PRU input push button
P8	27	ENC1	0x2E	Pulldown	Input	6 - PRU1 r31.t8	0x0E0	56	PRU encoder input - Motor 1
P8	28	ENC2	0x2E	Pulldown	Input	6 - PRU1 r31.t10	0x0E8	58	PRU encoder input - Motor 2
P8	29	ENC3	0x2E	Pulldown	Input	6 - PRU1 r31.t9	0x0E4	57	PRU encoder input - Motor 3
P8	30	ENC4	0x2E	Pulldown	Input	6 - PRU1 r31.t11	0x0EC	59	PRU encoder input - Motor 4
P8	39	M4-0	0x0D	Disabled	Output	5 - PRU1 r30.t6	0x0B8	46	PRU PWM control - Motor 4 input 0
P8	40	M4-1	0x0D	Disabled	Output	5 - PRU1 r30.t7	0x0BC	47	PRU PWM control - Motor 4 input 1
P8	41	M2-0	0x0D	Disabled	Output	5 - PRU1 r30.t4	0x0B0	44	PRU PWM control - Motor 2 input 0
P8	42	M2-1	0x0D	Disabled	Output	5 - PRU1 r30.t5	0x0B4	45	PRU PWM control - Motor 2 input 1
P8	43	M1-0	0x0D	Disabled	Output	5 - PRU1 r30.t2	0x0A8	42	PRU PWM control - Motor 1 input 0
P8	44	M1-1	0x0D	Disabled	Output	5 - PRU1 r30.t3	0x0AC	43	PRU PWM control - Motor 1 input 1
P8	45	M3-1	0x0D	Disabled	Output	8 - PRU1 r30.t6	0x0A0	40	PRU PWM control - Motor 4 input 1
P8	46	M3-0	0x0D	Disabled	Output	8 - PRU1 r30.t7	0x0A4	41	PRU PWM control - Motor 4 input 0

Table 2.1: BeagleBone Black Pins Used by SIUE Robot Cape

This allows the board to be loaded at boot time by matching the correct compiled device tree overlay (device tree blob) located in the /sys/firmware folder. However, some changes need to be made to the uboot bootloader environment before this process can happen seamlessly. When a device tree overlay is loaded, that overlay has claim on the pins used by the different devices described by the overlay.

A large portion of input/output PRU pins are included in the cape are claimed by the HDMI (video interface) and McASP (audio interface) normally and the HDMI overlay must be disabled. This is done by editing the uEnv.txt located in the /boot folder and uncommenting a line designated for removal if the HDMI and audio overlays are not needed.

The last step requires editing the uBoot file itself so the board overlay is available to the bootloader. This is the result of a "chicken and the egg" problem with the BeagleBones EMMC (Embedded Multi-Media Card) which holds the actual Linux kernel and the bootloader which is running its own program to set up the boot environment. The EMMC is not actually native to the system and needs to be made available to the system

Name	Offset	Size (bytes)	Contents
Header	0	4	0xAA, 0x55, 0x33, 0xEE
EEPROM Revision	4	2	Revision number of the overall format of this EEPROM in ASCII =A1
Board Name	6	32	Name of board in ASCII so user can read it when the EEPROM is dumped. Up to developer of the board as to what they call the board..
Version	38	4	Hardware version code for board in ASCII. Version format is up to the developer. i.e. 02.1…00A1….10A0
Manufacturer	42	16	ASCII name of the manufacturer. Company or individual's name.
Part Number	58	16	ASCII Characters for the part number. Up to maker of the board.
Number of Pins	74	2	Number of pins used by the daughter board including the power pins used. Decimal value of total pins 92 max, stored in HEX.
Serial Number	76	12	Serial number of the board. This is a 12 character string which is: WWYY&&&&nnnn where: WW = 2 digit week of the year of production YY = 2 digit year of production &&&&=Assembly code to let the manufacturer document the assembly number or product. A way to quickly tell from reading the serial number what the board is. Up to the developer to determine. nnnn = incrementing board number for that week of production
Pin Usage	88	148	Two bytes for each configurable pins of the 74 pins on the expansion connectors MSB LSB Bit order: 15 14 ……..……1..0 Bit 15……..Pin is used or not……..…..0=Unused by cape 1=Used by cape Bit 14-13……Pin Direction……..…..1 0=Output 01=Input 11=BDIR Bits 12-7……Reserved……should be all zeros Bit 6……..Slew Rate ……..0=Fast 1=Slow Bit 5……..Rx Enable……..0=Disabled 1=Enabled Bit 4……..Pull Up/Dn Select……..0=Pulldown 1=PullUp Bit 3……..Pull Up/DN enabled……..0=Enabled 1=Disabled Bits 2-0 ……Mux Mode Selection……..Mode 0-7
VDD_3V3B Current	236	2	Maximum current in milliamps. This is HEX value of the current in decimal 1500mA=0x05 0xDC 325mA=0x01 0x45
VDD_5V Current	238	2	Maximum current in milliamps. This is HEX value of the current in decimal 1500mA=0x05 0xDC 325mA=0x01 0x45
SYS_5V Current	240	2	Maximum current in milliamps. This is HEX value of the current in decimal 1500mA=0x05 0xDC 325mA=0x01 0x45
DC Supplied	242	2	Indicates whether or not the board is supplying voltage on the VDD_5V rail and the current rating 000=No 1-0xFFFF is the current supplied storing the decimal equivalent in HEX format
Available	244	32543	Available space for other non-volatile codes/data to be used as needed by the manufacturer or SW driver. Could also store presets for use by SW.

Table 2.2: EEPROM Memory Map

through the use of an overlay. Overlays loaded on boot are only done once *i.e.* before the unpacking of the kernel on the EMMC happens. So the SIUE Robot Cape device tree overlay needs to be in the bootloaders firmware folder at boot time and not solely in the Linux firmware folder.

This is fixed by using a script provided by the BeagleBone developers which uses *initramfs-tools* to update the boot loader. A script must be placed in the initramfs-tools/scripts directory telling the tools to copy the firmware folder containing all the device tree blobs into the the firmware folder for the bootloader. Once this is done the bootloader now knows about the custom cape that it is attached to via the EEPROM and can load it properly.

2.4 I2C Interface

On the BeagleBone Black there are three available I2C busses but only two are accessible to the user on the P8-P9 headers. These two buses I2C-1 and I2C-2 are claimed for use by the SIUE Robot cape. In the current Debian 3.8 kernel there is a slight confusion regarding the naming of these buses. In hardware schematics they are labeled I2C1 and I2C2. When these devices mount, their names become flipped I2C1 mounts in /dev as I2C-2 and I2C2 mounts in /dev as I2C-1.

To reduce this confusion, they are labeled on the board with their I2C /dev names and henceforth in this paper they will be referred to as I2C-1 and I2C-2. The I2C-1 bus plays an important role in the loading of capes, because it is the bus that gets queried at certain addresses for attached EEPROMs and attempts to read the contents.

There are only 4 addresses that are queried for custom capes. These 8-bit I2C addresses are 0x54, 0x55, 0x56, and 0x57. This means that only 4 capes can be loaded at boot at any given time but more capes can be loaded later by the user. I2C-1 is also where the on-cape I2C devices are located. These I2C devices are the accelerometer breakout board and the real time clock with EEPROM (called TimeStamp) breakout board.

A listing of addresses that are either taken or reserved is presented in Figure 2.3. The I2C-2 bus is broken out in two different spots with two voltage logic levels (3.3 Volts and 5 Volts) and have no attached I2C devices. A level translator is provided on this bus. It is the the PCA9306 which implements the Phillips specifications for I2C level translation and allows for devices that use 0 to 5 volt logic levels to talk to 0 to 3.3 volt systems such as the BeagleBone [NXP, 2014].

On the board there are optional placements for SCL and SDA pull-up resistors, but often breakout boards using I2C have attached pullups so it may or may not be wanted. The final I2C device on the board is the PCA9685 breakout board which can control up to 16 servo motors. Since this device operates in a 0 to 5 volt logic environment, its connection to the level translator is on the high (5 Volt) voltage side.

Device	I2C-1 Address (7-bit)
Accelerometer	0x1D
TimeStamp RAM	0x50
EEPROM Cape 1	0x54
EEPROM Cape 2	0x55
EEPROM Cape 3	0x56
EEPROM Cape 4	0x57
TimeStamp RTC	0x68

Table 2.3: I2C-1 Addresses

2.5 H-Bidge Circuits

The H-Bridge chosen for this application is a Texas Instruments DRV8872. This can be used as DC motor driver with a dual input control scheme, current sensing capabilities, and fault feedback. The H-Bridge circuit can drive 6.5 to 45 volts at 3.6 amperes peak in both a forwards and backwards rotation depending on the input signals [Texas-Instruments, 2016].

This H-Bridge can also perform a hard brake or a slow decay (coast) based again on the input signals. Figure 2.2 shows a logic block diagram of the chip, indicating

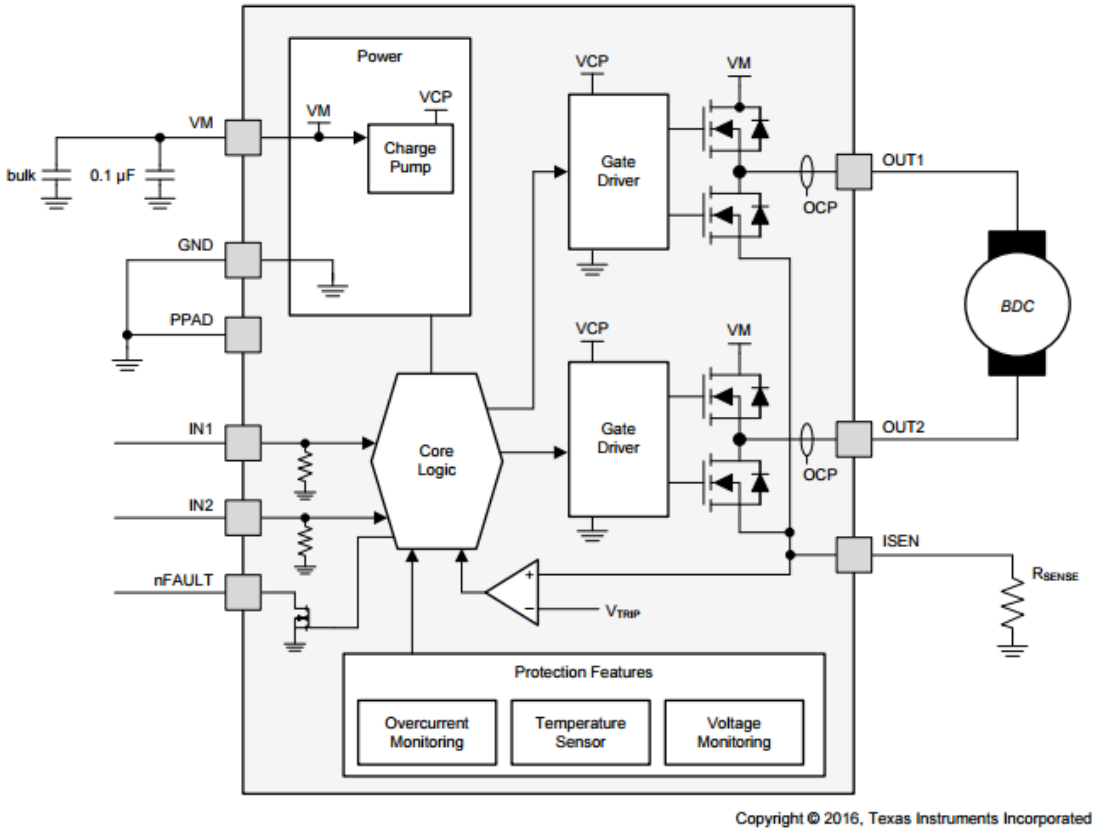

Figure 2.2: DRV8872 Functional Block Diagram

some of the functionality of the chip [Texas-Instruments, 2016]. There are a number of other features proveded by the DRV8872 shown in the block diagram such as overcurrent monitoring, temperature monitoring, and voltage monitoring which control the value of nFAULT which is the feedback pin on this chip. This functionality is not supported in the current version of this board and the pin is grounded. There is also an ISEN pin dedicated to current sensing, but this pin is grounded to achieve the maximum peak current possible.

Physically, the chip comes in a 8-SOIC package with an extra pad on the bottom for thermal relief to a ground plane, which is dubbed the PowerPAD by Texas Instruments. This chip was mostly chosen for its small form factor and high current properties. In fact, the chip is physically smaller than the 47 μF bulk capacitors in parallel to it.

On the board the input signals are driven by an 8-bit buffer and connected to motor power through wide traces via the power supply connection labeled "Motor Voltage" in Figure 2.1. Beneath the outcropping section is the ground pour which has to be at the same ground potential as the digital side of the board. To help prevent potential surges of power crossing over to the digital side, two ground pours were created and thin connections made between making them the same potential but restrictive in a high current sense.

Large bulk capacitors are used along with smaller decoupling capacitors to help prevent fluctuations in power when the motors are active. The final output signal for these drivers are the individual motor outputs labeled in Figure 2.1 which are meant for the two wires of a DC motor or one drive pair for a stepper motor.

2.6 Buffers and XOR Gate

This board comes with two buffer chips: CD74HC243 (4-bit) and CD74HC245 (8-bit) and one quad XOR gate package (SN74HC86). These chips were added to handle situations where signals on boot or default values affect either the motors or the boot

process of the BeagleBone. When the BeagleBone powers up some of the pins used by the PRUs in the final application are used to determine boot state.

For example if one pin is pulled high on this bus it will attempt to boot from the SD card slot rather than the built in EMMC storage. Additionally the default state of these pins after the boot process may be left floating at voltages that would trip the logic levels of the H-bridges causing the motors to spin on boot and shortly after.

The wheel encoders suffer from a similar issue. If a pin is to be an input during the boot process but it is attached to an encoder there is a chance for it to be high which in turn could interfere with the boot process. To address this situation buffer chips are used so that when the BeagleBone is in a state that would not require pin outputs or inputs, the buffers keep the system side in a high impedance state when not used. In this way the H-bridges will never see an errant voltage level without being set properly first and the encoder input pins would not see a signal unless they were ready to.

The high impedance state is controlled by a GPIO pin that is pulled up to 3.3 volts. During boot this makes the voltage high enough to trigger the high impedance state on the buffers and only by setting the GPIO pin to logic low state (*i.e.* 0) can the buffers be activated. The 8-bit buffer controls the H-bridges by passing along the output of the PRUs and the 4 bit buffer handles passing the encoder signal to the input of the PRU after being handled by the quad XOR gate.

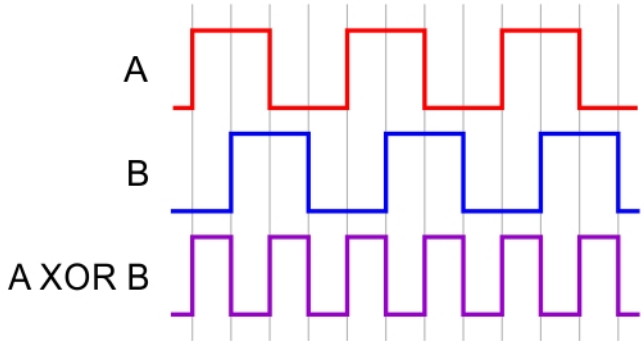

Figure 2.3: Encoder Waveforms Using XOR Gate

The quad XOR gate receives the signal from the motor encoders and passes it to the 4-bit buffer chip. The role of the XOR gate is twofold: (1) provides the ability to effectively read a pair of quadrature encoder signal without the actual need for inputs for both the A and B channels, and (2) provides a means of level translate a 5 volt signal to 3.3 volts.

Quadrature encoders are fairly common. The term implies that there are two signal in the form of square waves with a 90 degree phase shift between them. As the motor rotates and depending on which channel is read first, the rotation direction can be determined. However, if the motor direction is being controlled it becomes a pointless feature and just ends up taking an extra input to get full resolution of the encoder ticks. Using an XOR gate for the two input channels (A and B), full resolution of the encoder ticks can be achieved as there will be a pulse for every positive edge of the A and B signals as shown in Figure 2.3 [Pololu, 2016].

The input pins on the SN74HC86 are up to 7 volt tolerant and their output voltage level is determined by the supply voltage. By hooking up the supply voltage to 3.3 volts, the SN74HC86 performs level translation. It accepts 5 Volt signals and translates them to 3.3 Volts.

2.7 Real-Time Clock

The *TimeStamp* board was developed by Dr. Noble of SIUE for a water sensor application but works well as a general RTC (Real-Time Clock) breakout board. *TimeStamp* also comes equipped with a 16 KByte FRAM (Ferro-electronic RAM) chip for storing constants if necessary and a battery holster for a 3 volt BR1225 coin battery to continue RTC operation without an external power supply.

The RTC used in this module is a DS3231 clock module made by Maxim which has a time accuracy of ± 0.432 seconds a day and leap year compensation up until the year 2100 [Maxim, 2015]. Information is retrieved and updated through typical I2C operations.

This board was added later in the design and was also chosen because of its small form factor and how easy it was to integrate into the cape design.

2.8 Accelerometer

Typical accelerometers available on the market are small devices made to be utilized in cell phone applications and as a consequence are incredibly small and hard to hand solder. Using a breakout board is very typical in these situations. A company called Sparkfun sells an I2C accelerometer breakout board using the MMA8452. It is a three axis accelerometer with two output interrupt signals that can be tied to a number of different events.

The device has an output data rate ranging from 1.56 Hz to 800 Hz and user selectable full scales of ±2g, ±4g, and ±8g as well as the options to output high pass filtered data or unfiltered [Freescale-Semiconductor, 2013] data. The interrupt signal can be connected to a data ready flag, freefall alert, or even an orientation signal to name a few.

On the board the INT1 and INT2 signals are connected to a PRU input and a GPIO input respectively. Currently they are unsupported in our software. The accelerometer is useful in a robot because it can potentially provide information on how it is orientated, the direction it is traveling in space, or if a collision has happened. When developing in the future such a tool can be used to supplement the encoder feedback by observing things like slippage.

2.9 Servo Motor Controller

As explained in Chapter 1, it is important that the cape support all three types of motors. In order to support servo motors, the board includes a place to plug in a 16-Channel 12-bit PWM/Servo Driver. The PCA9685 is available from *Adafruit* at at a cost of about fifteen dollars. The breakout board was selected because of its following features.

- I2C-controlled PWM driver with a built-in clock,
- 5 Volt compliant,
- adjustable frequency PWM up to about 1.6 KHz,
- 12-bit resolution for each output *i.e.* (for servos that means about 4us resolution at a 60 Hz update rate),
- configurable push-pull or open-drain output,
- and output enable pin to quickly disable all the outputs.

2.10 User LEDs and Switches

The SIUE Robot Cape also provides the user with 2 momentary (pushbutton) switches and 2 LEDs (Light Emitting Diodes). One LED-switch pair is connected to GPIO pins. The other pair is attached to PRU fast I/O pins. The pair connected to GPIO pins can be accessed by either the PRU or by the ARM. The other pair is accessible only by PRU 1.

LEDs and pushbutton switches are some of the most rudimentary debugging tools and can be useful when a terminal is not available to the user. In past competitions, SIUE teams have had to implement a start button for the robot that would start the execution of the main program, and frequently blinking an LED is a required task which allows the judges to know when the robot has completed the competition or a specific task.

For the switches, when pressed a 3.3 volt signal is connected to an input pin in a pull-down configuration. The LEDs are controlled by N-Channel MOSFET (BS-223) whose gate is connected to an output pin, and the LED in series with a resistor are placed between the 3.3 volt supply and the drain. When the associated output pin is brought high, the MOSFET conducts and current flows from the drain to the source which in turn causes current to flow through the LED, thereby turning it on.

2.11 Schematic and PCB Layout

The schematics for the SIUE Robot Cape are presented in Figure 2.5 and Figure 2.6. Each IC in the schematic has its own decoupling 0.1μF capacitor to prevent supply voltage fluctuations. The printed circuit board (PCB) layout is highlighted in Figure 2.4. Schematic entry and PCB layout was performed using EAGLE 7 software from *CadSoft Computer*.

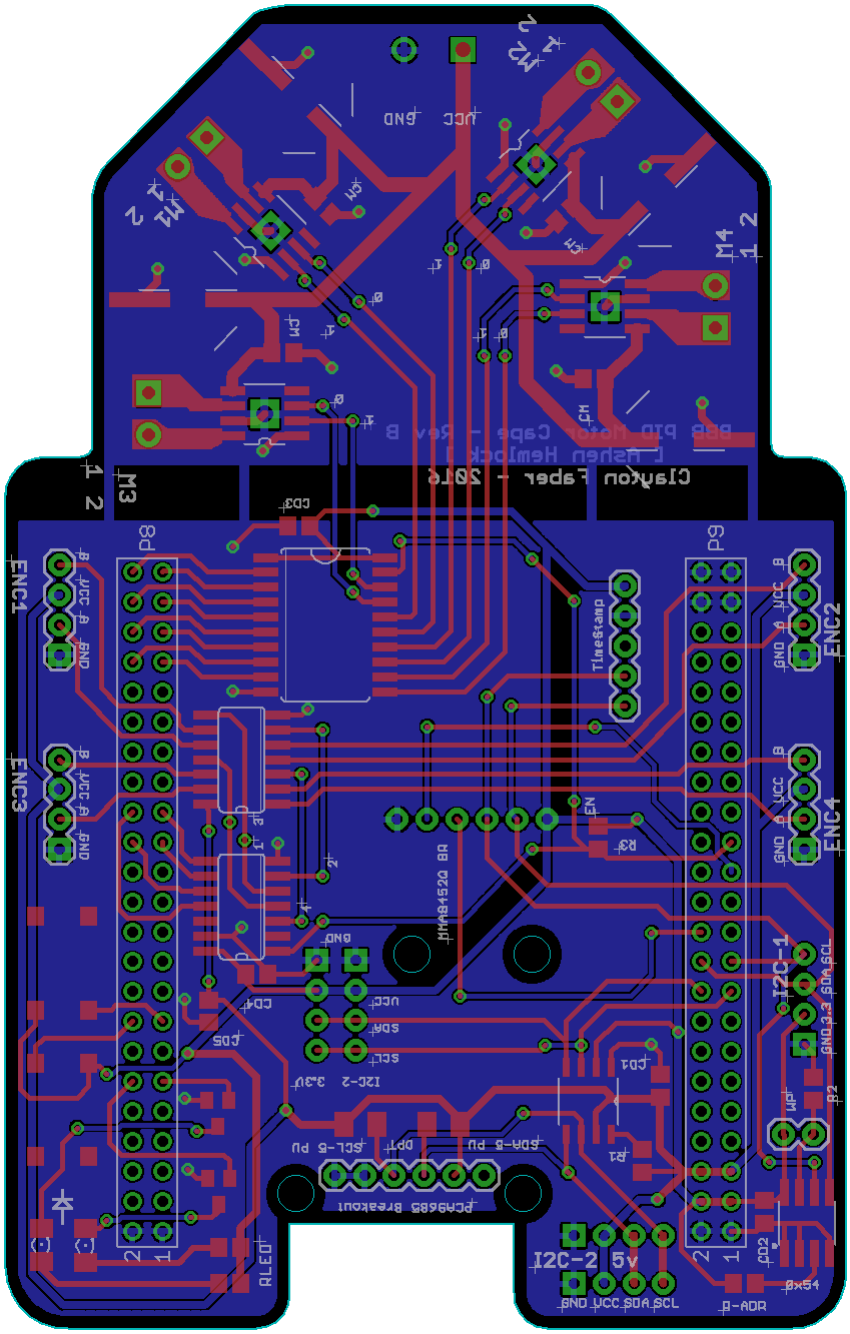

Figure 2.4: PCB Layout of SIUE Robot Cape

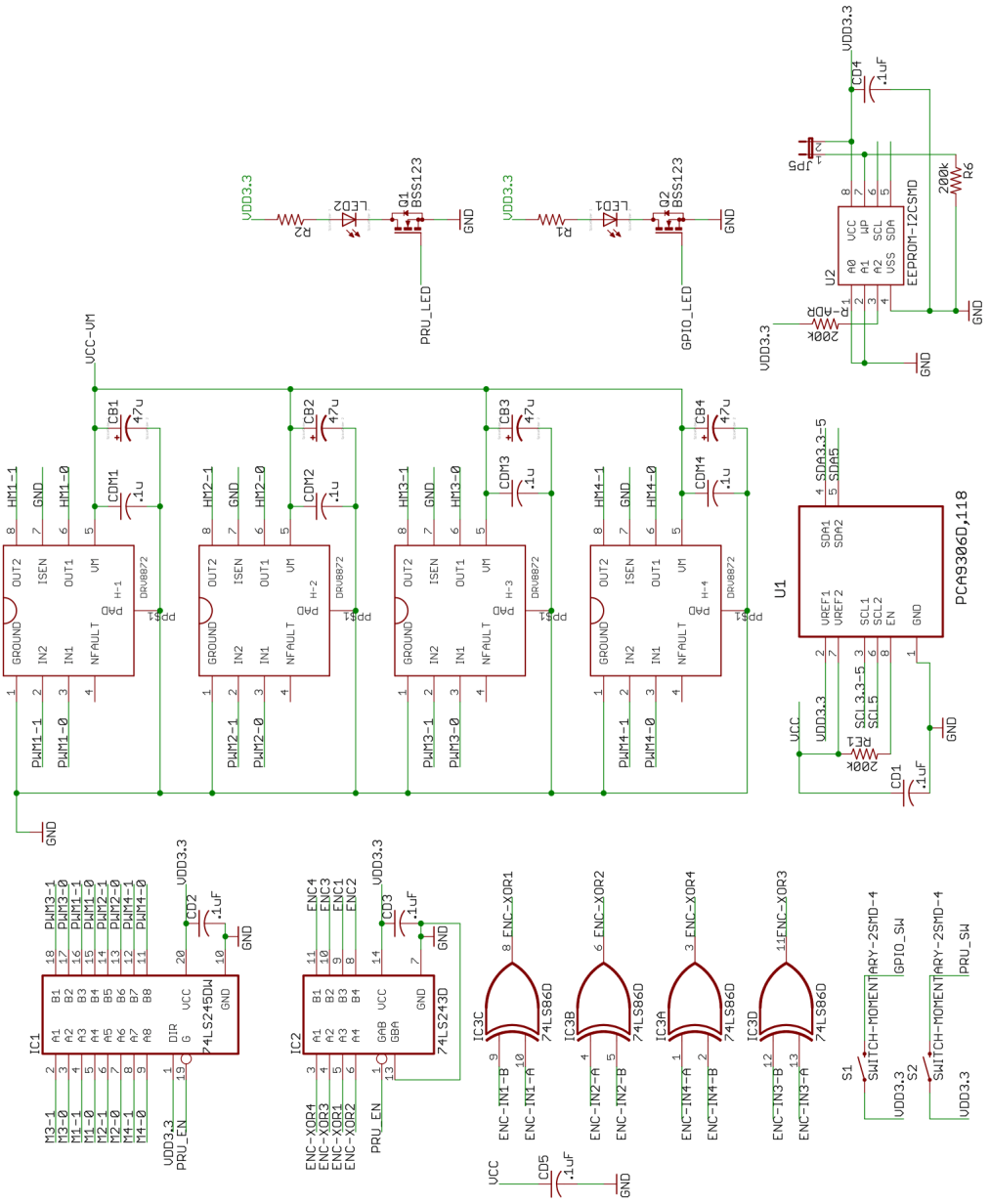

Figure 2.5: Schematic of SIUE Robot Cape (1 of 2)

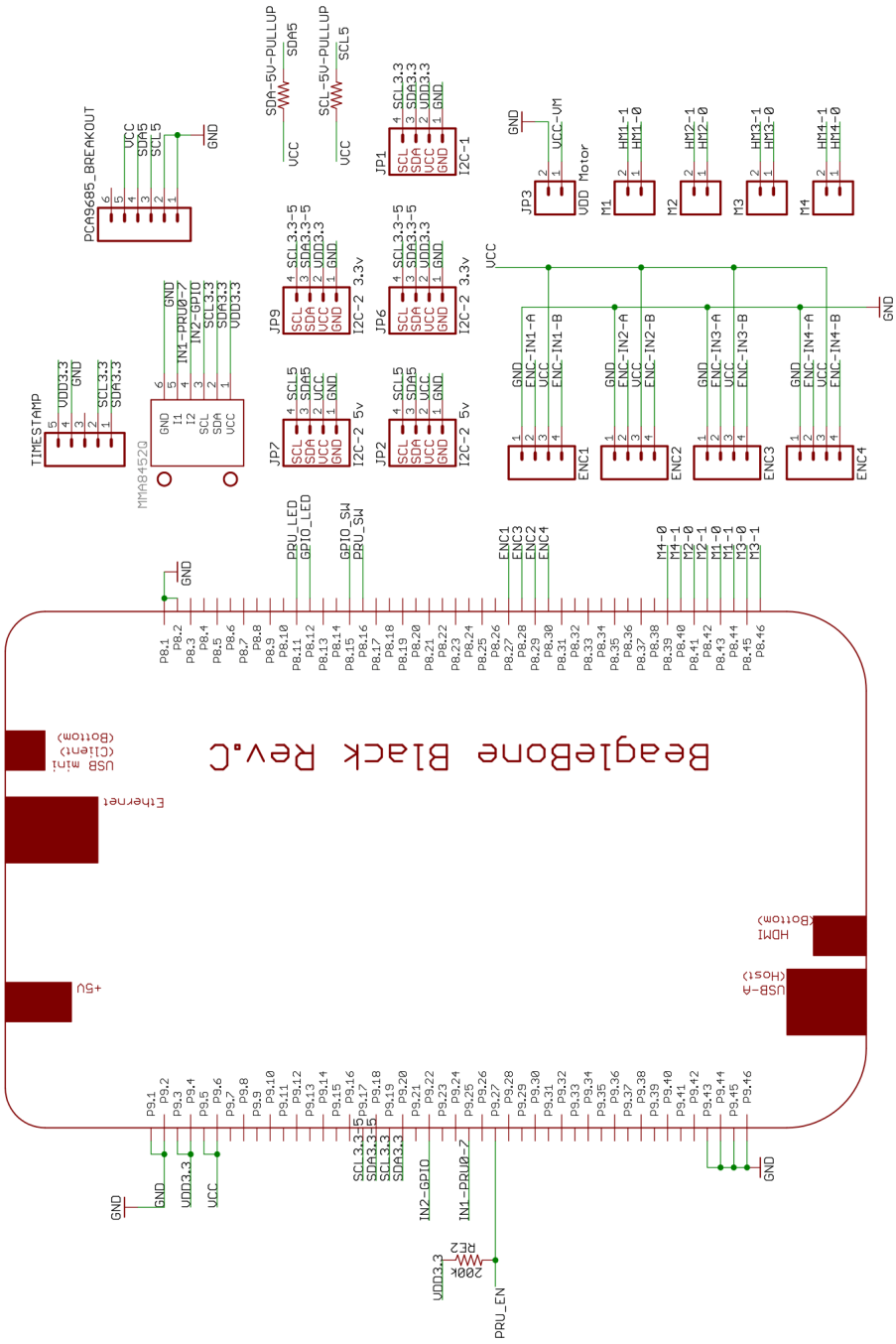

Figure 2.6: Schematic of SIUE Robot Cape (2 of 2)

CHAPTER 3

PRU SOFTWARE

3.1 Software Design Objectives

The program controlling a robot is as (and perhaps even more) important than the robot frame itself, and while the cape is an important piece of hardware, it cannot drive the motors without some type of control program. In this chapter a general overview, software design objectives, algorithms, and software routines used to control a pair of DC motors will be discussed.

As already described, the BeagleBone Black possesses **two** 32-bit RISC cores, called PRUs (Programmable Real-time Units), each with their own data and instruction memories and also with direct access to the P8 and P9 headers. Moreover, each PRU has access to a 32-element register file with just two of the registers *reserved* for GPIO and system status use (*i.e.* there are 30 for general use). Each PRU can access 8 KB of program memory and 8 KB of data memory. They also have access to a 12 KB block of shared memory which can be used by both PRUs as well as by the ARM core.

The two processors can run programs independent of each other and independent of the ARM. All processing of motor control signals will be directly handled by the two PRUs in a master-slave relationship. The ARM requests movement by setting values in shared memory and can then monitor a specific location in the shared memory to determine when the command has completed. All three processors (PRU10, PRU 1, ARM) use the shared memory as a way to communicate through the use of command and data words.

A DC motor controller needs a way to handle encoder feedback, perform calculations based on that feedback, and then in turn control the PWM duty cycle for the H-bridges which directly drive the motors. For a two-wheeled robot, we will see that this will entail implementing 3 PID loops.

Each PRU runs at a clock speed of 200 MHz (*i.e.* a 5 nanosecond instruction execute time) and while this is quite fast it would not guarantee enough time to do both calculations and handle the real-time applications of the PWM and encoder feedback for two motors. To solve this problem, it is advantageous to break up the workload between the two PRUs: one (PRU 0) doing calculations while the other (PRU 1) provides a direct interface to the SIUE Robot Cape hardware. The unit directly interacting with the hardware needs use its clock cycles efficiently and is therefore best programmed in assembler.

There is 12 KB of shared memory that can be used by all three processors and sits on the PRUs bus for their direct access without the need to access ARM system RAM which would require further processor time. Design objectives for the ARM system will be covered in detail in the following chapter, but it is important to note that its role in this design is to issue commands to PRU 0, place configuration data in memory, and start the process of motor control. PRU 0 in turn communicates with PRU 1 which controls the hardware on the cape.

3.2 PID Algorithm Overview

While PID algorithms have been discussed at length in other publications, a brief overview will be presented. A PID controller is used to maintain a certain setpoint through calculations based on a feedback signal. PID is an acronym of three separate control algorithms: proportional, integral, and derivative using a feedback signal as an input. The three partial terms are calculated and then their sum determines the output.

The three PID loops have their own gain values which are used to tune the loop. In this application we wish to control two DC motors and as such there are three PID loops in use. Two loops are set up to control each individual motor's velocity and a third differentially controls the two motors' velocity setpoints. A system block diagram depicting the two slave loops and the master loop is shown in Figure 3.1.

This scheme helps get the most out of a differential drive machine and helps compensate

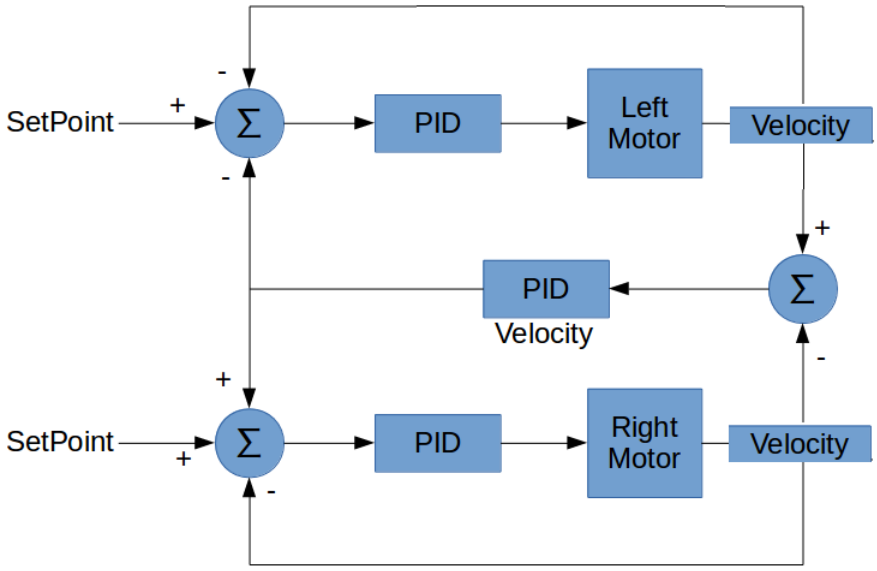

Figure 3.1: Block Diagram of PID Control Used in Differential Drive Design

for disparities between DC motors [Yeong Chin Koo, 2015]. By having a master control loop that differentially drives the wheels, the robot can account for errors during travel. For example, if one wheel happens to encounter a carpet causing it to decrease in velocity the master PID control loop would cause the wheel on the carpet to spin at a faster rate in order to overcome the additional resistance presented by carpet and at the same time the wheel not on the carpet would slow down at a proportional rate to attempt to keep the robot moving in a straight line.

This also comes in handy when dealing with the step response of DC motors. DC motors are not created equal and as a consequence their inductances do not always match. Because of this, one motor may react quicker than the other when starting from a dead stop and cause the robot to become misaligned. With the third PID loop in place, the PID loop will observe that one wheel is spinning faster than the other and in turn will

attempt to solve the problem.

The input of the PID controllers driving the two DC motors is the error (*i.e.* e(n)) between the setpoint value and the encoder count observed during a single sample period. The setpoint value is the velocity of the motor or more specifically the number of encoder tics that should occur during a single sample period for the corresponding velocity. The third PID loop's input is the difference in wheel velocities. The setpoint for the third loop in the current implementation is zero so as to ensure straight-line movement but this need not be the case. One could drive the differential velocity to something other than zero which would cause the robot to follow an arc. In this way, arbitrary paths could be followed.

The "differential" form of the PID algorithm (Equation 3.1) is used in this work [AdamBots, 2008]. The next PWM output is computed by taking the current output and adding a "delta" which is calculated using the PID constants. The "past" error is e(n-1) and the "past-past" error is e(n-2). The PID constants are K_p, K_i, and K_d. This results in a much simpler algorithm, only requiring a couple lines of code with just a few fixed-point math operations.

$$\begin{aligned} pwm(n) &= pwm(n-1) + \Delta(n) \\ \Delta(n) &= \Delta_p(n) + \Delta_i(n) + \Delta_d(n) \\ \Delta_p(n) &= K_p \cdot e(n) \\ \Delta_i(n) &= K_i \cdot [e(n) - e(n-1)] \\ \Delta_d(n) &= K_d \cdot [e(n) - e(n-1) + 2 \cdot e(n-2)] \end{aligned} \tag{3.1}$$

Fixed-point arithmetic is used rather than floating-point due to the large overhead associated with floating-point calculations and the amount of program space it takes up. Values are calculated in a Q6 fixed-point format (6 fractional bits). By using a Q value of 6 we ensure a very high dynamic range (150 dB) but yet retain sufficient fractional

information to keep the errors within reasonable bounds.

The conversion between fixed- and floating-point representation occur within the ARM program which places the fixed-point values in shared memory for the PID controller to use. It should be noted that the Q-value can be easily changed since it is defined in the fixed-point include file.

3.3 Shared Memory Map

Each program relies on shared memory for accessing values set by other processors. In the case of PRU 1, PWM values are read and encoder values are written to shared memory. PRU 0 reads constants for us in calculations and sets status flags in the movement process for the ARM core to read later. All three devices have to agree on how the memory is structured with both single word values and structures being maintained in it. This task is achieved through the mem.h file where, along with other constants and aliases, a shared memory structure is defined.

The mem.h file exists as one file in the project with symbolic links in both the PRU and ARM compilation directories. The ARM and PRU 0 coompilers think they have their own copy of the file but in reality there is only one copy which makes editing simple because a change in one file causes the change to show up for everyone with a link to it.

The mem.h file has a structure definition called shared_memory_t. Both the ARM and PRU 0 code define a pointer to this shared memory structure and its value is set equal to the start of shared memory. When a structure is created in C, memory is allocated based on the data length of the structure. When accessing a certain value in a structure using a pointer, the C compiler takes care of the offset for the user.

When the ARM program sets a value in the shared memory structure, PRU 0 reads it in the same way the ARM core wrote it because they share the definition of the structure. In Figure **??** the listing of the shared memory structure is defined on the left which contains all the values and structures used by all three processors. Because it does not

have access to the pointer and structure declarations, the memory locations PRU 1 uses are placed at the beginning of the structure. This makes determining the offset easier. After PRU 1 values are defined, other structures and flags are created to help organize data. The structures available in shared memory are shown in Figure 3.2.

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Our shared memory structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int32_t     pwm[NUM_MOTORS] ;      // shared mem byte os of 0
    int32_t     enc[NUM_MOTORS] ;      // os of 16
    int32_t     delay ;                // os of 32
    int32_t     state ;                // os of 36
    int32_t     PWMclkCnt ;            // os of 40
    int32_t     PWMres ;               // os of 44
    int32_t     exitFlag ;             // exit when true
    int32_t     interruptCounter ;     // sample counter
    int32_t     motorType ;            // DC or stepper
    int32_t     motorENA[NUM_MOTORS] ; // Motor enables
    int32_t     scr ;                  // scratchpad register
    int32_t     wheelDiam ;            // diameter in inches (Q)
    int32_t     ticsPerInch;           // encoder tics per inch (Q)
    int32_t     enc_data[BUF_LEN] ;    // Buffer of encoder data
    command_t   command ;              // Motor command structure
    DCmotor_t   motor[NUM_MOTORS] ;    // DC motor structure
    PID_t       setpointPID ;
}   shared_memory_t ;
```

Figure 3.2: Shared-Memory Map

3.4 PRU 1: Hardware Interface

PRU 1 was chosen to directly interface to the SIUE Cape hardware due to its access to input/output pins on the P8 header. We require 8 outputs for the H-Bridge motors and 4 inputs for the encoder feedback. Along with these real-time signals, PRU 1 must also act as the timekeeper letting PRU 0 know that the sample period is complete and that it is time to calculate new PWM outputs. As a result of these real-time constraints, PRU 1 is programmed in assembly code in order to keep track of execution time which, per instruction except memory access, is 5 nanoseconds. Fortunately, in order to make things easier for programmers the assembler allows for the use of "macros" and "defines" in the code through the use of header files.

In the header file for PRU 1, pru1.h, there are a list of defines and macros which makes the program, pru1.p, easier to understand. Because the PRU system has a large number (32) of registers it was best to use defines to create an alias for each register and give it a specific purpose. These assignments are given in Figure 3.3.

The motor control bits are defined first which are part of the output register, r30, and have a .t# following them to define the bit number. These are not in numerical order in order to make the layout of the board easier by following the numbering of the H-Bridges. As an example, r30.t1 corresponds to M3-1 which is connected to the output pin P8.45 which is in the very last row of P8 connector which is connected to the corresponding H-bridge on the top of the board as pictured in Figure 2.1.

Moving down the list, a *nop* alias is defined which is used in NOP operations as a means of effectively "stalling" the processor. Next, the registers that will hold the encoder and PWM values are defined. The encOLD, encNEW, and encEDGE registers play a role in reading encoder ticks by saving values from the input register, r31, which will be described in more detail shortly.

Next, a set of 6 GPIO registers are listed which can be used to access the Linux space GPIO pins that are connected to the motor cape. These can be used for debugging purposes. The next three are registers that hold control values. The register pwmResReg holds the maximum value corresponding to the resolution of the PWM signal. For example, an 8-bit PWM signal would mean that this register holds an unsigned value of 255. The stateReg acts as a way for the PRU 1 and PRU 0 to communicate without solely relying on interrupts as well as the control of the motor spin direction.

Certain bits are designated as flags and are used to let PRU 1 know what mode it is in and if it needs to stop or start. Similarly a section of the register is used to designate how the motor will spin and are also is used to start the PWM outputs. This is achieved by loading the bottom byte of stateReg into the bottom byte of r30, the output register,

effectively starting the on timer for the PWM signal. The delayClkCnt register holds the value corresponding to the number of PWM cycles making up a PID sample period.

The i and j registers are used for counting the iterations of the two loops: PWM and sample period. Lastly, aliases are given for the different registers that hold the memory address values which are used as pointers for the different locations in memory. Although there are more defines listed in the appendix, these give the general overview of what kinds of values are in use. These defines are used in the macros also defined in pru1.h, a listing of important macros are available in tables 3.1 and 3.2 with a short description of their usage and function. Together these are used in pru1.p and make up the main body of the program.

The pru1.p file contains the source code for PRU 1 which is assembled to a .bin file and loaded into the PRU 1 program memory. A general flowchart of pru1.p is given in Figure 3.4. The first task of PRU 1 is general setup which begins with the line label START. This involves clearing the registers, setting up pointers, and reading in constants from the shared memory. In the setup procedure, a hard brake is set so that the motors are guaranteed not to move while waiting for a run flag from PRU 0.

After setup there is a check to see if a run flag is set by PRU 0. This is to prevent PRU 1 from starting operation while PRU 0 is still taking time to setup. The program will loop and pull the stateReg value from memory until a run flag is seen by PRU 1. After the run flag is seen, the program enters the main loop whose first task is to send an interrupt to PRU0 to let it know that a new PID sample period has started.

Following that, the state register is updated and another check is made to see if the run flag is still set and if it is the encoder count registers are cleared. The i register is loaded with the clkCntReg value signaling the start of the i loop also known as the PID sample loop. The i loop starts by reading in the PWM values set by PRU 0. Immediately following that the motor signals are turned on using the bottom byte of statReg. The j

```
#define         M1_0            r30.t2
#define         M1_1            r30.t3
#define         M2_0            r30.t4
#define         M2_1            r30.t5
#define         M3_0            r30.t0
#define         M3_1            r30.t1
#define         M4_0            r30.t6
#define         M4_1            r30.t7

#define         nopReg          r0.b0

#define         enc1            r1
#define         enc2            r2
#define         enc3            r3
#define         enc4            r4
#define         pwm1            r5
#define         pwm2            r6
#define         pwm3            r7
#define         pwm4            r8

#define         encNEW          r9
#define         encOLD          r10
#define         encEDGE         r11

#define         GPIO_LED_STATE  r12
#define         GPIO_BUTTON     r13
#define         GPIO_LED        r14
#define         read_gpio1      r15
#define         set_gpio1       r16
#define         clr_gpio1       r17

#define         pwmResReg       r18
#define         stateReg        r19
#define         clkCntReg       r20

#define         i               r21
#define         j               r22

#define         sharedMem       r25
#define         pru1Mem         r26
```

Figure 3.3: PRU 1 Register Aliases

Macros	Usage and Description
get_state	Usage: get_state Update the status register value in pru1. b0 contains wheel control information, while b1-2 contain flag bits. Can be updated in shared memory from PRU0 or ARM
read_clk_cnt	Usage: read_clk_cnt Reads in the number of PWM cycles to run before interrupting PRU0.
read_pwm_res	Usage: read_pwm_res Reads in the pwm maximum count from sharedMemory. Either 255 (8 bit), 1023 (10 bit), or 4095 (12 bit)
read_pwm_values	Usage: read_pwm_values Reads in PWM high time from Shared Memory
pwm_timer	Usage: pwm_timer M#_ctrl, pwm#, M#_1, M#_0, NEXT_LINE This decrements the PWM register given the timer register and will stop the pwm signal if necessary.
pwm_start	Usage: pwm_start Uses the state register to start the PWM singal using values stored in statReg.b0

Table 3.1: PRU 1 Macros (1 of 2)

Macros	Usage and Description
check_encoder_edges	Usage: check_encoder_edges Saves the old encoder values, reads the new ones, and XORs the two to see if there is an edge.
enc_cnt	Usage: enc_cnt enc#, enc#_bit Reads the encoder tics from encEDGE and increments the register if need be.
zero_encoder_regs	Usage: zero_encoder_regs Clears the enc1, enc2, enc3, and enc4 registers.
store_encoder_values	Usage: store_encoder_values Stores the current encoder values to shared Shared Memory
brake	Usage: brake Stops the pwm by setting outputs to zero or to one depending on the brake bit.
NO_OP	Usage: NO_OP No operation (1-Cycle)
inc	Usage: inc r# Increments register
dec	Usage: dec r# Decrements register
send_ARM_interrupt	Usage: send_ARM_interrupt Sends an ARM interrupt Only to be used when halting.

Table 3.2: PRU 1 Macros (2 of 2)

loop is entered after the j register is set equal to the PWM resolution, and the process of checking encoder ticks and maintaining the PWM loop starts.

Each iteration of the j loop takes a snapshot of the encoder edges using the check_encoder_edges macro which saves the previous value and performs an XOR operation of the current value to look for positive edges. If edges are present the corresponding encoder count value is incremented. Next, the PWM timer is checked and if necessary motor signals are brought low. Leaving the j loop the j value is decremented and if it is equal to zero continues to the end of the i loop which in a similar fashion decrements the i value and exits if equal to zero.

When exiting the i loop, the encoder values are stored into shared memory before branching back to main. Finally, as an alternate scenario when returning to main if the run flag is de-asserted the program branches to the STOP portion which enables the brake and checks for a halt flag. If the halt flag is set, an interrupt is sent to the ARM and PRU 1 halts. If not, it waits for a run flag back in the WAIT_FOR_RUN_FLAG line. The resulting list file from pru1.p compilation shows that the program size ends up being around 424 bytes in size (taking up a small fraction of the 8 KB instruction memory).

3.5 PRU 0: PID Controller

PRU 0 is where the calculation of the PID loop is performed and as such only has to perform calculations every sample period. This may sound like it would be a daunting task but the PID sample period is typically in the 10s of milliseconds. Given that one millisecond equates to 200,000 clock cycles on the PRU, a modest sample period of even 5 milliseconds would result in plenty of time to set and calculate values.

Because the sampling periods are long (milli-seconds) and the PID code would be a bit daunting to program in assembly, the PRU 0 is coded in C using the PRU Code Generation Tools from Texas Instruments [Molloy, 2014]. Not only does this remove the headache of complex assembly code, but also gives the programmer access to functions,

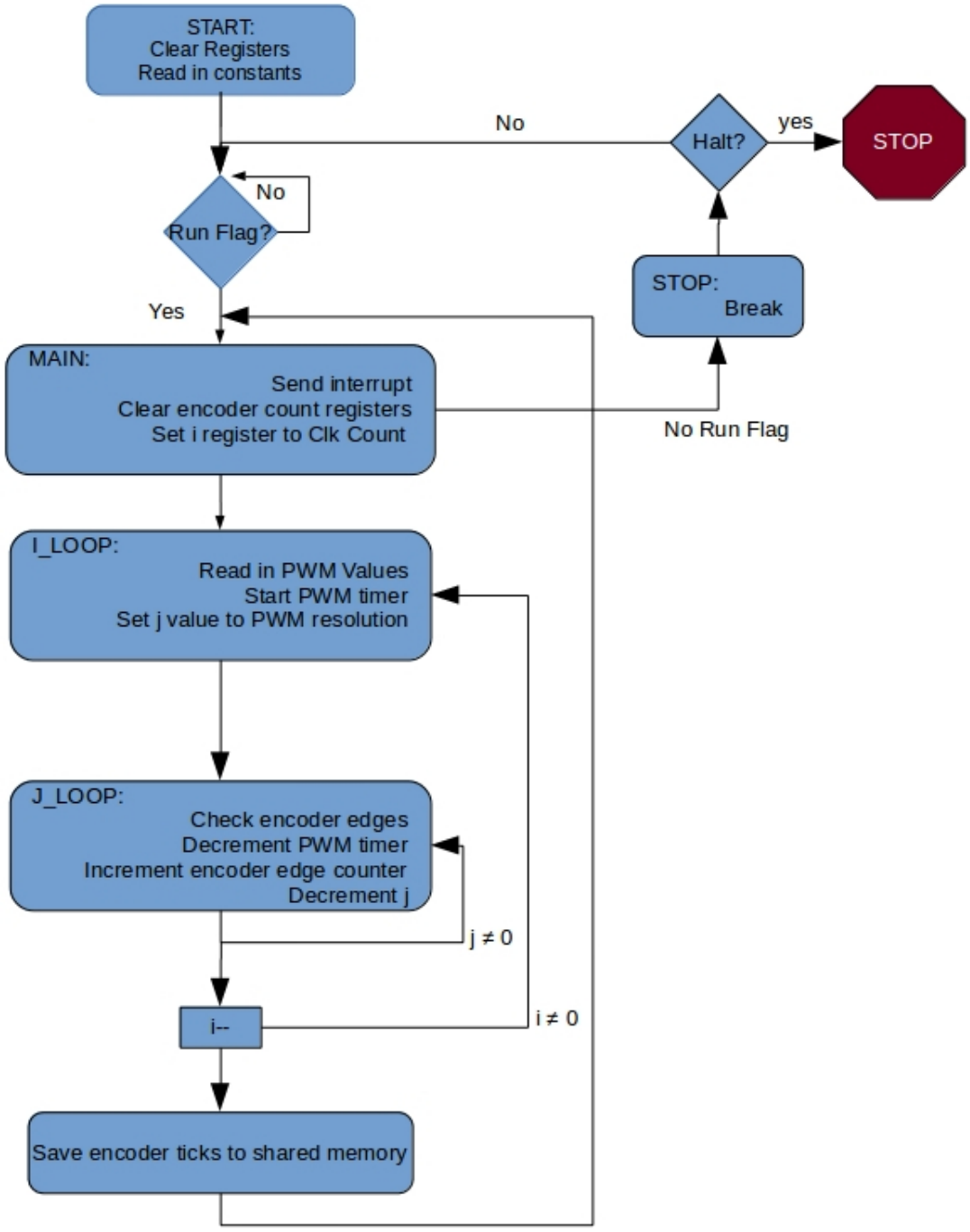

Figure 3.4: Flowchart for PRU 1 Code

Functions	Description
initGPIO(void)	Enables GPIO functionality
GPIO1pin(int pin, int value)	Controls a GPIO bank 1 pin based on the pin number and value
GPIO3pin(int pin, int value)	Controls a GPIO bank 3 pin based on the pin number and value
blinkLED(void)	Toggles the GPIO LED
enableBuffers(void)	enables the 8 and 4 bit buffers
disableBuffers(void)	disables the 8 and 4 bit buffers
initPRU(void)	clears the PRU1 interrupt channel and call initGPIO
waitForInterrupt(void)	Waits for an interrupt from PRU1 in a while loop. Before leaving it toggles the PRU LED on the board.
killTime(void)	Delays the PRU by roughly 50ms by counting up to 1,000,000.

Table 3.3: PRU 0 Functions

structures, and memory pointers. The PRU 0 code is split up between three C files: pru0Lib.c with functions for general PRU0 operation, motorLib.c contains functions for motor control like the PID loop, and pru0.c which holds the main entry point for the program. Along with the C files are header files containing masks and aliases for each library as well as a mem.h file describing the shared memory structure.

Table 3.3 contains a listing of the functions and aliases with descriptions available in pru0Lib.c, pru0Lib.h, pru0.c, and pru0.h. Most of these are concerned with the use of Linux GPIOs which can be used by PRU0 if desired. One important note is that PRU

Functions	Descriptions
FADD(op1, op2)	adds two fixed point numbers.
FSUB(op1, op2)	subtracts two fixed point numbers.
FMUL(op1, op2, q)	Mutiplies two fixed point numbers together whose output has a fixed point value of Q.
FCONV(op1, q1, q2)	Converts op1 from a q1 fixed point to a q2 fixed point.
haltPRU(void)	Sets the halt flag for PRU1.
hardBrake(void)	Sets PRU1 up to use the hardbrake.
cost(void)	Sets PRU1 up to use the soft brake for stopping.
move(void)	Called when there is a request to move. Main loop of movement control.

Table 3.4: *motorLib* Function and Fix Macros (1 of 2)

0 controls the Linux GPIO that triggers the on/off status of the buffer chips previously described in Chapter 2 through the two functions enableBuffers() and disableBuffers(). This way the PRU0 acts as the absolute master of the H-bridges and will not enable the motor circuits until entirely set up and ready to begin movement. Most of the PID code and functionality lies within the motorLib.c, motorLib.h and fix.h files and are overviewed in tables 3.4 and 3.5.

Functions that are part of "motorLib" are accessed through the state machine in the main program which is shown in the flowchart depicted in Figure 3.5. The main program starts by by calling initPRU(), turning on the buffers, and setting the "mem" pointer equal to the address location of shared memory. Then the main state machine is entered which waits on an exit flag in shared memory and switches state based on the value of command.status in shared memory.

When PRU 0 sees a START status it calls the doCommand function from motorLib.c,

PID(dc_motor *motor int32_t enc)	Called inside move to calculate PID output for each motor, returns the calculated value in a non-fix format
adjustSetpoint(int32_t rightVel, int32_t)	Third PID loop that compares the velocities of the two motor and adjusts their setpoints to drive their difference to zero
doCommand(command_code)	State machine that calls functions like move() or haltPRU depending on the command code.
createState(void)	Sets up the state value for PRU1 in shared memory. Sets bits controlling the state of the motors and how they will rotate

Table 3.5: *motorLib* Function and Fix Macros (2 of 2)

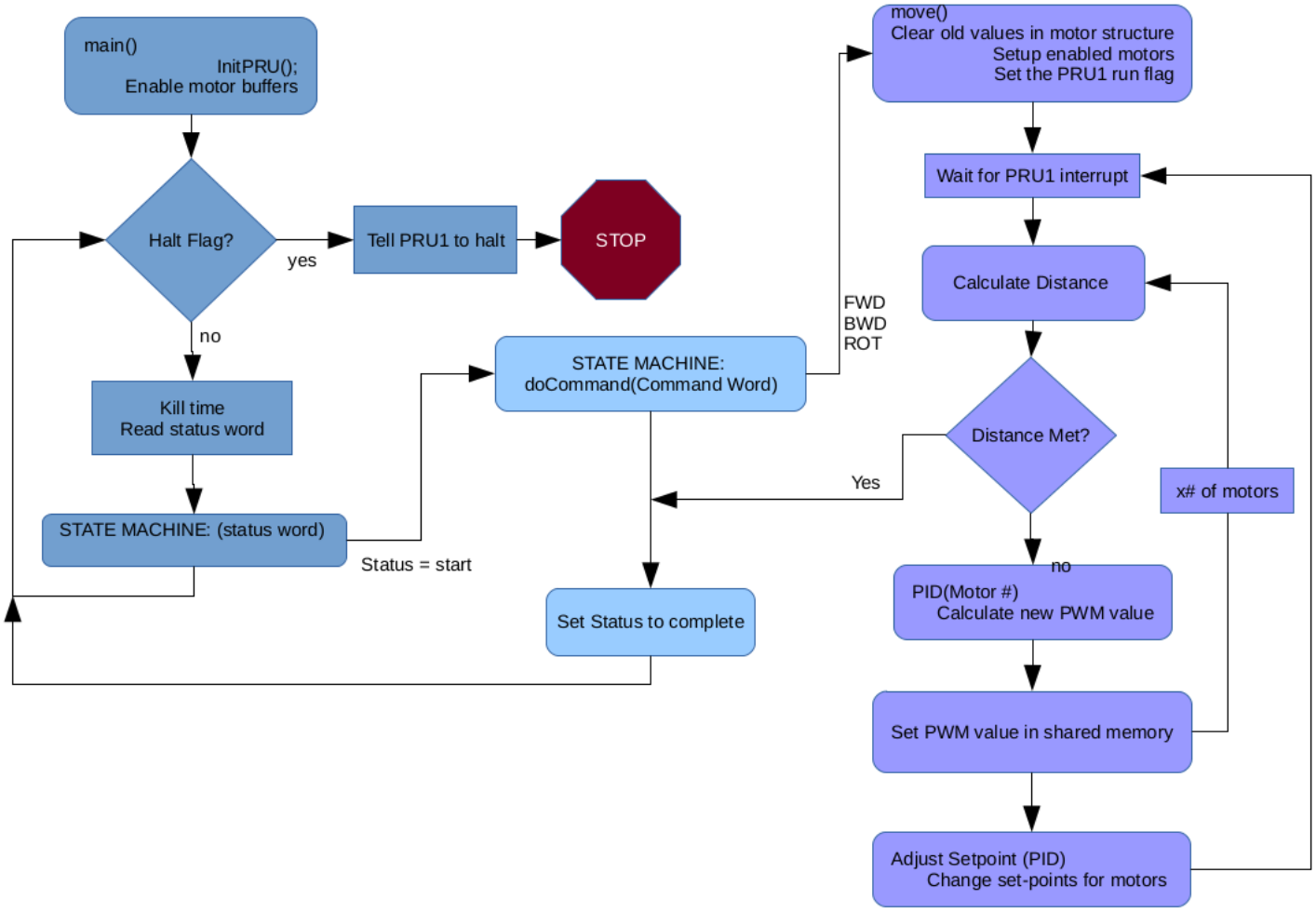

Figure 3.5: Code Flowchart of PRU 0 for a Typical Move Command

passing the command code also set in shared memory. The doCommand() function is another state machine that will call a function based on the command code sent to it and then set the status to COMPLETED when finished. When following an actual movement command code, like FWD (forward), the move function is called.

The move function uses values set in shared memory by the ARM core to begin the process of movement. the first order of business is to clear the errors, distance and PWM value for each motor. The routine createState() is then called to set up each motor for the direction that it needs to spin (clockwise or counterclockwise). After preparations are complete, the run flag is set which causes PRU 1 to start operations and begin the PWM process. From there PRU 0 begins the process of waiting for an interrupt from PRU 1, marking the start of new sample period. When an interrupt is received, PRU 0 begins working on the PID loops for the individual motors.

First, the total distance traveled by one motor is calculated followed by a check to see if the current set-point need to be increased. Then, if the target distance is not yet reached, work begins on the PID loop for one motor. The PID calculation first starts by saving the previous set-point errors and calculating the new set-point error. From there the delta P, I, and D values are calculated using gain parameters set by the ARM. Before returning the output value it is first checked to see if the output is within the maximum rate of change allowed via the maximum delta cap. Then to make sure that it is within range of the PWM counter another check is ran and capped if needed.

Returning from the PID calculation, the PWM value is stored in a temporary buffer to be used at the end of the calculation loop. Once each output of the PID controller is calculated the adjustSetpoint() function is called to try and drive the difference in velocities of two motors to zero. This is done in another PID loop and has the same checks on maximum rate of change as the previous one. The output of this PID loop is added to one wheel's set-point and subtracted from the other and which is mirrored by

the initial subtracting to find the difference in velocity which is the input into this PID loop. Until the target distance is reached the loop will continue to wait for interrupts from PRU1 recalculating the PID outputs.

Finally, once the target distance is reached, the run flag for PRU 1 is de-asserted by PR U0 and the program can leave the move function. Returning from the move function back into the doCommand state machine, the status is set to completed and then returns to main. From there, the state machine waits for either a new start status or exit flag. When an exit flag is present a doCommand is called with the parameter HALT_PRU which will cause PRU 1 to cease operation. Lastly, the buffers are disabled, the LED is turned off, and an interrupt is sent to the ARM core signaling the end of operation.

CHAPTER 4

DEMONSTRATION ROBOT

4.1 Robot Construction

In this chapter, a demonstration robot will be described in an effort to showcase the power of the SIUE Robot Cape. The robot is simple in design and only has the ability to move around in its environment. It has no additional sensors for interacting with the environment. The goal was to merely to demonstrate the reliable manner in which a robot, using the SIUE Robot cape (and related software), can successfully navigate a typical IEEE robotics arena.

Along with the robot, the Linux code library will be showcased. The user libraires which were developed contain routines for movement along with other utilities such as functionality for the I2C devices available on the board. A graphical user interface (GUI) will also be described which was created for ease of use which allow the user to make changes to system parameters. Finally, we conclude the chapter with a description of the program used to evaluate the quality of the work described in this thesis, and we present the results of the standard tests for robot movement.

In order to test the board, the demo robot was constructed using only the BeagleBone Black, the cape, two DC motors with encoders, and batteries. The only functionality that this robot has is the ability to drive forwards, backwards, or rotate. The robot also has a free spinning caster wheel for stability when moving.

The frame of the robot is created using *Makeblock*® which is a robotics construction kit aimed at customization and and re-usability. *Makeblock* is the next generation of construction platforms for hobbyist and student oriented robots. *Makeblock* offers strong mechanical parts, mostly made of hardened aluminum.*Makeblock* is the perfect choice to build all kind of robots. Whatever idea a student group might have in mind, it can

be implemented using *Makeblock* parts. The students would simply need to connect the parts together, and the frame of the robot will be constructed in no time at all.

Features of *Makeblock* include:

- high strength aluminum extrusion parts,
- smart threaded slot,
- high performance timing belt support,
- multi-functional adjustable components,
- and easy to use, fast to construct.

Parts available include:

- timing belts,
- stepper motors,
- DC motors with wheel encoders,
- linear and rotational bearings,
- caster wheels,
- sliders, axis and threaded shafts,
- and other standard industrial parts.

A complete robot can be built from these parts, and *Makeblock* should be considered for use in future competitions. Along with the SIUE Robot Cape, *MakeBlock* offer a total solution to teams competing in the annual IEEE robotics competition. The demonstration robot discussed in the remainder of the chapter is pictured in Figure 4.1 and Figure 4.2.

The robot pictured only required a few hours to assemble, is small (approximately 8 inches X 8 inches), light weight, and mechanically sound.

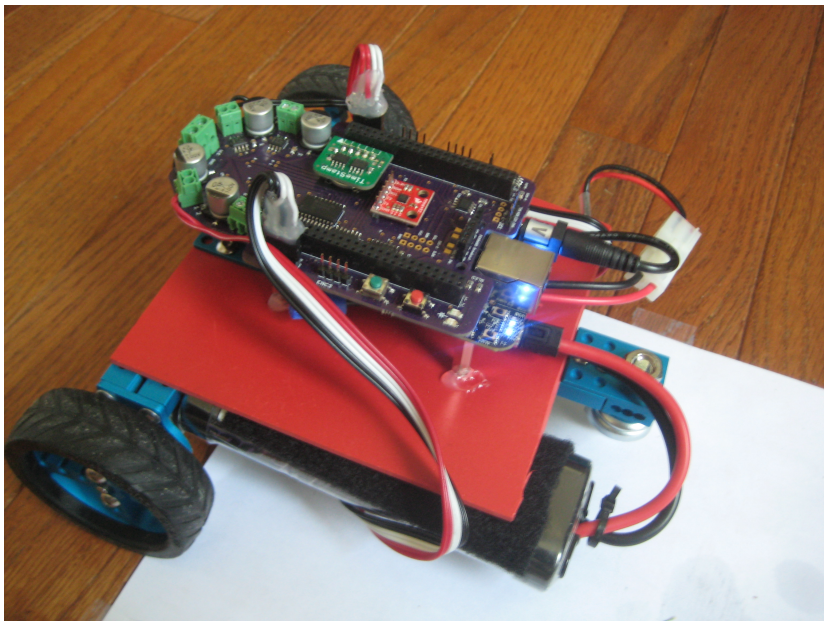

Figure 4.1: Demonstration Robot (top view)

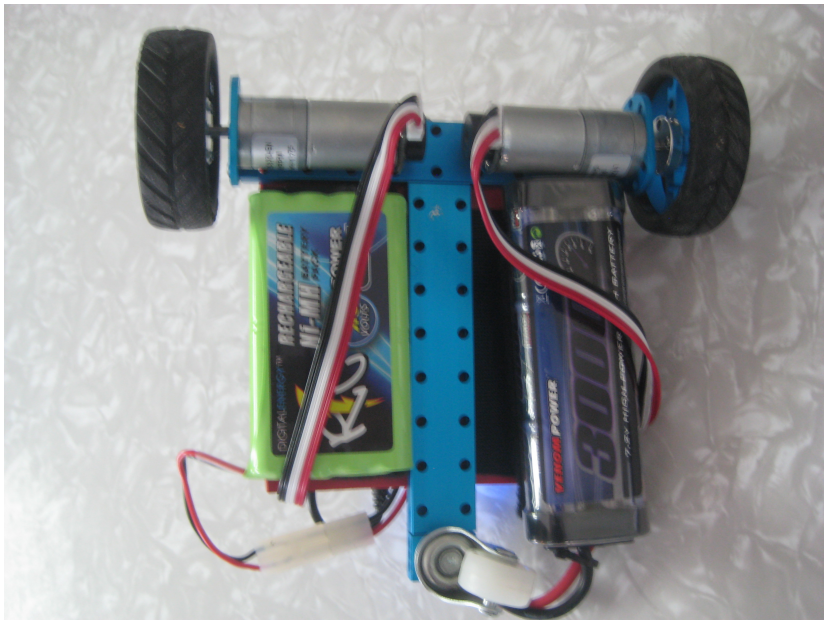

Figure 4.2: Demonstration Robot (bottom view)

In the bottom view of the robot two NiMH batteries can be seen. The 9.6 Volt battery

(1600 mA-H) powers the DC motors while the 7.2 Volt battery (3000 mA-H) is converted by a buck converter circuit (located under the BeagleBone Black board and hidden from view) to 5 Volts which is used to power the Black and the SIUE Robot Cape. While not absolutely necessary, the use of two batteries helps ensure that high-frequency noise generated by the PWM circuits does not inadvertently reboot the BeagleBone Black. This has been a common problem encountered by SIUE robotic teams over the years.

4.2 PRU Library Routines

The Linux code has a collection of functions that are used to access and configure the PRUs using the PRUSSDRV user space library which contains the functionality for basic PRU control, memory mapping and interrupt handling [Texas-Instruments, 2015]. Through the PRUSSDRV libary, a user is able to load the PRU with the .bin programing file. The creation of the .bin file needs to be done before executing the main program code on the Linux system.

One important note is that before the user can call PRUSSDRV functions, the uio_pruss module needs to be loaded, luckily this is handled by the operating system when a cape (for example, the SIUE Robot Cape) declares use of the PRUs is loaded. The use of the PRUSSDRV is illustrated in Figure 4.3 from Derek Molloy's book: Exploring BeagleBone. The figure describes the process of programming the PRUs from the viewpoint of the PRU and the Linux operating system.

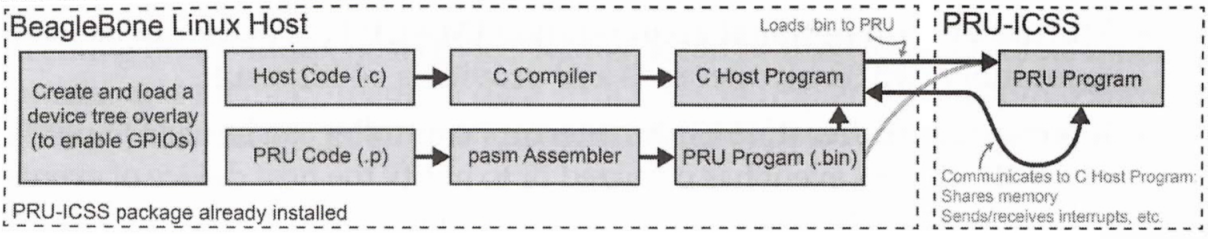

Figure 4.3: Process of Loading and Executing a Program Using PRUSSDRV and Linux

In attempt to simplify this process of calling driver functions, wrappers were created

to simplify calls and streamline the process. These wrappers were bundled together into *PRUlib*. These functions automatically take care of the proper PRU setup for the SIUE Robot Cape by loading in the correct programming files and setting up the correct interrupt channels. The functionality is split between three functions: PRUinit(), PRUstart(), and PRUstop(). PRUinit() handles the setup of the PRUs by calling the initialize function from PRUSSdrv. It also opens the interrupt channels for both PRUs and sets up the shared memory pointer used by the ARM to write values into shared memory.

It is important to note that PRUinit() relies on global variables declared in the main program which are accessed with extern declarations. PRUstart() handles the loading of the PRU programming files located in the same directory as the executable program. Finally, PRUstop() calls the disable functions for the PRU, halting the PRUs. A listing and description of these functions can be found in Table 4.1. When using these functions in a program, it is important that PRUinit() and PRUstart() are called before issuing a movement request.

Functions	Description
PRUinit(void)	initializes the PRUs, opens the interrupt channels, and sets up the shared memory pointer (Depends on a global shared_memory_t pointer named shared_memory)
PRUstart(void)	Load and execute the binaries for PRUs
PRUstop(void)	Disables the PRU and closes the memory mapings for PRUSSdrv

Table 4.1: *PRUlib* Functions

4.3 Beaglebone Black Library Routines

The BeagleBone Black library or *bbbLib* is a collection of routines created by Gavin Strunk. The library simplifies the job of controlling the various hardware available on the BeagleBone such as the the I2C buses and UART terminals. The use of the library in this thesis is the setup and control of the GPIO pins in the /class/sys/GPIO folder and control of the I2C buses.

This library can be used by users to create their own drivers for devices that use communication buses such as SPI, UART, or I2C. The ability to access the PWM generators and ADC functionality of the BeagleBone Black is also included in this library but, like any on-board peripheral,they must be set up through the device tree first. Table 4.2 and Table 4.3 describe the *bbbLib* functions that are used by the SIUE Robot Cape library which will be described in the next section of this thesis.

4.4 Robot Library Routines

A series of routines were developed to support the SIUE Robot Cape. These routines where bundled together into what we will call *robotLib*. This library handles the setup of the environment that will be used to control the motors by placing values into shared memory. As already mentioned, the shared memory is used by all three processors (ARM, PRU 0, PRU 1) to share information and to communicate with one another. The collection of functions that are available in *robotLib* are listed in Table 4.4, Table 4.5, and Table 4.6.

In Table 4.4 the first four functions are used for initialization related to memory and GPIO. These functions use a configuration string to setup variables used by the motor controller. The string is either read from a file (robot.config) or obtained from the GUI (Graphical User Interface) described in the next section. Using this configuration information the configPRU(), updateMotor(), and initSetpointPID() routines deal with actual loading of values into shared memory.

Functions	Description
initPin(int pinnum)	Initializes a GPIO pin with a corresponding GPIO number
setPinDirection(int pinnum, char* dir)	Sets the GPIO pin direction (input or output) based on the GPIO number and the direction specified. Inputs need to be setup as receivers in the device tree overlay.
setPinValue(int pinnum, int value)	Sets the output pin value to a logic high or low
getPinValue(int pinnum)	Returns the logic level value of the input pin
i2c_open(unsigned char bus, unsigned char addr)	Returns a handle to access a I2C device given a bus and an address
i2c_write(int handle, unsiged char* buf, unsiged char length)	Puts a char buffer of size length given the I2C handle
i2c_write_read(int handle, unsigned char addr_w, unsigned char *buf_w, unsigned int len_w, unsigned char addr_r, unsigned char *buf_r, unsigned int len_r)	Preforms a write/read operation on the I2C bus given the handle (necessary for reading from the I2C device as a write from the master telling the device what to send before the device controls the bus)
i2c_close(int handle)	Closes the I2C handle

Table 4.2: *bbbLib* Functions (1 of 2)

In many cases floating-point numbers are first converted to fixed-point before the values are stored in shared memory. The PRUs use encoder "tics" as a distance measure. Routines were created to convert from inches-to-tics and from tics-to-inches. Query functions are used to extract the motor and command status from shared memory and

Functions	Description
pauseSec(int sec)	Halts the program execution for a set number of seconds
delay_ms(unsigned int msec)	halts the program execution for a set number of microseconds
pauseNanoSec(long nano)	Halts the program execution for a set number of nanoseconds

Table 4.3: *bbbLib* Functions (2 of 2)

are also used in the waitForIdle() and waitrForComplete() functions.

Table 4.6 summarizes functions related to movement. Functions like fwd(), bwd(), rotate(), left() and right() call updateMotor() to initialize the DC motor structures and then eventually request movement by changing the command status value saved in shared memory. Figure 4.4 diagrams the sequence of events initiated by a fwd() call.

When fwd() is called, it checks to make sure that the motor is in an "idle" state. Following the check, it calls the updateMotor() routine which performs the necessary conversions from the requested operation in inches to tics as well as converts numbers from a floating-point to fixed-point format before storing these values into shared memory. After setup for movement is complete the status is set to "start" to notify the PRU to start motor operation given the current parameters.

The ARM core then waits for a complete calling waitForComplete and then sets the status to idle denoting that it is done with its operation. The rest of the movement commands follow a very similar flow with the rotate requiring a degree of rotation instead of the travel distance. These functions are all that is required for users to get a constructed robot moving after calling the appropriate setup functions providing ease of use to the user.

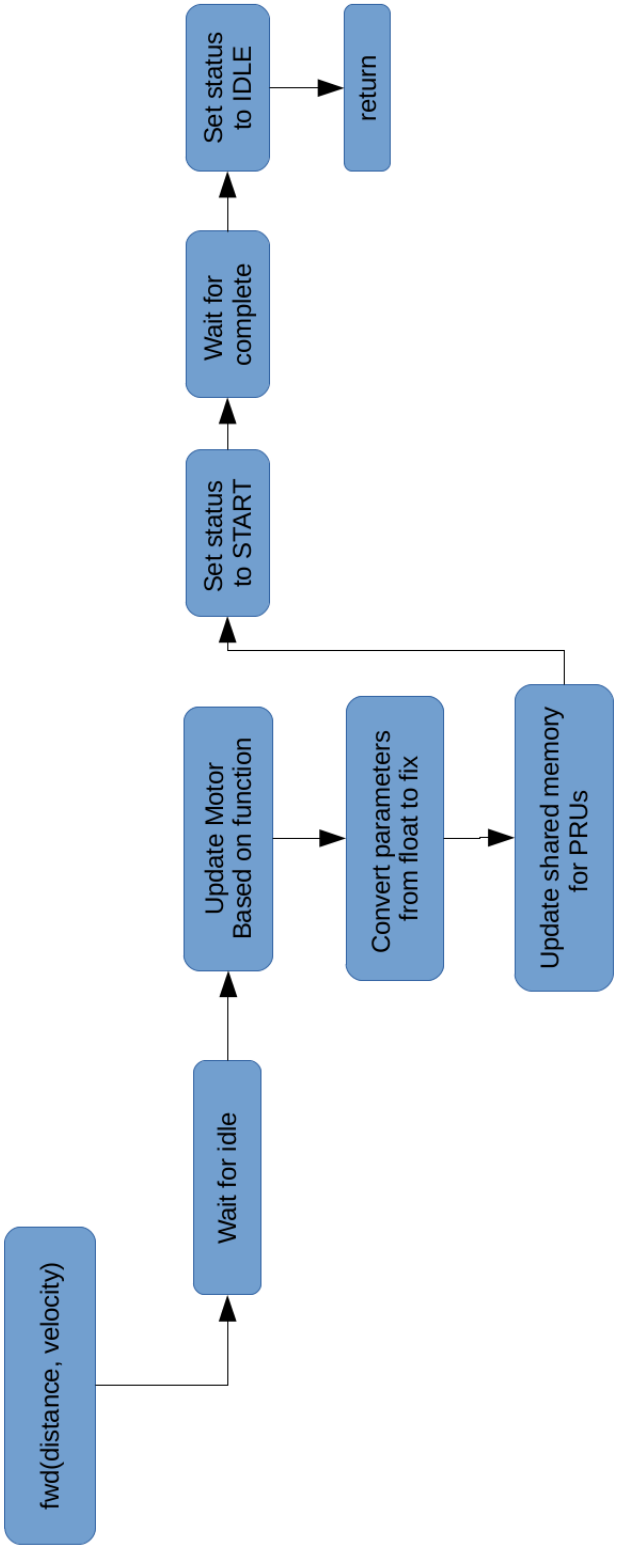

Figure 4.4: ARM fwd() Routine Flow Diagram

Functions	Description
GPIOinit(void)	Sets up the necessary GPIO pins such as the user LEDs and buffer enable pin
getGUIvars(char *str)	Uses sscanf to read the string outputted by the GUI environment to setup the values in the GUIvars structure which holds constants for the motor controller operation.
loadGuiVarsFromFile(char *str)	Opens a configuration file to setup the GUIvars structure.
configPRU(void)	Configures the PRUs, specificity the shared memory, with the values from GUIvars preforming necessary conversions in the process
initSetpointPID(void)	Setup shared memory with the values for the setpoint PID loop. Called in configPRU
resetPRU(void)	Sets the master PRU in a IDLE state
waitForIdle(void)	Waits until an IDLE state is set in shared memory by the master PRU
waitForComplete(void)	Waits until a COMPLETE state is set in memory by the master PRU

Table 4.4: *robotLib* Functions (1 of 3)

Functions	Description
inches2tics(float inches)	Returns a fixed point conversion of inches to inches based on shared memory values
tics2inches(int32_t tics)	Returns a float conversion of ticks to inches based on shared memory values
updateMotor(int motor_num, int dir, int brakeType, float distance, float velocity)	Updates the corresponding DC_motor_t structure in shared memory with the parameters for a movement operation. Performs the necessary conversions before updating the structure. This is called by the fwd, bwd, left, right, and rotate functions.
queryMotor(int motor_num, int item)	Returns the designated item from the designated DC_motor_t structure in shared memory (set-point, wheel direction, target distance, etc.)
query(int item)	Returns the designated item from shared memory, only for current command or status.
turnLED(int state)	Sets the GPIO LED on board based on the state.
buttonPress(void)	Returns the current value of the GPIO button.
memoryDump(void)	Prints values in shared memory to a text file in the execution directory.

Table 4.5: *robotLib* Functions (2 of 3)

Functions	Description
fwd(float distance, float velocity)	Requests a forward movement operation from the motor controller given the parameters. (velocity in inches per second)
bwd(float distance, float velocity)	Requests a backward movement operation from the motor controller given the parameters. (velocity in inches per second)
rotate(float degrees, float velocity, int direction)	Requests a rotation operation from the motor controller given the parameters. Rotates up to 360° in a clockwise or counter clockwise fashion.
left(void)	Calls rotate to turn the robot 90 degrees in a counter clockwise direction at 6" per second.
right(void)	Calls rotate to turn the robot 90 degrees in a clockwise direction at 6" per second.
applyBrake(void)	Requests a hard brake from the motor controller.

Table 4.6: *robotLib* Functions (3 of 3)

4.5 Graphical User Interface

The GUI (Graphical User Interface) for this application was created using Tcl/Tk (Tool Command Language with GUI Tool Kit). The GUI serves as system configuration interface. A Tcl shell is launched from main (*i.e.* beaglebot.c) and a two-way pipe in started. The two-way pipe allows the C code to communicate with the Tcl shell by simply reading from "stdin" and writing to "stdout". A command is sent to the Tcl shel to source the "gui.tcl" file. The configuration data is returned (via "stdin") to the C calling routine as a string. The parameters are colon delimited.

The advantage to the above approach is that the GUI can be debugged using a standard Tcl shell (on any platform that supports Tcl/Tk) and support for Tcl/Tk is readily available. The GUI Toolkit (Tk) makes the creation of a GUI a relatively simple and painless matter. Furthermore, the use of Tcl/Tk makes is easy for future SIUE teams to quickly and easily modify the GUI as the the need to do so arises. The colon-delimited string returned to the C calling routine is easily parsed. The paramters are stored in a global C structure called GUIvars.

The GUI screen is displayed in Figure 4.5. It contains a series of sliders, text boxes, and check boxes. The user can select which I2C peripherals (servos, sonars, accelerometer, real-time clock) are required. This is important since trying to access a non-existent device on an I2C bus will cause the ARM processor to hang. It is also important that the user select the motor type: stepper or DC. As described in earlier chapters, the hardware on the SIUE Robot Cape can support either 2 stepper or (up to) 4 DC motors. After selecting the motor type, it is important that the user enable the appropriate motor drivers. (The names used by the GUI correspond to the names silkscreened on the PCB).

While the user can select PWM resolution (8, 10, or 12 bits), testing revealed that the 8-bit mode should be used. Robot movement was **not** improved with 10 or 12 bit resolution and use of the other modes resulted in an audible tone when the motors were

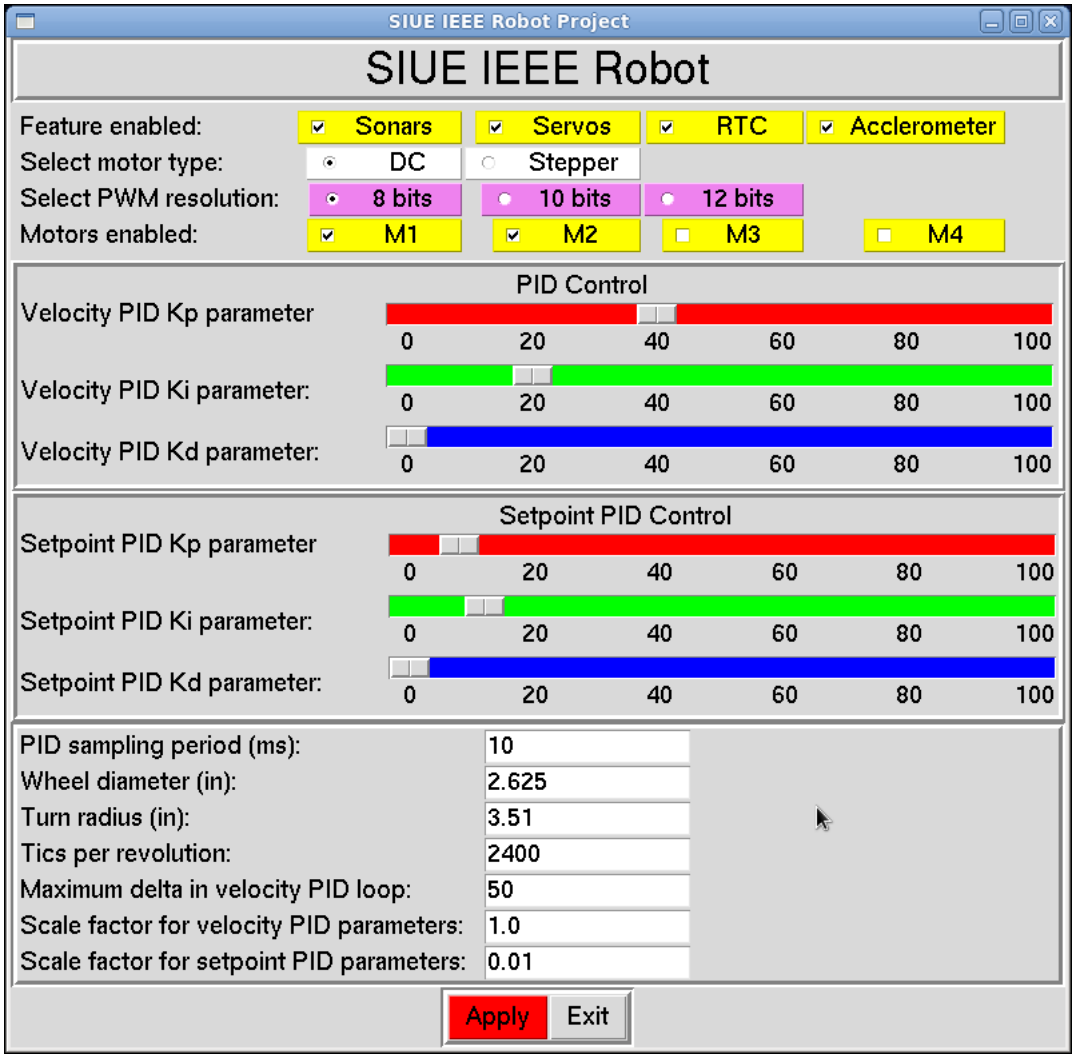

Figure 4.5: Tcl/Tk GUI Ensures Ease-Of-Use

running. The author suggests that the PWM resolution be removed from the GUI at some point in the future.

There are sliders for the PID gains which are multiplied by scale factors that can also be specified. The PID control sliders set the gain factors for the motor velocity PID loops. The *setpoint* PID sliders are used in the master PID loop that implments the differential velocity control. Lastly, there are text boxes so the user can change parameters like sampling period, wheel diameter, tics per revolution, turn radius, *etc.*

One important value that can be changed is the maximum "delta" used by the PID loops. Recall, the differential form of the PID algortion was used in this work, where the next PWM output is determined by taking the current value and adding a "delta". By being able to control the maximum "delta", we one prevent wheel slippage (by effectively limiting acceleration), and one can also help improve loop stability. Emperical evidence suggests that K_d can usually be set to zero in most mobile robot applications.

When all parameters are set, hitting the apply button will send the configuration string to "stdout". The string is parsed and the configuration data is stored in the global structure, GUIvars. This configuration data can then be accessed by the various routines in *robotLib*. The "Apply" button in the GUI turns from red to green in color when it is pressed.

4.6 Demonstration Program

The demonstration program (beaglebot.c) was created to demonstrate the movement capabilities of the board. As mentioned earlier, there is no environmental feedback for the demo robot other than encoder feedback so the only action it can really perform is driving which is exactly what it does! The demonstration program executes the following movements.

1. Drive forward for 48 inches at a speed of 12 inches per second.

2. Make a left turn and then forward at 12 inches per second for a distance of 24 inches.

3. Make a left turn and then forward at 12 inches per second fot a distance of 48 inches.

4. Make a left turn and then forward at 12 inches per second for a distance of 24 inches.

5. Pivot counter-clockwise 180° and traverse the path desribed above in reverse direction, making right turns, and then returning to orginal position.

6. Drive forward again at the same rate and perform another 180° turn.

7. This process repeats twice and then the robot stops in "precisely" the same spot and orientation from which it started,

All turns and pivots where executes at a speed of 6 inches per second. The slower speed on turns greatly improved reliability. The quality of the movements was judged on how "precisely" the robot returned to the starting position. In many of the mobile robot contests that our teams have participated in over the years, the robot must start at a prescribed location in the arena, navigate to other locations in the arena, and then return to the original starting location. The demonstration program was intended to replicate the type of movement that might be expected of the robot in future competitions.

4.7 Results of Navigation Test

When running the demo program the robot travels a total of 32 feet or a little over 10.6 yards before returning to its original starting position. The robot begins on a sheet of paper to mark its starting point and ends on the same piece of paper. This test attempts to demonstrate the accuracy of the motor controller over long distances of movement

where the systematic and non-systematic errors of robot movement would accumulate and could be easily visible.

In the trial runs, the robot perforned extremely well and returned to the sheet of paper each time with slight variation in its stopping position. The restults of two test runs can be seen in Figure 4.6 and Figure 4.7. While this is not a standardized test (they usually involve driving 12 feet rather than 4 feet as we did), the test is in the same spirit as the standardized test. The use of shorter distances is justified because it is consistent with movement required in the mobile robot arenas (typically just 8 feet x 8 feet) that our teams compete in.

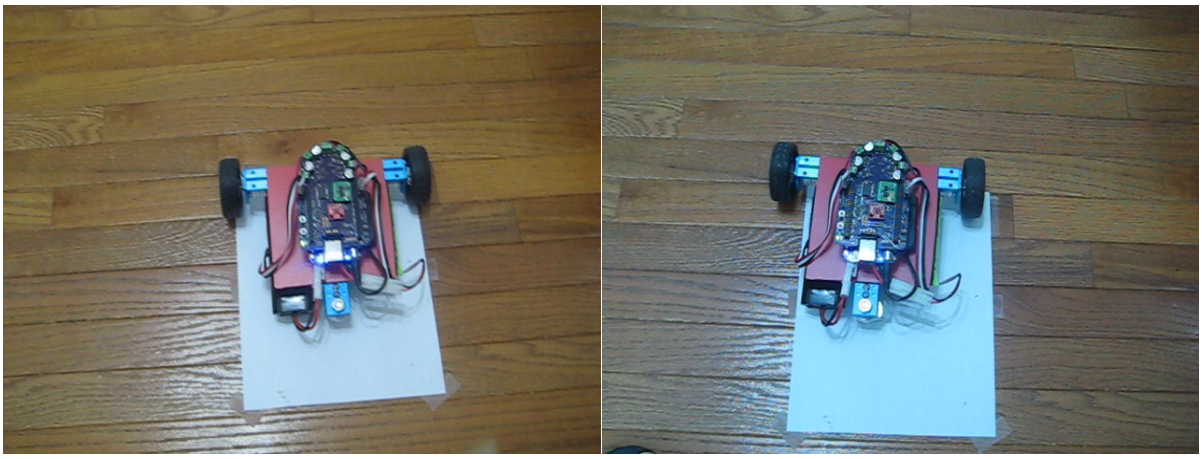

Figure 4.6: Robot Before (left) and After (right) Run - Trial #1

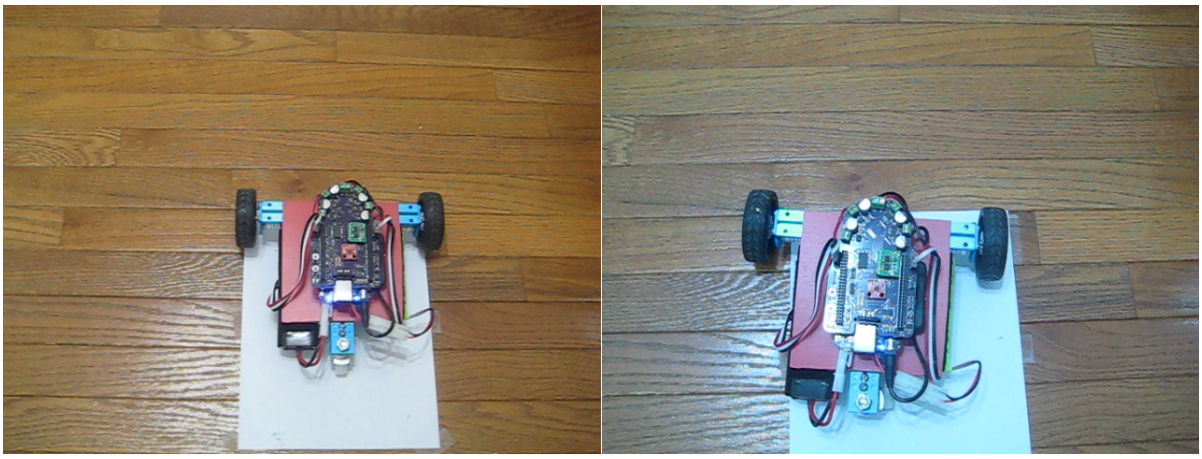

Figure 4.7: Robot before (left) and after (right) After Run, - Trial #2

CHAPTER 5

SUMMARY, COMCLUSIONS, AND FUTURE WORK

5.1 Summary

Using the Texas Instruments' BeagleBone Black, a custom board (the SIUE Robot Cape), and *Makeblock*® (mechanical parts), a small, easy-to-use, yet powerful robotics platform has been developed and the its usefulness has been demonstrated in this thesis. The platform which has been described in detail supports either four DC motors with quadrature wheel encoders or two stepper motors running open-loop. The ability to independently control four DC motors with encoder feedback affords the user the option of employing four omni-directional wheels in the robot design.

In addition to providing reliable movement, the SIUE Robot Cape supports two user I2C buses for quickly and easily accessing a wide variety of peripherals, a real-time clock, a 16-channel servo motor controller, and an accelerometer. The system can be powered from standard RC car NiMH batteries. The overall system strikes the right balance between flexibility and ease-of-use. In the pursuit of this end, a graphical user interface was developed to allow users to quickly configure the system and tune the various PID loops used for guaranteeing reliable navigation.

5.2 Conclusions

A large amount of work went into making the cape as easy to use as possible and also into making the platform suitable for use by SIUE student robotic groups in the future. Now that this platform is available, students and hobbyists may use the board in a number of different ways and applications. The cape is highly integrated with Texas Instruments' BeagleBone Black single-board computer, and the software described in this thesis can be easily changed to suit a myriad of mobile robot applications.

The PRUs do an excellent job of handling PID calculation, PWM signal generation, and reading of encoder feedback without taxing the main ARM system. The circuit board cape has a large amount of built in functionality and can be expanded by stacking capes or using other user circuit boards. With a two wheel differential drive design, the cape and related software desribed in this thesis performed well. The test robot produced a small amount of noticeable drift but with additional feedback (from the accelerometer), it could be made to be even more precise in its movements. Hopefully the SIUE Robot Cape will be useful for years to come. In short, the design objectives put forth in Chapter 1 have been succesively met.

5.3 Future Work

Much more work can be done to refine and improve the SIUE Robot Cape. Currently, only about half of the full functionality envisioned by the author at the onset of this project is available to the user. The following is list of possible improvements and work which still needs to be done in order to unlock the full potential of this cape.

- Add utilities for mecanum (omni-directional) wheel support, using all four motor drivers. At present, only two-wheel operation is fully supported by the software.
- Add support for arbitrary movement. In the current implementation only straight-line moves and pivots are supported.
- Add functionality for stepper motor use and control.
- Add the ability to make the the movement operations non-blocking. At present, control is not returned to the calling routine until the movement commands completes. While at times this is desirable, one can conceive of cases where it would be desirable to move on to another task while the move command is being executed. Because command status is stored in shared memory, the monitoring of whether the move command is complete or not could be left to another routine.

- Add support for threads. It is well-known that multi-threading is quite useful in robotic applications.
- Add RTC functionality for updating the system clock. Moreover, while the RTC was tested, at present no support for it is included in the current implementation of *robotLib*.
- Implement a better movement tracking sensor, one that not only measures acceleration readings but, through a digital gyroscope, angular velocity as well.
- Update to the 4.1 Kernal for use with the BeagleBone Green.
- Turn existing "libraries" into true C dynamically linkable libraries.

This is just a partial list, and the author is confident that future users of the cape and its software will only be limited by their own imaginations. It is the author's sincere hope that the work reported on in this thesis will spur future SIUE IEEE robotic teams to achieve great results in future competitions.

REFERENCES

[AdamBots, 2008] AdamBots (2008). Pid velocity control. `adambots.com/wiki/PID_Velocity_Control`. accessed : 2009-12-22.

[BeagleBoard.org, 2016] BeagleBoard.org (2016). Beagleboard:beagleboneblack. `http://elinux.org/Beagleboard:BeagleBoneBlack`. accessed : 2016-06-05.

[Benchoff, 2016] Benchoff, B. (2016). Introducing the raspberry pi 3. `https://hackaday.com/2016/02/28/introducing-the-raspberry-pi-3/`. accessed : 2016-6-9.

[Coley, 2014] Coley, G. (2014). *BeagleBone Black System Reference Manual*. BeagleBoard.org, revision c.1 edition.

[eLinux, 2016] eLinux (2016). Device tree. `http://elinux.org/Device_Tree`. accessed : 2015-09-12.

[Freescale-Semiconductor, 2013] Freescale-Semiconductor (2013). *Xtrinsic MMA8452Q 3-Axis, 12-bit/8-bit Digital Accelerometer*. Freescale-Semiconductor, rev. 8.1 edition.

[Maxim, 2015] Maxim (2015). *DS3231M 5ppm, I2C Real-Time Clock*. Maxim, 7 edition.

[Molloy, 2014] Molloy, D. (2014). *Exploring BeagleBone*. Wiley, Indiapolis, IN.

[NXP, 2014] NXP (2014). *PCA9306 Dual bidirectional I2C-bus and SMBus voltage-level translator*. NXP, pca9306 v.8 edition.

[Pololu, 2016] Pololu (2016). 3.4. quadrature encoders. `https://www.pololu.com/docs/0J63/3.4`. accessed : 2016-6-10.

[Tan, 2016] Tan, P. (2016). Dual motor controller cape (dmcc) mk.7. `http://exadler.myshopify.com/products/dual-motor-controller-cape-dmcc-mk-6`. accessed : 2016-6-17.

[Texas-Instruments, 2015] Texas-Instruments (2015). Pru linux application loader api guide. `http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader_API_Guide`. accessed : 2016-5-10.

[Texas-Instruments, 2016] Texas-Instruments (2016). *DRV8872 3.6-A Brushed DC Motor Driver With Fault Reporting (PWM Control)*. Texas Instruments, revision c edition.

[Yeong Chin Koo, 2015] Yeong Chin Koo, E. A. B. (2015). Motor speed controller for differential wheeled mobile robot. *ARPN Journal of Engineering and Applied Sciences*.

APPENDIX A

Global Defines and Structures

```
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Defines (mem.h)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// BUF_LEN is length of data buffer
// STR_LEN is length of string buffer
#define     BUF_LEN         200
#define     STR_LEN         250
// Motor defines
#define     NUM_MOTORS      4
#define     M1              0
#define     M2              1
#define     M3              2
#define     M4              3
#define     CRASH           -9999
// Wheel directions
#define   CW      0
#define   CCW     1
// Brake types
#define   COAST   0
#define   HARD    1
//
// Commands we can give PRU 0
//
#define   NOP            0
#define   FWD            1
#define   BWD            2
#define   ROT            3
#define   BRAKE_HARD     4
#define   BRAKE_COAST    5
#define   HALT_PRU       6
//
// Here the codes for status of commands
//
#define   CMD            1
#define   STATUS         2
#define   IDLE           0
#define   START          1
#define   ACTIVE         2
#define   COMPLETED      3
#define   ABORTED        4
// These are used when we query the motor structure
#define   SETPOINT       1
#define   DISTANCE       2
#define   TARGET_DIST    3
#define   WHEEL_DIR      4
#define   BRAKE_TYPE     5
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A structure that decribes a command from
// the ARM to PRU 0
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef struct {
    int32_t     code ;
    int32_t     status ;
} command_t ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Declare a structure to hold the GUI variables
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef struct {
    int      exitFlag ;
    int      sonarEna ;
    int      lineEna ;
    int      rtcEna ;
    int      accelEna ;
    int      motorType ;
    float    Kp ;
    float    Ki ;
    float    Kd ;
    float    samplePeriod ;
    float    wheelDiam ;
    float    turnRad ;
    float    ticsPerRev ;
    int      M1Ena ;
    int      M2Ena ;
    int      M3Ena ;
    int      M4Ena ;
    int      PWMresMode ;
    float    Kp_sp ;
    float    Ki_sp ;
    float    Kd_sp ;
    float    max_delta ;
    float    velPIDscale ;
    float    spPIDscale ;
} GUIvars_t ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A DC motor structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef struct {
    int32_t     setpoint ;          // desired velocity (in tics)
    int32_t     targetSetpoint ;    // will rammp up intil this is reached
    int32_t     deltaSetpoint ;     // steps we will take in ramping up
    int32_t     distance ;          // dist in tics (actual)
    int32_t     targetDistance ;    // dist in tics (desired)
    int32_t     wheelDirection ;    // CW or CCW
    int32_t     brakeType ;         // COAST or HARD
    int32_t     e0 ;                // current error
    int32_t     e1 ;                // past error
    int32_t     e2 ;                // past "past error"
    int32_t     Kp ;                // proportional gain (Q)
    int32_t     Ki ;                // integral gain (Q)
    int32_t     Kd ;                // deriviative gain (Q)
    int32_t     PWMmin ;            // minumum PWM out allowed
    int32_t     PWMmax ;            // maximum PWM out allowed
    int32_t     PWMout ;            // PWM output
    int32_t     max_delta ;         // Maximum change in PID output in a single step
}   DCmotor_t;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A PID structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef  struct {
    int32_t     e0 ;               // current error
    int32_t     e1 ;               // past error
    int32_t     e2 ;               // past past error
    int32_t     Kp ;               // proportional constant
    int32_t     Ki ;               // integral constant
    int32_t     Kd ;               // derivative constant
    int32_t     minVal ;           // minimum output
    int32_t     maxVal ;           // maximum output
    int32_t     output ;           // PID loop output
} PID_t ;
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Our shared memory structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
typedef struct {
    int32_t     pwm[NUM_MOTORS] ;      // shared mem byte os of 0
    int32_t     enc[NUM_MOTORS] ;      // os of 16
    int32_t     delay ;                // os of 32
    int32_t     state ;                // os of 36
    int32_t     PWMclkCnt ;            // os of 40
    int32_t     PWMres ;               // os of 44
    int32_t     exitFlag ;             // exit when true
    int32_t     interruptCounter ;     // sample counter
    int32_t     motorType ;            // DC or stepper
    int32_t     motorENA[NUM_MOTORS] ; // Motor enables
    int32_t     scr ;                  // scratchpad register
    int32_t     wheelDiam ;            // diameter in inches (Q)
    int32_t     ticsPerInch;           // encoder tics per inch (Q)
    int32_t     enc_data[BUF_LEN] ;    // Buffer of encoder data
    command_t   command ;              // Motor command structure
    DCmotor_t   motor[NUM_MOTORS] ;    // DC motor structure
    PID_t       setpointPID ;
}   shared_memory_t ;
```

APPENDIX B

Fixed-point Macros

```
//
// Macros that make doing fixed point operations easy (fix.h)
//
#define     PI          3.14159
#define     Q           6
#define     twoQ        12
#define     Q24         24
#define     Q12         12
#define     Q0          0
// Fixed point operationss
#define  FADD(op1,op2)      ( (op1) + (op2) )
#define  FSUB(op1,op2)      ( (op1) - (op2) )
#define  FMUL(op1,op2,q)    ((int32_t) (((int64_t) ((int64_t) (op1) * (int64_t) (op2)))
    >> q))
// #define  FDIV(op1,op2,q)    ( (int32_t) (((int64_t)(op1) << q)/ ((int64_t) op2 )) )
// Convert from a q1 format to q2 format
#define  FCONV(op1,q1,q2)     (((q2) > (q1)) ? ((op1) << ((q2)-(q1))) : ((op1) >> ((q1)
    -(q2))))
// Convert a float to a fixed-point representation in q format
#define  TOFIX(op1, q)       ((int32_t) ((op1) * ((float) (1 << (q)))))
// Convert a fixed-point number back to a float
#define  TOFLT(op1, q)       ( ((float) (op1)) / ((float) (1 << (q))) )
```

APPENDIX C

PRU 1 Assembly Code

C.1 pru1.h

```
//
// Defines for PRU #1 assembly code
//
#define         PRU_R31_VEC_VALID         32          // allows notification of program
    completion
#define         PRU_EVTOUT_1              4           // event number that is sent back
    for PRU 1 to ARM interrupt
#define         PRU0_PRU1_INTERRUPT       17          // PRU0->PRU1 interrupt number
#define         PRU1_PRU0_INTERRUPT       18          // PRU1->PRU0 interrupt number
#define         ARM_PRU1_INTERRUPT        37      //  ARM->PRU1 interrupt number
// Shared memory base addres AND
// Byte offsets for shared memory accesses
#define         PRU_SHARED_MEM_ADDR    0x00010000
#define         PWM_OS          0
#define         ENC_OS          16
#define         DELAY_OS        32
#define         STATE_OS        36
#define         CLK_CNT_OS      40
#define         PWM_RES_OS      44
// Define linux space GPIO access
#define         GPIO0                 0x44e07000
#define         GPIO1                 0x4804C000
#define         GPIO2                 0x481ac000
#define         GPIO3                 0x481ae000
#define         GPIO_CLEARDATAOUT     0x190           //Clearing GPIO
#define         GPIO_SETDATAOUT       0x194           //Setting GPIO
#define         GPIO_DATAOUT          0x138           //reading GPIO
/* gle
#define        GPIO_DATAOUT         0x13C
#define        GPIO_DATAIN          0x138
#define        GPIO_CLEARDATAOUT    0x190
#define        GPIO_SETDATAOUT      0x194
*/
#define        GPIO1_15_MASK     0x80       //SWITCH
#define        GPIO1_12_MASK     0x10       //LED
// Motor control signals
#define     M1_0   r30.t2
#define     M1_1   r30.t3
#define     M2_0   r30.t4
#define     M2_1   r30.t5
#define     M3_0   r30.t0
#define     M3_1   r30.t1
#define     M4_0   r30.t6
#define     M4_1   r30.t7
// Define register aliases
#define     nopReg            r0.b0
#define     enc1          r1
#define     enc2          r2
#define     enc3          r3
#define     enc4          r4
#define     pwm1          r5
#define     pwm2          r6
```

```
#define     pwm3           r7
#define     pwm4           r8
#define     encNEW            r9
#define     encOLD            r10
#define     encEDGE           r11
#define     GPIO_LED_STATE      r12
#define     GPIO_BUTTON         r13
#define     GPIO_LED            r14
#define     read_gpio1          r15
#define     set_gpio1           r16
#define     clr_gpio1           r17
#define     pwmResReg           r18
#define  stateReg       r19
#define  clkCntReg         r20
#define     i               r21
#define     j               r22
//Can be used to temporally hold values if needed
// scratchpad register
#define     scr                r23
// Currently not using the dela value
#define     delayValue         r24
// Shared memory base address
#define     sharedMem          r25
// R29 is used for subroutine calls
#define     M1_ctrl            stateReg.b0.t2
#define     M2_ctrl            stateReg.b0.t4
#define     M3_ctrl            stateReg.b0.t0
#define     M4_ctrl            stateReg.b0.t6
#define     run_flag             stateReg.b1.t0
#define     brake_flag           stateReg.b1.t1
#define     update_flag          stateReg.b1.t2
#define     halt_flag            stateReg.b1.t3
#define     enc1_bit        encEDGE.b1.t0
#define     enc2_bit        encEDGE.b1.t1
#define     enc3_bit        encEDGE.b1.t2
#define     enc4_bit        encEDGE.b1.t3
// Use r29 for subroutine calls
// Since r30.w0 is our output port!!!!!
// Else we get very odd behavior!
.setcallreg   r29.w0
// Define Macros
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_delay_value
//
// Description:
// read in the delay loop value from sharedMemory (written by host)
//
// At a byte offset of 32
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      read_delay_value
         lbbo  delayValue, sharedMem, DELAY_OS, 4
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// get_state
//
// Description:
// update the status register value in pru1 Memory
// The state value can be updated by pru0 and the ARM
// system if necesary it is used to determine wheel direction
// braking system, if a update of PWM values need to happen,
// or if the system should stop
//
// Status byte structure:  Wheel control:     b0
```

```
//          Status Flags:     b1-b2
//             b1.t0 - run flag    (1 if in run mode)
//             b1.t1 - brake flag  (1 if in hard brake)
//             b1.t2 - update flag (1 if an update for pwm)
//             b1.t3 - halt flag  (1 if we want to halt pru)
//             Nop (stays zero):  b3
//
// At a byte offset of 36
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.macro      get_state
         lbbo  stateReg, sharedMem, STATE_OS, 4
.endm
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// read_clk_cnt
//
// Description:
//   Tells us how many pwm clock cycles we should run
//   before interrupting PRU0
//
// At a byte offset of 40
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.macro      read_clk_cnt
         lbbo  clkCntReg, sharedMem, CLK_CNT_OS, 4
.endm
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// read_pwm_res
//
// Description:
// read in the pwm maximum count from sharedMemory (written by host)
// Either 255 (8 bit), 1023 (10 bit), or 4095 (12 bit)
//
// At a byte offset of 44
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.macro      read_pwm_res
         lbbo  pwmResReg, sharedMem, PWM_RES_OS, 4
.endm
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// pwm_timer
//
// Description:
// This decrements the PWM register given the motor
// timer register and will stop the pwm signal if
// necessary
//
// Usage:
//    pwm_timer   M1_ctrl, pwm1, M1_1, M1_1, NEXT_LINE
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.macro      pwm_timer
.mparam     M0_ctrl, pwm0, M0_0, M0_1
         qbbc  CCW, M0_ctrl      //Check if we are clockwise or counter clockwise
CW:         qbne  PWM_JMP, pwm0, 0  //Check if time to bring low
         clr       M0_0           //Set bit low
         qba       NEXT           //Jump to the next instruction
CCW:     qbne      PWM_JMP, pwm0, 0 //counter clockwise case
         clr       M0_1
         qba       NEXT
PWM_JMP:   dec     pwm0               //If it is not time to bring low decrement
         NO_OP                  //NO_OP to mimic jump
NEXT:
.endm
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// check_encoder_edges
//
```

```
// Description:
//      Transfers the old encoder values read the new ones
// and xor to see if there is an edge
//
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      check_encoder_edges
         mov   encOLD.b1, encNEW.b1
         mov   encNEW.b1, r31.b1
         xor   encEDGE.b1, encNEW.b1, encOLD.b1
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// zero_encoder_regs
//
// Description:
//   Clears the enc1, enc2, enc3, enc4 registers
//
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      zero_encoder_regs
            zero &enc1, 16
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// enc_cnt
//
// Description:
//      Reads the encoder tics and increments if need be
//
// Usage:
//      enc_cnt      enc1, enc1_bit
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      enc_cnt
.mparam     enc0, enc0_bit
            qbbc  ENC_JMP, enc0_bit //If clear jump to ENC_JMP
            inc       enc0          //If there is an edge increment
          qba      NEXT
ENC_JMP: NO_OP
          NO_OP
NEXT:
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// store_encoder_values
//
// Description:
// Stores the current encoder values to shared sharedMemory
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      store_encoder_values
         sbbo  &enc1, sharedMem, ENC_OS, 16
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// pwm_start
//
// Description:
// Uses the state register to start the pwm values
// stored in b0
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      pwm_start
         mov   r30.b0, stateReg.b0
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// brake
//
// Description:
// Stops the pwm by setting outputs to zero or to one
```

```
// depending on the brake bit
//
// Usage:
// stop_pwm  NEXT_LINE
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      brake
         qbbs  HARD_BR, brake_flag
         mov       r30.b0, 0x00
         qba       NEXT
HARD_BR: mov       r30.b0, 0xFF
         NO_OP
NEXT:
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// brake
//
// Description:
// Hard brake
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      hard_brake
         mov       r30.b0, 0xff
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_pwm_values
//
// Usage:
// read in PWM values from sharedMemory
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      read_pwm_values
         lbbo  pwm1, sharedMem, PWM_OS, 16
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// No operation
// (1 clock cycle)
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      NO_OP
         mov    nopReg, nopReg
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Clear All registers (R0-R28)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      clear_REGS
           zero    &r0, 116
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Turn the GPIO LED on (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      led_ON
            set    GPIO_LED_STATE, 12
            sbbo   GPIO_LED, set_gpio1, 0, 4
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Turn the GPIO LED off (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      led_OFF
            clr     GPIO_LED_STATE, 12
            sbbo    GPIO_LED, clr_gpio1, 0, 4
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Toggle the GPIO LED (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      led_TOGGLE
```

```
            xor     GPIO_LED_STATE.b1, GPIO_LED_STATE.b1, 1<<4
            qbbc    L0, GPIO_LED_STATE, 12
            led_ON
            qba     L1
L0:         led_OFF
L1:
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// QBPR : Quick branch if button press
//
// Usage:
// qbpr LOCATION
//
// Branches to target location if GPIO button is pressed
// (Green Button, p8.15, GPIO1[15], GPIO:47)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      qbpr
.mparam     LOCATION
           lbbo   GPIO_BUTTON, read_gpio1, 0, 4
            qbbs   LOCATION, GPIO_BUTTON.t15
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// movi32 : Move a 32bit value to a register
//
// Usage:
//     movi32   dst, src
//
// Sets dst = src. Src must be a 32 bit immediate value.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro   movi32
.mparam  dst, src
            mov     dst.w0, src & 0xFFFF
            mov     dst.w2, src >> 16
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// dec: Decrement value/register
//
// Usage:
// dec value
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      dec
.mparam     value
            sub     value, value, #1
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// inc: Increment value/register
//
// Usage:
// inc value
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro       inc
.mparam     value
         add   value,value, #1
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Copy a bit from source register to the destination register
// dreg is the destination register
// dbit is the bit number in the destination register
// sreg is the source register
// sbit is the bit number in the source register
// (4 clock cycles)
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      copy_bit
```

```
.mparam     dreg, dbit, sreg, sbit
            clr     dreg, dbit
            qbbc    END, sreg, sbit
            set     dreg, dbit
END:
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Macro to set up the GPIO utils
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      gpio_SETUP
            mov     read_gpio1, GPIO1 | GPIO_DATAOUT
            mov     set_gpio1, GPIO1 | GPIO_SETDATAOUT
            mov     clr_gpio1, GPIO1 | GPIO_CLEARDATAOUT
            set     GPIO_LED, 12
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// send_ARM_interrupt
//
// Description:
// Send an interrupt to the arm from pru1
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      send_ARM_interrupt
         mov   r31.b0, PRU_R31_VEC_VALID | PRU_EVTOUT_1
.endm
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// send_pru0_interrupt
//
// Description:
// Send an interrupt to pru0 from pru1
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      send_pru0_interrupt
         ldi   r31, PRU1_PRU0_INTERRUPT + 16
.endm
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Macro used to enable to OCP port
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      ocp_port_ENABLE
            lbco    r0, C4, 4, 4     // load SYSCFG reg into r0 (use c4 const addr)
           clr     r0, 4            // clear bit 4 (STANDBY_INIT)
           sbco    r0, C4, 4, 4     // store the modified r0 back at the load addr
.endm
```

C.2 pru1.p

```
// Assembly language code to run on pru 1
// PWM generation and reading of wheel encoders
#include    "./pru1.h"
.origin     0
.entrypoint START
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
START:
            clear_REGS
            movi32   sharedMem, PRU_SHARED_MEM_ADDR
         ocp_port_ENABLE
            read_pwm_res
            read_clk_cnt
// Keep reading state from shared_memory
// Waiting for PRU 0 to tell us to run
WAIT_FOR_RUN_FLAG:
            hard_brake
            get_state
            qbbc    WAIT_FOR_RUN_FLAG, run_flag
            zero_encoder_regs
MAIN:
            send_pru0_interrupt
            get_state
// See if run flag is still set
            qbbc    STOP, run_flag
// clkCntReg tells us how many PWM clocks we
// should generate before we interrupt PRU0
            mov     i, clkCntReg
I_LOOP:
          read_pwm_values
            pwm_start
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
// Will do the following either 256, 1024, or 4096 times
// depending on user selected PWM resolution
// Want to do this as fast as we possible can.
// Completing j-loop constitutes one PWM clock cycle.
         mov        j, pwmResReg
J_LOOP:
            check_encoder_edges
ENC_1_CHK:  enc_cnt      enc1, enc1_bit
ENC_2_CHK:  enc_cnt      enc2, enc2_bit
ENC_3_CHK:  enc_cnt      enc3, enc3_bit
ENC_4_CHK:  enc_cnt      enc4, enc4_bit
PWM_1_CHK:  pwm_timer    M1_ctrl, pwm1, M1_0, M1_1
PWM_2_CHK:  pwm_timer    M2_ctrl, pwm2, M2_0, M2_1
PWM_3_CHK:  pwm_timer    M3_ctrl, pwm3, M3_0, M3_1
PWM_4_CHK:  pwm_timer    M4_ctrl, pwm4, M4_0, M4_1
            dec     j
         qbne  J_LOOP, j, 0
            dec     i
            qbne    I_LOOP, i, 0
// End of sample period
// Write encoder values into shared memory and go back to MAIN
         sbbo  &enc1, sharedMem, ENC_OS, 16
            qba     MAIN
//
// We come here when run flag has been de-asserted
// Turn LED off!
//
STOP:
            hard_brake
         qbbs    HALT_PRU1, halt_flag
```

```
            qba     WAIT_FOR_RUN_FLAG
// HALT flag set .. we are quitting
HALT_PRU1:
            hard_brake
            send_ARM_interrupt
            halt
```

APPENDIX D

PRU 0 C Code

D.1 pru0.h

```
//
// Defines used by PRU 0
//
// 70 ms for 1,000,000
#define   KILL_TIME   1000000
// For convenience
#define   TRUE       1
#define   FALSE      0
//#define PRU addresses
/*
#define PRU0
#define HOST1_MASK      (0x80000000)
#define HOST0_MASK      (0x40000000)
#define PRU0_PRU1_EVT      (16)
#define PRU1_PRU0_EVT      (18)
#define PRU0_ARM_EVT    (34)
#define SHARE_MEM    (0x00010000)
*/
// Bit 15 is P8-11
// Bit 14 is p8-16
// Bit 07 is p9-25
// Bit 05 is p9-27
#define TOGGLE_PRU_LED     (__R30 ^= (1 << 15))      //Bit 15
#define OFF_PRU_LED     (__R30 &= (0xFFFF7FFF))
#define ON_PRU_LED         (__R30 |= (0x00008000))
#define DISABLE_DRV        (__R30 |= (1 << 5))       //Bit 5 Neg logic
#define ENABLE_DRV         (__R30 &= (0xFFFFFFDF))   //Bit 5 Neg logic
#define TOGGLE_DRV         (__R30 &= (1 << 5))       //Bit 5 Neg logic
#define PRU_SW_VALUE    (__R31  & (1 << 14))   //Bit 14
#define  ACC_IN1_VAL       (__R31  & (1 << 7))        //Bit 7
```

D.2 pru0.c

```
//
// This is the motor control code that will run on PRU 0
//
#include <stdint.h>
#include "pru_cfg.h"
#include "pru_intc.h"
#include "mem.h"
#include "pru0Lib.h"
#include "motorLib.h"
#include "pru0.h"
// Define input and output registers
volatile register uint32_t __R30;
volatile register uint32_t __R31;
/* Mapping Constant table register to variable */
volatile pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
volatile far pruIntc CT_INTC __attribute__((cregister("PRU_INTC", far), peripheral));
// Global pointer to memory stucture
shared_memory_t   *mem ;
// Global variables that allow us to handle GPIO
int  *clrGPIO1_reg ;
int  *setGPIO1_reg ;
int  *readGPIO1_reg ;
int  *clrGPIO3_reg ;
int  *setGPIO3_reg ;
int  *readGPIO3_reg ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to perform PRU initialization
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void initPRU(void) {
   CT_INTC.SICR = PRU1_PRU0_EVT ;
   initGPIO();
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to wait for an interrupt.
// It also clears the interrupt
// Toggles the PRU LED at interrupt rate
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void waitForInterrupt(void) {
   while (!(__R31 & HOST0_MASK)) { } ;  // wait for interrupt
   CT_INTC.SICR = PRU1_PRU0_EVT ;       // clear interrupt
   TOGGLE_PRU_LED ;
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to kill some time
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void killTime(int32_t delay) {
   int i ;
   for (i=0; i<delay; i++) ;
   return ;
} // end killTime()
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// MAIN PROGRAM STARTS HERE
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void main() {
// Point to 12 kB of shared memory
   mem = (shared_memory_t *) PRU_SHARED_MEM_ADDR ;
// Perform some PRU initialization tasks
   initPRU() ;
// Enable the motor driver signals
```

```
    enableBuffers() ;
// Keep implementing commands until we are told to exit
// Put in some delay.
// Don't want to check the status too rapidly so we
// kill some time ...
    while (!mem->exitFlag) {
        killTime(KILL_TIME) ;
        switch (mem->command.status) {
            case  IDLE:      break ;
            case  START:     doCommand(mem->command.code) ;
                             break ;
            case  ACTIVE:    break ;
            case  COMPLETED: break ;
            case  ABORTED:   break ;
        } // end switch
     } // end while
    doCommand(HALT_PRU) ;
// Disable the motor driver signals
    disableBuffers() ;
// Turn the PRU LED off
    OFF_PRU_LED;
//    GPIO1pin(LED_PIN, OFF) ;
    __R31 = 35;     // PRU 0 to ARM interrupt
    __halt();       // halt the PRU
} // end main
```

D.3 pru0Lib.h

```
//
//  Defines
//
// Misc defines
#define PRU0
#define HOST1_MASK      (0x80000000)
#define HOST0_MASK      (0x40000000)
//#define PRU0_PRU1_EVT (16)
#define PRU1_PRU0_EVT      (18)
// Shared memory address
#define   PRU_SHARED_MEM_ADDR     0x00010000
// GPIO bank addresses
#define      GPIO0                0x44e07000
#define      GPIO1                0x4804c000
#define      GPIO2                0x481ac000
#define      GPIO3                0x481ae000
// These can be OR'ed with the above bank addresses
#define      GPIO_DATAOUT         0x13C
#define      GPIO_DATAIN          0x138
#define      GPIO_CLEARDATAOUT    0x190
#define      GPIO_SETDATAOUT      0x194
// GPIO LED GPIO1[12]
#define      LED_PIN      12
// GPIO LED GPIO3[19]
#define      DRV_PIN      19
#define      OFF          0
#define      ON           1
//
// Library function prototype declarations
//
void       initGPIO(void) ;
void       GPIO1pin(int pin, int value) ;
void       blinkLED(void) ;
void       GPIO3pin(int pin, int value) ;
void       enableBuffers(void) ;
void       disableBuffers(void) ;
```

D.4 pru0Lib.c

```
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// Define some useful subroutines
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
#include <stdint.h>
#include "pru_cfg.h"
#include "pru_intc.h"
#include "pru0Lib.h"
// Global variables that allow us to handle GPIO
extern   int   *clrGPIO1_reg ;
extern   int   *setGPIO1_reg ;
extern   int   *readGPIO1_reg ;
extern   int   *clrGPIO3_reg ;
extern   int   *setGPIO3_reg ;
extern   int   *readGPIO3_reg ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// GPIO 0 intialization
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void initGPIO(void) {
   clrGPIO1_reg = (int *) (GPIO1 | GPIO_CLEARDATAOUT) ;
   setGPIO1_reg = (int *) (GPIO1 | GPIO_SETDATAOUT) ;
   readGPIO1_reg = (int *) (GPIO1 | GPIO_DATAIN) ;
   clrGPIO3_reg = (int *) (GPIO3 | GPIO_CLEARDATAOUT) ;
   setGPIO3_reg = (int *) (GPIO3 | GPIO_SETDATAOUT) ;
   readGPIO3_reg = (int *) (GPIO3 | GPIO_DATAIN) ;
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to write to GPIO1 pin
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void GPIO1pin(int pin, int value) {
   switch (value) {
      case OFF:   *clrGPIO1_reg = *readGPIO1_reg  | (1 << pin) ;
                  break ;
      case ON:    *setGPIO1_reg = *readGPIO1_reg |  (1 << pin) ;
                  break ;
   }
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to write to GPIO3 pin
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void GPIO3pin(int pin, int value) {
   switch (value) {
      case OFF:   *clrGPIO3_reg = *readGPIO3_reg  | (1 << pin) ;
                  break ;
      case ON:    *setGPIO3_reg = *readGPIO3_reg |  (1 << pin) ;
                  break ;
   }
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to enable buffers
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void enableBuffers(void) {
   GPIO3pin(DRV_PIN, 0) ;
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to disable buffers
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void disableBuffers(void) {
```

```
    GPIO3pin(DRV_PIN, 1) ;
    return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine slowly blink LED
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void blinkLED(void) {
    static int LEDstate = OFF ;
    if (LEDstate == ON) {
        GPIO1pin(LED_PIN, OFF) ;
        LEDstate = OFF ;
    } else {
        GPIO1pin(LED_PIN, ON) ;
        LEDstate = ON ;
    } // end else
    return ;
} // blink LED
```

D.5 motorLib.h

```
//
// Function prototype declarations
//
//
// Defines used by motorLib
//
#define   FALSE   0
#define   TRUE    1
#define   M_RUN           (1 << 8)
#define   M_STOP          (~M_RUN)
#define   M_HARD_BRAKE    (1 << 9)
#define   M_UPDATE        (1 << 10)
#define   M_HALT      (1 << 11)
#define   M_COAST     0
// Wheel directions
#define   CW      0
#define   CCW     1
// Motor control for state register
#define   M1_CW      (0x00000004)
#define   M1_CCW  (0x00000008)
#define   M2_CW      (0x00000010)
#define   M2_CCW  (0x00000020)
#define   M3_CW      (0x00000001)
#define   M3_CCW  (0x00000002)
#define   M4_CW      (0x00000040)
#define   M4_CCW  (0x00000080)
void      haltPRU(void) ;
void      hardBrake(void) ;
void      coast(void) ;
int32_t   PID(DCmotor_t *motor, int32_t enc) ;
void      move(void) ;
void      doCommand(int32_t  command_code) ;
void      adjustSetpoint(int32_t rightVel, int32_t  leftVel) ;
int32_t   createState(void) ;
```

D.6 motorLib.c

```
//
// Library of routines which run on PRU0
// for handling the motors
//
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include "mem.h"
#include "fix.h"
#include "motorLib.h"
// Pointer to shared memory is a global variable
  extern   shared_memory_t   *mem ;
// Wait for interrupt
  void    waitForInterrupt(void) ;
// ********************************************************
// Routine to halt PRU #1
// Sets halt flag bit in state variable
// We also have to clear the run flag.
// ********************************************************
void  haltPRU(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_HALT ;
   return ;
}
// ********************************************************
// Routine to apply hard brake
// ********************************************************
void hardBrake(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_HARD_BRAKE ;
   return ;
}
// ********************************************************
// Routine to coast to a stop
// ********************************************************
void coast(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_COAST ;
   return ;
}
// ************************************************************
// Routine to implement PID loop on a single DC motor
// Using the velocity or differential PID
//
// delta_p = Kp * error
// delta_i = Ki * (error - past_error)
// delta_d = Kd * (error - past_error + 2 * past_past_error)
//
// output = previous_output + (delta_p + delta_i + delta_d)
//
// Errors in Q6 format.
// enc in Q0 format so convert it to Q6 format
// Kp, Ki, Kd are in Q6 format,
// delta_p, delta_i, and delta_d are in Q6 format
// Output in Q0 format.
//
// ************************************************************
int32_t PID(DCmotor_t * motor, int32_t enc) {
   int32_t   delta_p, delta_i, delta_d ;
   int32_t   scr, out ;
   int32_t   max_delta ;
```

```
    max_delta = motor->max_delta ;
// Update past_error and past_past_error
    motor->e2 = motor->e1 ;
    motor->e1 = motor->e0 ;
// Compute new error term
// First convert enc value to Q6 format
// Motor setpoint stored in Q6 format
    enc = FCONV(enc, 0, Q) ;
    motor->e0 = FSUB(motor->setpoint, enc) ;
// Compute delta_p, delta_i, and delta_d
// Deltas will be in Q12 format
    delta_p = FMUL(motor->Kp, motor->e0, 0) ;
    scr = FSUB(motor->e0, motor->e1) ;
    delta_i = FMUL(motor->Ki, scr, 0) ;
    scr = FADD(scr, motor->e2 << 1) ;
    delta_d = FMUL(motor->Kd, scr, 0) ;
    scr = FADD(delta_i, delta_d) ;
    scr = FADD(scr, delta_p) ;
// Convert the delta from Q12 to Q0 format
    scr = FCONV(scr, twoQ, 0) ;
//
// Limit the delta
// This helps ensure stability and keeps the
// robot from "lurching" forward
//
    if (scr > max_delta) scr = max_delta ;
    if (scr < -max_delta) scr = -max_delta ;
// Make sure "out" is in range
    out = FADD(motor->PWMout, scr)  ;
    if (out > motor->PWMmax) {
        out = motor->PWMmax ;
    } else {
        if (out < motor->PWMmin) {
            out = motor->PWMmin ;
        } // end if
    } // end if-then-else
// Save the output and also return it
    motor->PWMout = out ;
    return    out ;
}
// ********************************************************************
// Routine to create state variable
// ********************************************************************
int32_t  createState(void) {
    int32_t  state ;
// Look at direction field so we can set the state correctly
// Also look at the breaking mode so when we stop.
// We do so either by braking hard or by coasting.
    state = 0 ;
    if (mem->motorENA[M1]) {
         if (mem->motor[M1].wheelDirection == CW) {
            state |= M1_CW ;
         } else {
            state |= M1_CCW ;
         } // end if-then-else
         if (mem->motor[M1].brakeType == HARD) {
            state |= M_HARD_BRAKE ;
         } // end if
    } // end if
    if (mem->motorENA[M2]) {
         if (mem->motor[M2].wheelDirection == CW) {
            state |= M2_CW ;
         } else {
```

```
            state |= M2_CCW ;
         } // end if-then-else
         if (mem->motor[M2].brakeType == HARD) {
            state |= M_HARD_BRAKE ;
         } // end if
   } // end if
   if (mem->motorENA[M3]) {
         if (mem->motor[M3].wheelDirection == CW) {
            state |= M3_CW ;
         } else {
            state |= M3_CCW ;
         } // end if-then-else
         if (mem->motor[M3].brakeType == HARD) {
            state |= M_HARD_BRAKE ;
         } // end if
   } // end if
   if (mem->motorENA[M4]) {
         if (mem->motor[M4].wheelDirection == CW) {
            state |= M4_CW ;
         } else {
            state |= M4_CCW ;
         } // end if-then-else
         if (mem->motor[M4].brakeType == HARD) {
            state |= M_HARD_BRAKE ;
         } // end if
   } // end if
// Set the run bit
   state |= M_RUN ;
   return state ;
} // end createState()
// ********************************************************************
// Routine to move.
// ********************************************************************
void move(void) {
   int        i ;
   DCmotor_t  *motor ;
// Set status to ACTIVE
   mem->command.status = ACTIVE ;
// Set the errors to zero
// Also set the distance traveled to 0
// The setpoint, brake mode, wheel direction
// and target distance all get set by the routines
// in robotLib. Also clear out pwm array.
// enc array is cleared in PRU 1 asm code
   for (i=0; i<NUM_MOTORS; i++) {
      mem->motor[i].e0 = 0 ;
      mem->motor[i].e1 = 0 ;
      mem->motor[i].e2 = 0 ;
      mem->motor[i].distance = 0 ;
      mem->motor[i].PWMout = 0 ;
      mem->pwm[i] = 0 ;
      mem->enc[i] = 0 ;
   } // end for
//
// Initialize the setpoint PID loop also
//
  mem->setpointPID.e0 = 0 ;
  mem->setpointPID.e1 = 0 ;
  mem->setpointPID.e2 = 0 ;
// Main loop
// Keep looping until distance traveled on a single
// motor exceeds the target distance.
// PRU 1 will interrupt us at the desired sample rate.
```

```
// The waitForInterrupt routine toggles the PRU LED
// at the sample rate to provide user with visual feedback.
//
   int32_t    velInTics[NUM_MOTORS] ;
   int        loop = TRUE ;
// Write state out to shared memory
// PRU1 should start generating PWM outputs
   int32_t  state ;
   state = createState() ;
   mem->state = state ;
   while (loop) {
     waitForInterrupt() ;
     for (i=0; i < NUM_MOTORS; i++) {
         if (mem->motorENA[i]) {
            motor = &mem->motor[i] ;
            velInTics[i] = mem->enc[i] - motor->distance ;
            motor->distance = mem->enc[i] ;
            if ((motor->distance) <= (motor->targetDistance)) {
                mem->pwm[i] = PID(motor, velInTics[i]) ;
            } else {
                loop = FALSE ;
                mem->state = M_STOP & mem->state ; // clear run bit
            } // end if-then-else
         } // end if
      } // end for
// This is for two wheels only.
// Implement a PID loop to adjust setpoints on two wheels so
// that the two wheels will travel at same velocity.
      adjustSetpoint(velInTics[M2], velInTics[M1]) ;
   } // end while
   return  ;
} // end move()
// ****************************************************************
//  Executes the command
//  The move() routine will look at the state and call appropriate
//  subroutine.  Upon return from the called routine, the status
//  will be updated to reflect that the commmand has COMPLETED.
//  The move() routine looks at the mem->motor struct to figure
//  out exactly what is must do.
// ****************************************************************
void doCommand(int32_t  command_code) {
   switch (command_code) {
       case  NOP:         mem->command.status = IDLE ;
                          break ;
       case  FWD:         move() ;
                          mem->command.status = COMPLETED ;
                          break ;
       case  BWD:         move() ;
                          mem->command.status = COMPLETED ;
                          break ;
       case  ROT:         move() ;
                          mem->command.status = COMPLETED ;
                          break ;
       case BRAKE_HARD:   hardBrake() ;
                          mem->command.status = COMPLETED ;
                          break ;
       case BRAKE_COAST:  coast() ;
                          mem->command.status = COMPLETED ;
                          break ;
       case HALT_PRU:     haltPRU() ;
                          mem->command.status = COMPLETED ;
                          break ;
       default:           mem->command.status = IDLE ;
```

```
                        break ;
    } // end switch
   return ;
} // end do_command()
// ****************************************************************
// Implements the setpoint PID loop
//
// delta_p = Kp * error
// delta_i = Ki * (error - past_error)
// delta_p = Kp * error
// delta_i = Ki * (error - past_error)
// delta_d = Kd * (error - past_error + 2 * past_past_error)
//
// output = previous_output + (delta_p + delta_i + delta_d)
//
// Errors in Q6 format.
// Kp, Ki, Kd are in Q6 format,
// delta_p, delta_i, and delta_d are in Q6 format
// Output in Q0 format.
//
// output = previous_output + (delta_p + delta_i + delta_d)
//
// Errors in Q6 format.
// Kp, Ki, Kd are in Q6 format,
// delta_p, delta_i, and delta_d are in Q6 format
// Output in Q6 format.
// ****************************************************************
void adjustSetpoint(int32_t rightVel, int32_t  leftVel)  {
// Compute difference in wheel velocities
// Convert to Q format
    int32_t   velDiff, scr ;
    int32_t   delta_p, delta_i, delta_d, delta_sp ;
// Compute the error
// M2 is the right motor
// M1 is the left motor
// If velDiff is positive then we will need to slow
// down M2 and speed up M1!!!!
    velDiff = rightVel - leftVel ;
// Update past_error (e1) and past_past_error (e2)
    mem->setpointPID.e2 = mem->setpointPID.e1 ;
    mem->setpointPID.e1 = mem->setpointPID.e0 ;
// Compute new error term
// We want it in Q format where currently Q is 6
    mem->setpointPID.e0 = FCONV(velDiff, 0, Q) ;
// Compute delta_p, delta_i, and delta_d
    delta_p = FMUL(mem->setpointPID.Kp, mem->setpointPID.e0, 0) ;
    scr = FSUB(mem->setpointPID.e0, mem->setpointPID.e1) ;
    delta_i = FMUL(mem->setpointPID.Ki, scr, 0) ;
    scr = FADD(scr, mem->setpointPID.e2 << 1) ;
    delta_d = FMUL(mem->setpointPID.Kd, scr, 0) ;
    scr = FADD(delta_i, delta_d) ;
    scr = FADD(scr, delta_p) ;
// Convert the delta from 2Q to Q format
    scr = FCONV(scr, twoQ, Q) ;
    delta_sp = FADD(mem->setpointPID.output, scr)  ;
// Make sure "out" is in range
    if (delta_sp > mem->setpointPID.maxVal) {
        delta_sp = mem->setpointPID.maxVal ;
    } else {
        if (delta_sp < mem->setpointPID.minVal) {
            delta_sp = mem->setpointPID.minVal ;
        } // end if
    } // end if-then-else
```

```
// Save the output
   mem->setpointPID.output = delta_sp ;
// Adjust the setpoint for the two wheel PID loops
   mem->motor[M1].deltaSetpoint = delta_sp ;
   mem->motor[M2].deltaSetpoint = delta_sp ;
// We adjust in a differential manner!
   mem->motor[M2].setpoint = mem->motor[M2].targetSetpoint - delta_sp ;
   mem->motor[M1].setpoint = mem->motor[M1].targetSetpoint + delta_sp ;
   return ;
}  // end adjustSetpoint()
```

APPENDIX E

ARM C Code

E.1 beaglebot.c

```
//
// Main program for demonstration robot (beaglebot.c)
//
#include   <stdio.h>
#include   <stdlib.h>
#include   <assert.h>
#include   <stdint.h>
#include   <math.h>
#include   "prussdrv.h"
#include   "pruss_intc_mapping.h"
#include   "mio.h"
#include   "child.h"
#include   "bbbLib.h"
#include   "mem.h"
#include   "PRUlib.h"
#include   "robotLib.h"
// TRUE and FALSE
#define   TRUE   1
#define   FALSE  0
// GUI mode or not
int         GUImode = TRUE ;
// We need a global variable that wil point to the shared memory
shared_memory_t  *shared_memory ;
// Need a variable for debugging
int          debug = TRUE ;
// Need global structure to hold GUI variables
GUIvars_t   GUIvars ;
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Routine to execute PRU programs
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void execPRUprograms() {
// Initialize the PRUs
   if (debug) printf("Initializing the PRUs.\n") ;
   PRUinit() ;
// Configure PRU 0 based on GUI settings
   if (debug) printf("Configuring PRU 0 with GUI data\n") ;
   configPRU() ;
// Start the PRUs
   if (debug) printf("Start the PRUs ...\n") ;
   PRUstart() ;
   return ;
}
// ********************************
// Main program
// ********************************
int main (void) {
   FILE    *read_from, *write_to;
   char    str[STR_LEN] ;
   int     exitFlag = FALSE ;
   int     runFlag = FALSE ;
// Print a welcome statement
   if (debug) printf("\nSIUE Beaglebot Project\n") ;
```

```
   if (debug) printf("24-Jul-2016\n\n") ;
// GPIO initialization
   if (debug) printf("Initializing the GPIOs which we will use ...\n") ;
   GPIOinit() ;
   turnLED(OFF) ;
// Look to see if user wants to use the GUI
   if (GUImode) {
// Start the gui
// Two-way pipe
      start_child("tclsh", &read_from, &write_to);
      fprintf(write_to, "source ./tcl/gui.tcl \n") ;
// Get data from GUI
      while (!exitFlag) {
          if  (fgets(str, STR_LEN, read_from) != NULL) {
             getGUIvars(str) ;
             exitFlag = GUIvars.exitFlag ;
             if (!exitFlag) {
                if (!runFlag) {
                   execPRUprograms() ;
                   testRobot() ;
                   runFlag = TRUE ;
                } else {;
                   PRUstop() ;
                   execPRUprograms() ;
                   testRobot() ;
                } // end if then elseif
             } // end if
          } //end if
      } // end while
   } else {          // NO GUI!!!!
// Get the GUI string from the robot.config file
// and parse it as usual
        loadGuiVarsFromFile(str) ;
        getGUIvars(str) ;
// The string we load may have the exit flag set
// We don' want this ...
        GUIvars.exitFlag = FALSE ;
// Load, configure, and start the PRUs
        execPRUprograms() ;
// Run the the test robot code
        testRobot() ;
   } // end if-the-else
// User wants to exit so let PRU0 know
   exitFlag = TRUE ;
   shared_memory->exitFlag = TRUE ;
// Delay for 1 sec before disabling PRUs
   if (debug) memoryDump() ;
   pauseSec(1) ;
   PRUstop() ;
   return 0;
} // end main
```

E.2 PRUlib.h

```
//
// PRUlib.h
//
#define      PRU0     0
#define      PRU1     1
// Function declarations
void PRUinit(void) ;
void PRUstop(void) ;
void PRUstart(void) ;
```

E.3 PRUlib.c

```
// ********************************
// PRUlib.c
// ********************************
#include "prussdrv.h"
#include "pruss_intc_mapping.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include "mem.h"
#include "PRUlib.h"
// Global variable that points to shared memory
extern  shared_memory_t  *shared_memory ;
// Debug variable
extern  int   debug ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// PRU initialization routine
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void PRUinit(void) {
// Initialize structure used by prussdrv_pruintc_intc
// PRUSS_INTC_INITDATA is found in pruss_intc_mapping.h
   tpruss_intc_initdata pruss_intc_initdata = PRUSS_INTC_INITDATA;
/* Allocate and initialize memory */
   prussdrv_init ();
// For PRU 0
   prussdrv_open (PRU_EVTOUT_0);
// For PRU 1
   prussdrv_open (PRU_EVTOUT_1);
// Map PRU's INTC
   prussdrv_pruintc_init(&pruss_intc_initdata);
// Set up pointer to shared memory
   static void *pruSharedDataMemory;
   prussdrv_map_prumem(PRUSS0_SHARED_DATARAM, &pruSharedDataMemory);
   shared_memory = (shared_memory_t *) pruSharedDataMemory;
   return ;
}
// ************************************
// Routine to clean up after the PRUs
// ************************************
void PRUstop(void) {
/* Disable PRU and close memory mappings */
   if (debug) printf("Disabling the PRUs\n") ;
   prussdrv_pru_disable(PRU0);
   prussdrv_pru_disable(PRU1);
   prussdrv_exit ();
   return ;
}
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Here is a subroutine to interact with the PRUs
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void PRUstart(void) {
// Load and execute binary on PRU0
// Since using C-code for PRU, we need to give START_ADDR
   if (debug) printf("Starting PRU0 program\n") ;
   prussdrv_exec_program_at(PRU0, "./text.bin", START_ADDR);
/* Load and execute binary on PRU1 */
   if (debug) printf("Starting PRU1 program\n") ;
```

```
    prussdrv_exec_program(PRU1, "./pru1.bin");
    return ;
}
```

E.4 robotLib.h

```
//
// robotLib.h
//
#define    FALSE    0
#define    TRUE     1
#define    PASS     1
#define    FAIL     0
#define    M_RUN           (1 << 8)
#define    M_HARD_BRAKE    (1 << 9)
#define    M_UPDATE        (1 << 10)
#define    M_HALT       (1 << 11)
// Motor control for state register
#define    M1_CW       (0x00000004)
#define    M1_CCW   (0x00000008)
#define    M2_CW       (0x00000010)
#define    M2_CCW   (0x00000020)
#define    M3_CW       (0x00000001)
#define    M3_CCW   (0x00000002)
#define    M4_CW       (0x00000040)
#define    M4_CCW   (0x00000080)
// Period of PWM clock in ms
// Measure on the scope and enter
// correct value here
#define      PWM_CLK_PERIOD_12BIT      0.8
#define      PWM_CLK_PERIOD_10BIT      0.2
#define      PWM_CLK_PERIOD_8BIT       0.05
// PWM resolution modes
#define      BITS_IS_8           1
#define      BITS_IS_10          2
#define      BITS_IS_12          3
#define      GPIO_LED_PIN       44
#define      GPIO_SW_PIN        47
#define      ACCEL_PIN           2
#define      DRV_PIN           115
// Delay (in ms) to wait between status checks
#define      STATUS_DELAY       50
// Function prototype declarations
void     getGUIvars(char *str) ;
void     loadGuiVarsFromFile(char *str) ;
void     GPIOinit(void) ;
void     turnLED(int state) ;
int      buttonPress(void) ;
void     configPRU(void) ;
void     memoryDump(void) ;
int32_t  inches2tics(float inches) ;
float    tics2inches(int32_t tics) ;
void     updateMotor(int motor_num, int dir, int brakeType, float distance, float
    velocity) ;
int32_t  queryMotor(int motor_num, int item) ;
int32_t  query(int item) ;
void     resetPRU(void) ;
void     waitForIdle(void) ;
void     waitForComplete(void) ;
int      fwd(float distance, float velocity) ;
int      bwd(float distance, float velocity) ;
int      rotate(float degrees, float velocity, int direction) ;
int      right(void) ;
int      left(void) ;
int      releaseBrake (void) ;
int      applyBrake (void) ;
void     testDrive(void) ;
```

```
void    testSonar(void) ;
void    testServo(void) ;
void    testRobot(void) ;
void    initSetpointPID(void) ;
```

E.5 robotLib.c

```
//
// robotLib
//
#include    <stdio.h>
#include    <stdlib.h>
#include    <stdint.h>
#include    <math.h>
#include    <string.h>
#include   "bbbLib.h"
#include   "fix.h"
#include   "mem.h"
#include   "srf02.h"
#include   "servo_driver.h"
#include   "robotLib.h"
// Global variable
// Pointer to shared memory
  extern shared_memory_t  *shared_memory ;
// GUI variables
  extern  GUIvars_t       GUIvars ;
// Debug variable
  extern  int             debug ;
// ******************************************
// Routine to initialize the GPIO we need
// ******************************************
void GPIOinit(void) {
   initPin(ACCEL_PIN) ;                     // GPIO0[2] Accel GPIO interrupt
   setPinDirection(ACCEL_PIN, IN) ;
   initPin(GPIO_LED_PIN) ;                  // GPIO1[12] GPIO LED
   setPinDirection(GPIO_LED_PIN, OUT) ;
   initPin(GPIO_SW_PIN) ;                   // GPIO1[15] GPIO SWITCH
   setPinDirection(GPIO_SW_PIN, IN) ;
   initPin(DRV_PIN) ;                       // GPIO3[19] buffer enable
   setPinDirection(DRV_PIN, OUT) ;
   setPinValue(DRV_PIN, ON) ;
   return ;
} // end GPIOinit()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Rotuine to read the GPIO switch
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int buttonPress(void) {
   int  value ;
   value = getPinValue(GPIO_SW_PIN) ;
   return value ;
} //
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to turn GPIO LED (GPIO1[12]) board on or OFF
// 0 is OFF and 1 is ON
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void turnLED(int state) {
    setPinValue(GPIO_LED_PIN, state) ;
    return ;
} // end turnLED()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to get GUI variables
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void getGUIvars(char *str) {
   FILE   *fid ;
// Save the str sent back from tcl script to a file
   fid = fopen("./robot.config", "w") ;
   fprintf(fid, "%s\n", str) ;
   fclose(fid) ;
```

```
// Parse the string sent back from the gui
   sscanf(str, "%d:%d:%d:%d:%d:%d:%f:%f:%f:%f:%f:%f:%f:%d:%d:%d:%d:%d:%f:%f:%f:%f:%f:%f"
       ,
          &GUIvars.exitFlag,
          &GUIvars.sonarEna,
          &GUIvars.lineEna,
          &GUIvars.rtcEna,
          &GUIvars.accelEna,
          &GUIvars.motorType,
          &GUIvars.Kp,
          &GUIvars.Ki,
          &GUIvars.Kd,
          &GUIvars.samplePeriod,
          &GUIvars.wheelDiam,
          &GUIvars.turnRad,
          &GUIvars.ticsPerRev,
          &GUIvars.M1Ena,
          &GUIvars.M2Ena,
          &GUIvars.M3Ena,
          &GUIvars.M4Ena,
        &GUIvars.PWMresMode,
        &GUIvars.Kp_sp,
        &GUIvars.Ki_sp,
        &GUIvars.Kd_sp,
          &GUIvars.max_delta,
          &GUIvars.velPIDscale,
          &GUIvars.spPIDscale
          ) ;
// Dump to the screen if we are in debug mode
   if (debug) {
          printf("exit flag is %d\n", GUIvars.exitFlag) ;
          printf("sonarEna is %d\n", GUIvars.sonarEna) ;
          printf("lineEna is %d\n", GUIvars.lineEna) ;
          printf("rtcEna is %d\n", GUIvars.rtcEna) ;
          printf("accelEna is %d\n", GUIvars.accelEna) ;
          printf("motorType is %d\n", GUIvars.motorType) ;
          printf("Kp is %f\n", GUIvars.Kp) ;
          printf("Ki is %f\n", GUIvars.Ki) ;
          printf("Kd is %f\n", GUIvars.Kd) ;
          printf("samplePeriod is %f\n", GUIvars.samplePeriod) ;
          printf("wheelDiam is %f\n", GUIvars.wheelDiam) ;
          printf("turnRad is %f\n", GUIvars.turnRad) ;
          printf("ticsPerRev is %f\n", GUIvars.ticsPerRev) ;
          printf("M1Ena is %d\n", GUIvars.M1Ena) ;
          printf("M2Ena is %d\n", GUIvars.M2Ena) ;
          printf("M3Ena is %d\n", GUIvars.M3Ena) ;
          printf("M4Ena is %d\n", GUIvars.M4Ena) ;
          printf("PWMresMode is %d\n", GUIvars.PWMresMode) ;
          printf("Kp (setpoint) is %f\n", GUIvars.Kp_sp) ;
          printf("Ki (setpoint) is %f\n", GUIvars.Ki_sp) ;
          printf("Kd (setpoint) is %f\n", GUIvars.Kd_sp) ;
          printf("max_delta is %f\n", GUIvars.max_delta) ;
          printf("velPIDscale is %f\n", GUIvars.velPIDscale) ;
          printf("spPIDscale is %f\n", GUIvars.spPIDscale) ;
   } // end if
   return ;
} // getGUIvars()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to load robot paramters from a file
// rather than from the GUI
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void loadGuiVarsFromFile(char * str) {
   FILE  *fid ;
```

```
    fid = fopen("./robot.config", "r") ;
    fscanf(fid, "%s", str) ;
    if (debug) printf("robot.config string read -> %s\n", str) ;
    fclose(fid) ;
    return  ;
} // end loadGuiVarsFromFile()
// ########################################################
// Routine to configure PRUs
// ########################################################
void configPRU(void) {
    float  scr = 0.0 ;  // scratchpad
    if (debug) printf("In configPRU() \n") ;
// Tells us when to exit program from GUI mode
    shared_memory->exitFlag = GUIvars.exitFlag ;
// Not currently using delay
    shared_memory->delay = 0 ;
// PWM resolution
// Sample period is in ms
    switch (GUIvars.PWMresMode) {
       case BITS_IS_8:    shared_memory->PWMres = 255 ;
                          scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_8BIT) + 0.5 ;
                          break ;
       case BITS_IS_10:   shared_memory->PWMres = 1023 ;
                          scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_10BIT) + 0.5 ;
                          break ;
       case BITS_IS_12:   shared_memory->PWMres = 4095 ;
                          scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_12BIT) + 0.5 ;
                          break ;
    } // end switch
// Number of PWM clock cycles making up a PID sample period
    shared_memory->PWMclkCnt = (uint32_t) (scr) ;
// Either DC or Servo
    shared_memory->motorType = GUIvars.motorType ;
// Motor enables
    shared_memory->motorENA[M1] = GUIvars.M1Ena ;
    shared_memory->motorENA[M2] = GUIvars.M2Ena ;
    shared_memory->motorENA[M3] = GUIvars.M3Ena ;
    shared_memory->motorENA[M4] = GUIvars.M4Ena ;
// Wheel diameter and tics per inch
    shared_memory->wheelDiam = TOFIX(GUIvars.wheelDiam, Q) ;
    scr = GUIvars.ticsPerRev / (PI * GUIvars.wheelDiam) ;
    shared_memory->ticsPerInch = TOFIX(scr, Q) ;
// Initialize DC motor structures
// Multiply the Kp, Ki, Kd values by constant
// We do this so we can use sliders in the GUI
// which span to 0 to 100 range.
    float k ;
    k = GUIvars.velPIDscale ;
    int i ;
    for (i = 0; i < NUM_MOTORS; i++) {
        shared_memory->motor[i].Kp = TOFIX((k * GUIvars.Kp), Q) ;
        shared_memory->motor[i].Ki = TOFIX((k * GUIvars.Ki), Q) ;
        shared_memory->motor[i].Kd = TOFIX((k * GUIvars.Kd), Q) ;
        shared_memory->motor[i].PWMmax = shared_memory->PWMres ;
        shared_memory->motor[i].PWMmin = 0 ;
        shared_memory->motor[i].max_delta = (int32_t) GUIvars.max_delta ;
    } // end i loop
// Freeze the PRUs implementing motor control
    shared_memory->command.code = NOP ;
    shared_memory->command.status = IDLE ;
    shared_memory->state = 0 ;
// Initialize the PID loop which adjusts the setpoint
// to make sure both wheels traveling at the same velocity
```

```
    initSetpointPID() ;
    return ;
} // end configPRU()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to initialize the setpoint PID loop.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void initSetpointPID(void) {
// PID error signals
    shared_memory->setpointPID.e0 = 0 ;
    shared_memory->setpointPID.e1 = 0 ;
    shared_memory->setpointPID.e2 = 0 ;
// PID parameters
    float     k ;
    k = GUIvars.spPIDscale ;
    shared_memory->setpointPID.Kp = TOFIX(k * GUIvars.Kp_sp, Q) ;
    shared_memory->setpointPID.Ki = TOFIX(k * GUIvars.Ki_sp, Q) ;
    shared_memory->setpointPID.Kd = TOFIX(k * GUIvars.Kd_sp, Q) ;
// Limits
    shared_memory->setpointPID.minVal = 0 ;
    shared_memory->setpointPID.maxVal = 0 ;
    shared_memory->setpointPID.output = 0 ;
    return ;
} // end initSetpointPID()
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Dump of entire memory structure to a file
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void memoryDump(void) {
    FILE  *fid ;
    int   i ;
    if (debug) printf("Dumping contents of shared memory to file.\n") ;
    fid = fopen("memory_dump.txt", "w") ;
// Motor enable values
    for (i = 0; i < NUM_MOTORS; i++) {
        fprintf(fid, "mem->motorENA[%d] = %d\n", i+1, shared_memory->motorENA[i]) ;
    } // end i loop
// PWM and encoder values
    for (i = 0; i < NUM_MOTORS; i++) {
        fprintf(fid, "mem->pwm[%d] = %d\n", i+1, shared_memory->pwm[i]) ;
        fprintf(fid, "mem->enc[%d] = %d\n", i+1, shared_memory->enc[i]) ;
    } // end i loop
// Motor parameters
    for (i = 0; i < NUM_MOTORS; i++) {
        fprintf(fid, "mem->motor[%d].setpoint = %d\n", i+1, shared_memory->motor[i].
            setpoint) ;
        fprintf(fid, "mem->motor[%d].distance = %d\n", i+1, shared_memory->motor[i].
            distance) ;
        fprintf(fid, "mem->motor[%d].targetDistance = %d\n", i+1, shared_memory->motor[i
            ].targetDistance) ;
        fprintf(fid, "mem->motor[%d].wheelDirection = %d\n", i+1, shared_memory->motor[i
            ].wheelDirection) ;
        fprintf(fid, "mem->motor[%d].brakeType = %d\n", i+1, shared_memory->motor[i].
            brakeType) ;
        fprintf(fid, "mem->motor[%d].e0 = %d\n", i+1, shared_memory->motor[i].e0) ;
        fprintf(fid, "mem->motor[%d].e1 = %d\n", i+1, shared_memory->motor[i].e1) ;
        fprintf(fid, "mem->motor[%d].e2 = %d\n", i+1, shared_memory->motor[i].e2) ;
        fprintf(fid, "mem->motor[%d].Kp = %d\n", i+1, shared_memory->motor[i].Kp) ;
        fprintf(fid, "mem->motor[%d].Ki = %d\n", i+1, shared_memory->motor[i].Ki) ;
        fprintf(fid, "mem->motor[%d].Kd = %d\n", i+1, shared_memory->motor[i].Kd) ;
        fprintf(fid, "mem->motor[%d].PWMmin = %d\n", i+1, shared_memory->motor[i].PWMmin
            ) ;
        fprintf(fid, "mem->motor[%d].PWMmax = %d\n", i+1, shared_memory->motor[i].PWMmax
            ) ;
```

```
        fprintf(fid, "mem->motor[%d].PWMout = %d\n", i+1, shared_memory->motor[i].PWMout
            ) ;
        fprintf(fid, "mem->motor[%d].PWMout = %d\n", i+1, shared_memory->motor[i].
            max_delta) ;
    } // end i loop
// Other parameters
    fprintf(fid, "mem->wheelDiam = %d\n", shared_memory->wheelDiam) ;
    fprintf(fid, "mem->ticsperInch = %d\n", shared_memory->ticsPerInch) ;
    fprintf(fid, "mem->delay = %d\n", shared_memory->delay) ;
    fprintf(fid, "mem->state = %x\n", shared_memory->state) ;
    fprintf(fid, "mem->PWMclkCnt = %d\n", shared_memory->PWMclkCnt) ;
    fprintf(fid, "mem->PWMres = %d\n", shared_memory->PWMres) ;
    fprintf(fid, "mem->exitFlag = %d\n", shared_memory->exitFlag) ;
    fprintf(fid, "mem->motorType = %u\n", shared_memory->motorType) ;
    fprintf(fid, "mem->scr = %d\n", shared_memory->scr) ;
    fprintf(fid, "mem->interruptCounter = %u\n", shared_memory->interruptCounter) ;
    fprintf(fid, "mem->command.code = %u\n", shared_memory->command.code) ;
    fprintf(fid, "mem->command.status = %u\n", shared_memory->command.status) ;
// Data buffer
/*
    for (i=0; i<BUF_LEN; i++) {
       fprintf(fid, "mem->enc_data[%d] = %u\n", i, shared_memory->enc_data[i]) ;
    }
*/
// Close the file and exit
    fclose(fid) ;
    return ;
} // end memoryDump()
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Routine to convert a distance in inches to an
// equivalent number of encoder tics.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
int32_t inches2tics(float inches) {
    int32_t    tics ;
    int32_t    inches_Q ;
    inches_Q = TOFIX(inches, Q) ;
    tics = FMUL(shared_memory->ticsPerInch, inches_Q, 0) ;
    tics = FCONV(tics, twoQ, 0) ; // integer
    return tics ;
} // end inches2tics()
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Routine to convert a distance in tics to an
// equivalent number of inches
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
float tics2inches(int32_t tics) {
    float    inches ;
    inches = ((float) tics) / shared_memory->ticsPerInch ;
    return inches ;
} // end tics2inches()
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Routine to update a motor structure.
// Values are stored into motor structures.
// Target distance, setpoint, wheel direction and braking
// mode get written out to shared memory.
// The final setpoint is actually stored in targetSetpoint.
// The setpoint is always set to 0.  The move routine in
// PRU 0 will setpoint up to the target.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void updateMotor(int motor_num, int dir, int brakeType, float distance, float velocity)
     {
    int32_t    distInTics, velInTics ;
    float      delX ;
// delX is distance we would move in one sample period
```

```
// samplePeriod is in ms
   delX = velocity * GUIvars.samplePeriod * 0.001 ;
// Need the distance and the velocity that we desire
// converted to tics
   distInTics = inches2tics(distance) ;
   velInTics =  inches2tics(delX) ;
   shared_memory->motor[motor_num].targetDistance = distInTics ;
// Setpoints should be in Q6 format
   shared_memory->motor[motor_num].targetSetpoint = FCONV(velInTics, 0, Q) ;
   shared_memory->motor[motor_num].setpoint = FCONV(velInTics, 0, Q)  ;
   shared_memory->motor[motor_num].deltaSetpoint = 0 ;
// Wheel direction and braking mode
   shared_memory->motor[motor_num].wheelDirection = dir ;
   shared_memory->motor[motor_num].brakeType = brakeType ;
// Differential setpoint limits
   shared_memory->setpointPID.maxVal = FCONV(velInTics, 0, Q) ;
   shared_memory->setpointPID.minVal = -FCONV(velInTics, 0, Q) ;
   return ;
} // end updateMotor()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to query a motor structure for important values.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int32_t queryMotor(int motor_num, int item) {
   int32_t  value ;
   switch(item) {
      case  SETPOINT:    value = shared_memory->motor[motor_num].setpoint ;
                         break ;
      case  DISTANCE  :  value = shared_memory->motor[motor_num].distance ;
                         break ;
      case  TARGET_DIST: value = shared_memory->motor[motor_num].targetDistance ;
                         break ;
      case  WHEEL_DIR:   value = shared_memory->motor[motor_num].wheelDirection ;
                         break ;
      case  BRAKE_TYPE:  value = shared_memory->motor[motor_num].brakeType ;
                         break ;
      default:           value = CRASH;
                         break ;
   } // end switch
   return value ;
} // end updateMotor()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to query the command structure
// Makes it easy to determine status of command
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int32_t query(int item) {
   int32_t  value ;
   switch(item) {
      case  CMD:         value = shared_memory->command.code ;
                         break ;
      case  STATUS:      value = shared_memory->command.status ;
                         break ;
      default:           value = CRASH ;
                         break ;
   } // end switch
   return value ;
} // end updateMotor()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Wait for IDLE to be true
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void waitForIdle(void) {
   while (query(STATUS) != IDLE) { delay_ms(STATUS_DELAY) ; }
   return ;
} // end for waitForIdle()
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Wait for Complete to be true
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void waitForComplete(void) {
   while (query(STATUS) != COMPLETED) { delay_ms(STATUS_DELAY) ; }
   return ;
} // waitForComplete()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Reset the PRUs
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void resetPRU(void) {
   if (debug) printf("Entering resetPRU()\n") ;
   shared_memory->command.code = BRAKE_HARD ;
   shared_memory->command.status = START ;
   waitForComplete() ;
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   return ;
} // end resetPRU()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive robot forward
// a distance (in inches) with a given velocity
// (in inches per sec)
//
// Return a 1 if successfully started command
// Return a 0 if not successful
//
// Only implementing for two motors.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int fwd(float distance, float velocity) {
// Wait until we are idle
   if (debug) { printf("Waiting for IDLE in fwd().\n") ; }
   waitForIdle() ;
// Update motor structures with information about
// how forward command should be carried out
   updateMotor(M1, CW, HARD, distance, velocity) ;
   updateMotor(M2, CW, HARD, distance, velocity) ;
// Update command structure to indicate we desre to drive FWD
   shared_memory->command.code = FWD ;
   shared_memory->command.status = START ;
   if (debug) { printf("Waiting for COMPLETED in fwd().\n") ; }
   waitForComplete() ;
// After command is seen to complete then set
// command to a no-op and state as being idle
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;
   return PASS ;
} // end fwd()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive robot backward
// Only implementing for two motors.
// M1 is left motor
// M2 motor is right
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int  bwd(float distance, float velocity) {
// Wait until we are idle
   if (debug) { printf("Waiting for IDLE in bwd();\n") ; }
   waitForIdle() ;
// Update motor structures with information about
// how forward command should be carried out
   updateMotor(M1, CCW, HARD, distance, velocity) ;
   updateMotor(M2, CCW, HARD, distance, velocity) ;
```

```
// Update command structure to indicate we desre to drive BWD
   shared_memory->command.code = BWD ;
   shared_memory->command.status = START ;
   if (debug) { printf("Waiting for COMPLETED in bwd() ;\n") ; }
   waitForComplete() ;
// After command is seen to complete then set
// command to a no-op and state as being idle
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;
   return PASS ;
} // end bwd()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to spin robot in direction specified
// # of degrees at particular velocity.
// Only implementing for two motors.
// One motor gets driven CW and the other CCW.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int rotate(float degrees, float velocity, int direction) {
   float  distance ;
//
// Need to determin distance we need to travel
//
   distance = (PI / 180.0) * GUIvars.turnRad * degrees ;
// Wait until we are idle
   if (debug) { printf("Waiting for IDLE in rotate().\n") ; }
   waitForIdle() ;
// Update motor structures with information about
// how forward command should be carried out
   if (direction == CW) {
      updateMotor(M1, CW, HARD, distance, velocity) ;
      updateMotor(M2, CCW, HARD, distance, velocity) ;
   } else {
      updateMotor(M1, CCW, HARD, distance, velocity) ;
      updateMotor(M2, CW, HARD, distance, velocity) ;
   } // end if-then-else
// Update command structure to indicate we desre to ROT
   shared_memory->command.code = ROT ;
   shared_memory->command.status = START ;
   if (debug) { printf("Waiting for COMPLETED in rotate().\n") ; }
   waitForComplete() ;
// After command is seen to complete then set
// command to a no-op and state as being idle
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;
   return PASS ;
} // end rotate() ;
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive to make a right turn
// Just calling the rotate() routine.
// Only implementing for two motors.
// Turn at 6 in/sec
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int right(void) {
   if (rotate(90.0, 6.0, CW) == PASS) {
      return PASS ;
   } else {
      return FAIL ;
   } // end if-then-else
} // end right()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive to make a left turn
```

```
// Just calling the rotate() routine.
// Only implementing for two motors.
// Turn at 6 in/sec
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int left(void) {
   if (rotate(90.0, 6.0, CCW) == PASS) {
      return PASS ;
   } else {
      return FAIL ;
   } // end if-then-else
} // end left()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to apply the brake
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int applyBrake (void) {
   shared_memory->command.code = BRAKE_HARD ;
   shared_memory->command.status = START ;
   waitForComplete() ;
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   return PASS ;
} // end brake()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to release the brake
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int releaseBrake (void) {
   return PASS ;
} // end brake()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to take a test drive
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void testDrive(void) {
// Reset PRU .. hard brake
   resetPRU() ;
   fwd(48.0, 12.0) ;
   left() ;
   fwd(24.0, 12.0) ;
   left() ;
   fwd(48.0, 12.0) ;
   left() ;
   fwd(24.0, 12.0) ;
   left() ;
   fwd(48.0, 12.0) ;
   rotate(180.0, 6.0, CW) ;
   fwd(48.0, 12.0) ;
   rotate(180.0, 6.0, CW) ;
   fwd(48.0, 12.0) ;
   rotate(180.0, 6.0, CW) ;
   fwd(48.0, 12.0) ;
   rotate(180.0, 6.0, CW) ;
   return  ;
} // end test_drive()
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to test our robot
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void testRobot(void) {
   if (debug) printf("Entering testRobot()\n") ;
   testDrive() ;
   if (debug) printf("Leaving testRobot()\n") ;
   return ;
}
```

E.6 bbbLib.h

```
/*
//Filename: libBBB.h
//Version : 0.1
//
//Project : Argonne National Lab - Forest
//Author  : Gavin Strunk
//Contact : gavin.strunk@gmail.com
//Date    : 28 June 2013
//
//Description - This is the main header file for
//    the libBBB library.
//
//Revision History
// 0.1:  Wrote basic framework with function
//    prototypes and definitions. \GS
*/
/*
Copyright (C) 2013 Gavin Strunk
Permission is hereby granted, free of charge, to any person obtaining a copy of this
    software and associated documentation files (the "Software"), to deal in the
    Software without restriction, including without limitation the rights to use, copy,
    modify, merge, publish, distribute, sublicense, and/or sell copies of the Software,
    and to permit persons to whom the Software is furnished to do so, subject to the
    following conditions:
The above copyright notice and this permission notice shall be included in all copies or
     substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
    INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
    PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
    HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
     CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
    SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
#ifndef _libBBB_H_
#define _libBBB_H_
//Includes
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <errno.h>
#include <time.h>
//
// Set TTY to which ever UART you plan to use
//
#define TTY     "/dev/ttyO1"
// Define a UART structure
//Type definitions
typedef struct {
      struct termios u;
}UART;
//Definitions
#define OUT "out"
#define IN  "in"
#define ON  1
```

```
#define OFF 0
#define USR1   "usr1"
#define USR2   "usr2"
#define USR3   "usr3"
#define P8_13  "P8_13"
#define E   65
#define RS  27
#define D4  46
#define D5  47
#define D6  26
#define D7  44
#define AIN0   "/AIN0"
#define AIN1   "/AIN1"
#define AIN2   "/AIN2"
#define AIN3   "/AIN3"
#define AIN4   "/AIN4"
#define AIN5   "/AIN5"
#define AIN6   "/AIN6"
#define AIN7   "/AIN7"
//Device Tree Overlay
//int addOverlay(char *dtb, char *overname);
//USR Prototypes
int setUsrLedValue(char* led, int value);
//GPIO Prototypes
int initPin(int pinnum);
int setPinDirection(int pinnum, char* dir);
int setPinValue(int pinnum, int value);
int getPinValue(int pinnum);
//PWM Prototypes
int initPWM(int mgrnum, char* pin);
int setPWMPeriod(int helpnum, char* pin, int period);
int setPWMDuty(int helpnum, char* pin, int duty);
int setPWMOnOff(int helpnum, char* pin, int run);
//UART Prototypes
//int initUART(int mgrnum, char* uartnum);
int initUART();
void closeUART(int fd);
int configUART(UART u, int property, char* value);
int txUART(int uart, unsigned char data);
unsigned char rxUART(int uart);
int UARTputstr(int uart,  char* buf);
int UARTgetstr(int uart,  char* buf);
//I2C Prototypes
int i2c_open(unsigned char bus, unsigned char addr);
int i2c_write(int handle, unsigned char* buf, unsigned int length);
int i2c_read(int handle, unsigned char* buf, unsigned int length);
int i2c_write_read(int handle,
                   unsigned char addr_w, unsigned char *buf_w, unsigned int len_w,
                   unsigned char addr_r, unsigned char *buf_r, unsigned int len_r);
int i2c_write_ignore_nack(int handle,
                          unsigned char addr_w, unsigned char* buf, unsigned int length)
                              ;
int i2c_read_no_ack(int handle,
                    unsigned char addr_r, unsigned char* buf, unsigned int length);
int i2c_write_byte(int handle, unsigned char val);
int i2c_read_byte(int handle, unsigned char* val);
int i2c_close(int handle);
// Provides an inaccurate delay (may be useful for waiting for ADC etc).
// The maximum delay is 999msec
int delay_ms(unsigned int msec);
//SPI Prototypes
int initSPI(int modnum);
void closeSPI(int device);
```

```
int writeByteSPI(int device,unsigned char *data);
int writeBufferSPI(int device, unsigned char *buf, int len);
int readByteSPI(int device, unsigned char *data);
int readBufferSPI(int device, int numbytes, unsigned char *buf);
//LCD 4-bit Prototypes
int initLCD();
int writeChar(unsigned char data);
int writeCMD(unsigned char cmd);
int writeString(char* str, int len);
int LCD_ClearScreen();
int LCD_Home();
int LCD_CR();
int LCD_Backspace();
int LCD_Move(int location);
//ADC Prototypes
int initADC(int mgrnum);
int readADC(int helpnum, char* ach);
//Time Prototypes
void pauseSec(int sec);
int  pauseNanoSec(long nano);
#endif
```

E.7 bbbLib.c

```
/*
//Filename: libBBB.c
//Version : 0.1
//
//Project : Argonne National Lab - Forest
//Author  : Gavin Strunk
//Contact : gavin.strunk@gmail.com
//Date    : 28 June 2013
//
//Description - This is the main library file that
//    eases the interface to the BeagleBone
//    Black. It includes functions for GPIO,
//    UART, I2C, SPI, ADC, Timing, and Overlays.
//
//Revision History
// 0.1:  Wrote the basic framework for all the
//    functions. \GS
*/
/*
Copyright (C) 2013 Gavin Strunk
Permission is hereby granted, free of charge, to any person obtaining a copy of this
    software and associated documentation files (the "Software"), to deal in the
    Software without restriction, including without limitation the rights to use, copy,
    modify, merge, publish, distribute, sublicense, and/or sell copies of the Software,
    and to permit persons to whom the Software is furnished to do so, subject to the
    following conditions:
The above copyright notice and this permission notice shall be included in all copies or
     substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
    INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
    PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
    HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
     CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
    SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
#include "bbbLib.h"
//Local functions not used by outside world
void initCMD(unsigned char cmd);
//*************************************************
//*                USR FUNCTIONS                  *
//*************************************************
int setUsrLedValue(char* led, int value)
{
   FILE *usr;
   char buf[20];
   char buf2[50] = "/sys/class/leds/beaglebone:green:";
   //build file path to usr led brightness
   sprintf(buf,"%s",led);
   strcat(buf2,strcat(buf,"/brightness"));
   usr = fopen(buf2, "w");
   if(usr == NULL) printf("USR Led failed to open\n");
   fseek(usr,0,SEEK_SET);
   fprintf(usr,"%d",value);
   fflush(usr);
   fclose(usr);
   return 0;
}
//*************************************************
//*               GPIO FUNCTIONS                  *
//*************************************************
int initPin(int pinnum)
```

```
{
   FILE *io;
   io = fopen("/sys/class/gpio/export", "w");
   if(io == NULL) printf("Pin failed to initialize\n");
   fseek(io,0,SEEK_SET);
   fprintf(io,"%d",pinnum);
   fflush(io);
   fclose(io);
   return 0;
}
int setPinDirection(int pinnum, char* dir)
{
   FILE *pdir;
   char buf[10];
   char buf2[50] = "/sys/class/gpio/gpio";
   //build file path to the direction file
   sprintf(buf,"%i",pinnum);
   strcat(buf2,strcat(buf,"/direction"));
   pdir = fopen(buf2, "w");
   if(pdir == NULL) printf("Direction failed to open\n");
   fseek(pdir,0,SEEK_SET);
   fprintf(pdir,"%s",dir);
   fflush(pdir);
   fclose(pdir);
   return 0;
}
int setPinValue(int pinnum, int value)
{
   FILE *val;
   char buf[5];
   char buf2[50] = "/sys/class/gpio/gpio";
   //build path to value file
   sprintf(buf,"%i",pinnum);
   strcat(buf2,strcat(buf,"/value"));
   val = fopen(buf2, "w");
   if(val == NULL) printf("Value failed to open\n");
   fseek(val,0,SEEK_SET);
   fprintf(val,"%d",value);
   fflush(val);
   fclose(val);
   return 0;
}
int getPinValue(int pinnum)
{
   FILE *val;
   int value;
   char buf[5];
   char buf2[50] = "/sys/class/gpio/gpio";
   //build file path to value file
   sprintf(buf,"%i",pinnum);
   strcat(buf2,strcat(buf,"/value"));
   val = fopen(buf2, "r");
   if(val == NULL) printf("Input value failed to open\n");
   fseek(val,0,SEEK_SET);
   fscanf(val,"%d",&value);
   fclose(val);
   return value;
}
//***********************************************
//*                PWM FUNCTIONS                *
//***********************************************
int initPWM(int mgrnum, char* pin)
{
```

```
    FILE *pwm;
    char buf[5];
    char buf2[50] = "/sys/devices/bone_capemgr.";
    char buf3[20] = "bone_pwm_";
    //build file paths
    sprintf(buf,"%i",mgrnum);
    strcat(buf2,strcat(buf,"/slots"));
    strcat(buf3,pin);
    pwm = fopen(buf2, "w");
    if(pwm == NULL) printf("PWM failed to initialize\n");
    fseek(pwm,0,SEEK_SET);
    fprintf(pwm,"am33xx_pwm");
    fflush(pwm);
    fprintf(pwm,"%s",buf3);
    fflush(pwm);
    fclose(pwm);
    return 0;
}
int setPWMPeriod(int helpnum, char* pin, int period)
{
    FILE *pwm;
    char buf[5];
    char buf2[60] = "/sys/devices/ocp.2/pwm_test_";
    //build file path
    sprintf(buf,"%i",helpnum);
    printf("%s\n",pin);
    strcat(buf2,pin);
    strcat(buf2,".");
    strcat(buf2,strcat(buf,"/period"));
    printf("%s\n",buf2);
    pwm = fopen(buf2, "w");
    if(pwm == NULL) printf("PWM Period failed to open\n");
    fseek(pwm,0,SEEK_SET);
    fprintf(pwm,"%d",period);
    fflush(pwm);
    fclose(pwm);
    return 0;
}
int setPWMDuty(int helpnum, char* pin, int duty)
{
    FILE *pwm;
    char buf[5];
    char buf2[50] = "/sys/devices/ocp.2/pwm_test_";
    //build file path
    sprintf(buf,"%i",helpnum);
    strcat(buf2,pin);
    strcat(buf2,".");
    strcat(buf2,strcat(buf,"/duty"));
    pwm = fopen(buf2, "w");
    if(pwm == NULL) printf("PWM Duty failed to open\n");
    fseek(pwm,0,SEEK_SET);
    fprintf(pwm,"%d",duty);
    fflush(pwm);
    fclose(pwm);
    return 0;
}
int setPWMOnOff(int helpnum, char* pin, int run)
{
    FILE *pwm;
    char buf[5];
    char buf2[50] = "/sys/devices/ocp.2/pwm_test_";
    //build file path
    sprintf(buf,"%i",helpnum);
```

```
    strcat(buf2,pin);
    strcat(buf2,".");
    strcat(buf2,strcat(buf,"/run"));
    pwm = fopen(buf2, "w");
    if(pwm == NULL) printf("PWM Run failed to open\n");
    fseek(pwm,0,SEEK_SET);
    fprintf(pwm,"%d",run);
    fflush(pwm);
    fclose(pwm);
    return 0;
}
//*************************************************
//*               UART FUNCTIONS                  *
//*************************************************
int initUART()
{
    //return the int reference to the port
    struct termios old;
    struct termios new;
    int fd;
    fd = open(TTY, O_RDWR | O_NOCTTY);
    if(fd < 0)
    {
        printf("Port failed to open\n");
        return fd;
    }
    tcgetattr(fd,&old);
    bzero(&new, sizeof(new));
    new.c_cflag = B4800 | CS8 | CLOCAL | CREAD;
    new.c_iflag = IGNPAR | ICRNL;
    new.c_oflag = 0;
    new.c_lflag = 0;
    new.c_cc[VTIME] = 0;
    new.c_cc[VMIN]  = 1;
    //clean the line and set the attributes
    tcflush(fd,TCIFLUSH);
    tcsetattr(fd,TCSANOW,&new);
    return fd;
}
void closeUART(int fd)
{
    close(fd);
}
int configUART(UART u, int property, char* value)
{
    //This is used to set the configuration values
    //for the uart module
    return 0;
}
int txUART(int uart, unsigned char data)
{
    //write a single byte
    write(uart,&data,1);
    tcdrain(uart);
    return 0;
}
unsigned char rxUART(int uart)
{
    //read in a single byte
    unsigned char data;
    read(uart,&data,1);
    return data;
}
```

```
int UARTputstr(int uart,  char* buf)
{
   int i;
   for(i=0; i < strlen(buf); i++)
      txUART(uart,buf[i]);
   return 0;
}
int UARTgetstr(int uart,  char* buf)
{
    int i ;
    i = 0 ;
   while (1) {
        buf[i] = rxUART(uart) ;
        if (buf[i] == '\n') break ;
        i += 1 ;
    }
    i += 1 ;
    buf[i] = '\x0'  ;
    return 0 ;
}
//************************************************
//*                I2C FUNCTIONS                 *
//************************************************
// Returns a handle for i2c device at "addr" on bus "bus"
int i2c_open(unsigned char bus, unsigned char addr)
{
  int file;
  char filename[16];
  sprintf(filename,"/dev/i2c-%d", bus);
  if ((file = open(filename,O_RDWR)) < 0)
  {
    fprintf(stderr, "i2c_open open error: %s\n", strerror(errno));
    return(file);
  }
  if (ioctl(file,I2C_SLAVE,addr) < 0)
  {
    fprintf(stderr, "i2c_open ioctl error: %s\n", strerror(errno));
    return(-1);
  }
  return(file);
}
// Write out a data buffer to i2c device
int i2c_write(int handle, unsigned char* buf, unsigned int length)
{
  if (write(handle, buf, length) != length)
  {
    fprintf(stderr, "i2c_write error: %s\n", strerror(errno));
    return(-1);
  }
  return(length);
}
//  Write out a single byte to i2c device
int i2c_write_byte(int handle, unsigned char val)
{
  if (write(handle, &val, 1) != 1)
  {
    fprintf(stderr, "i2c_write_byte error: %s\n", strerror(errno));
    return(-1);
  }
  return(1);
}
// Read a specified number of bytes from i2c device
int i2c_read(int handle, unsigned char* buf, unsigned int length)
```

```
{
  if (read(handle, buf, length) != length)
  {
    fprintf(stderr, "i2c_read error: %s\n", strerror(errno));
    return(-1);
  }
  return(length);
}
// Read a single byte from the device
int i2c_read_byte(int handle, unsigned char* val)
{
  if (read(handle, val, 1) != 1)
  {
    fprintf(stderr, "i2c_read_byte error: %s\n", strerror(errno));
    return(-1);
  }
  return(1);
}
// Close the handle to the device
int i2c_close(int handle)
{
  if ((close(handle)) != 0)
  {
    fprintf(stderr, "i2c_close error: %s\n", strerror(errno));
    return(-1);
  }
  return(0);
}
// Write and read
int i2c_write_read(int handle,
                   unsigned char addr_w, unsigned char *buf_w, unsigned int len_w,
                   unsigned char addr_r, unsigned char *buf_r, unsigned int len_r)
{
   struct i2c_rdwr_ioctl_data msgset;
   struct i2c_msg msgs[2];
   msgs[0].addr=addr_w;
   msgs[0].len=len_w;
   msgs[0].flags=0;
   msgs[0].buf=buf_w;
   msgs[1].addr=addr_r;
   msgs[1].len=len_r;
   msgs[1].flags=1;
   msgs[1].buf=buf_r;
   msgset.nmsgs=2;
   msgset.msgs=msgs;
   if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
      fprintf(stderr, "i2c_write_read error: %s\n",strerror(errno));
    return -1;
  }
  return(len_r);
}
// Write and ignore NACK
int i2c_write_ignore_nack(int handle,
                          unsigned char addr_w, unsigned char* buf, unsigned int length)
{
   struct i2c_rdwr_ioctl_data msgset;
   struct i2c_msg msgs[1];
   msgs[0].addr=addr_w;
   msgs[0].len=length;
   msgs[0].flags=0 | I2C_M_IGNORE_NAK;
   msgs[0].buf=buf;
   msgset.nmsgs=1;
```

```
   msgset.msgs=msgs;
   if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
      fprintf(stderr, "i2c_write_ignore_nack error: %s\n",strerror(errno));
    return -1;
  }
  return(length);
}
// Read and ignore no ACK
int i2c_read_no_ack(int handle,
                    unsigned char addr_r, unsigned char* buf, unsigned int length)
{
   struct i2c_rdwr_ioctl_data msgset;
   struct i2c_msg msgs[1];
   msgs[0].addr=addr_r;
   msgs[0].len=length;
   msgs[0].flags=I2C_M_RD | I2C_M_NO_RD_ACK;
   msgs[0].buf=buf;
   msgset.nmsgs=1;
   msgset.msgs=msgs;
   if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
      fprintf(stderr, "i2c_read_no_ack error: %s\n",strerror(errno));
    return -1;
  }
  return(length);
}
// Delay for specified number of msec
int delay_ms(unsigned int msec)
{
  int ret;
  struct timespec a;
  if (msec>999)
  {
    fprintf(stderr, "delay_ms error: delay value needs to be less than 999\n");
    msec=999;
  }
  a.tv_nsec=((long)(msec))*1E6d;
  a.tv_sec=0;
  if ((ret = nanosleep(&a, NULL)) != 0)
  {
    fprintf(stderr, "delay_ms error: %s\n", strerror(errno));
  }
  return(0);
}
//************************************************
//*                LCD FUNCTIONS                 *
//************************************************
/*NOTE: DO NOT directly include libBBB.h for LCD functions!
 *    Instead include libLCD.h as this implements the
 *    screen control and full initialization.
*/
int initLCD()
{
   //initialize the pins
   initPin(RS);
   initPin(E);
   initPin(D4);
   initPin(D5);
   initPin(D6);
   initPin(D7);
   //set direction
   setPinDirection(RS,OUT);
```

```
    setPinDirection(E,OUT);
    setPinDirection(D4,OUT);
    setPinDirection(D5,OUT);
    setPinDirection(D6,OUT);
    setPinDirection(D7,OUT);
    setPinValue(E,OFF);
    //initialize the screen
    pauseNanoSec(1500000);
    initCMD(0x30);
    pauseNanoSec(5000000);
    initCMD(0x30);
    pauseNanoSec(5000000);
    initCMD(0x30);
    pauseNanoSec(5000000);
    initCMD(0x20);
    pauseNanoSec(5000000);
    writeCMD(0x2C);
    pauseNanoSec(5000000);
    writeCMD(0x08);
    pauseNanoSec(5000000);
    writeCMD(0x01);
    pauseNanoSec(2000000);
    writeCMD(0x06);
    pauseNanoSec(5000000);
    writeCMD(0x0E);
    pauseNanoSec(5000000);
    return 0;
}
void initCMD(unsigned char cmd)
{
    //bring rs low for command
    setPinValue(RS,OFF);
    pauseNanoSec(500000);
    //send the highest nibble only
    setPinValue(E,ON);
    setPinValue(D7,((cmd >> 7) & 1));
    setPinValue(D6,((cmd >> 6) & 1));
    setPinValue(D5,((cmd >> 5) & 1));
    setPinValue(D4,((cmd >> 4) & 1));
    pauseNanoSec(500000);
    setPinValue(E,OFF);
    pauseNanoSec(500000);
}
int writeChar(unsigned char data)
{
    //bring rs high for character
    pauseNanoSec(500000);
    setPinValue(RS,ON);
    pauseNanoSec(500000);
    //send highest nibble first
    setPinValue(E,ON);
    setPinValue(D7, ((data >> 7) & 1));
    setPinValue(D6, ((data >> 6) & 1));
    setPinValue(D5, ((data >> 5) & 1));
    setPinValue(D4, ((data >> 4) & 1));
    pauseNanoSec(500000);
    setPinValue(E,OFF);
    pauseNanoSec(500000);
    //send the low nibble
    setPinValue(E,ON);
    setPinValue(D7, ((data >> 3) & 1));
    setPinValue(D6, ((data >> 2) & 1));
    setPinValue(D5, ((data >> 1) & 1));
```

```
    setPinValue(D4, (data & 1));
    pauseNanoSec(500000);
    setPinValue(E,OFF);
    pauseNanoSec(500000);
    return 0;
}
int writeCMD(unsigned char cmd)
{
    //bring rs low for command
    setPinValue(RS, OFF);
    pauseNanoSec(500000);
    //send highest nibble first
    setPinValue(E,ON);
    setPinValue(D7, ((cmd >> 7) & 1));
    setPinValue(D6, ((cmd >> 6) & 1));
    setPinValue(D5, ((cmd >> 5) & 1));
    setPinValue(D4, ((cmd >> 4) & 1));
    pauseNanoSec(500000);
    setPinValue(E,OFF);
    pauseNanoSec(500000);
    //send the low nibble
    setPinValue(E,ON);
    setPinValue(D7, ((cmd >> 3) & 1));
    setPinValue(D6, ((cmd >> 2) & 1));
    setPinValue(D5, ((cmd >> 1) & 1));
    setPinValue(D4, (cmd & 1));
    pauseNanoSec(500000);
    setPinValue(E, OFF);
    pauseNanoSec(500000);
    return 0;
}
//************************************************
//*                ADC FUNCTIONS                 *
//************************************************
int initADC(int mgrnum)
{
    FILE *ain;
    char buf[5];
    char buf2[50] = "/sys/devices/bone_capemgr.";
    //build path to setup ain
    sprintf(buf,"%i",mgrnum);
    strcat(buf2,strcat(buf,"/slots"));
    ain = fopen(buf2, "w");
    if(ain == NULL) printf("Analog failed load\n");
    fseek(ain,0,SEEK_SET);
    fprintf(ain,"cape-bone-iio");
    fflush(ain);
    fclose(ain);
    return 0;
}
int readADC(int helpnum, char* ach)
{
    FILE *aval;
    int value;
    char buf[5];
    char buf2[50] = "/sys/devices/ocp.2/helper.";
    //build file path to read adc
    sprintf(buf,"%i",helpnum);
    strcat(buf2,strcat(buf,ach));
    aval = fopen(buf2, "r");
    if(aval == NULL) printf("Analog failed to open\n");
    fseek(aval,0,SEEK_SET);
    fscanf(aval,"%d",&value);
```

```
   fflush(aval);
   fclose(aval);
   return value;
}
//*****************************************
//*            TIME FUNCTIONS             *
//*****************************************
void pauseSec(int sec)
{
   time_t now,later;
   now = time(NULL);
   later = time(NULL);
   while((later - now) < (double)sec)
      later = time(NULL);
}
int pauseNanoSec(long nano)
{
   struct timespec tmr1,tmr2;
   //assume you are not trying to pause more than 1s
   tmr1.tv_sec = 0;
   tmr1.tv_nsec = nano;
   if(nanosleep(&tmr1, &tmr2) < 0)
   {
      printf("Nano second pause failed\n");
      return -1;
   }
   return 0;
}
```

APPENDIX F

GUI Tcl/Tk Code

```
#!/usr/bin/env tclsh
package require Tk
# Process to report state of all variables
proc reportState { } {
   global   exitFlag
   global   sonarEna
   global   lineEna
   global   rtcEna
   global   accelEna
   global   motorType
   global   Kp
   global   Ki
   global   Kd
   global   samplePeriod
   global   wheelDiam
   global   turnRad
   global   ticsPerRev
   global   M1Ena
   global   M2Ena
   global   M3Ena
   global   M4Ena
   global   PWMresMode
   global   Kp_sp
   global   Ki_sp
   global   Kd_sp
   global   max_delta
   global   velPIDscale
   global   spPIDscale
   puts -nonewline "$exitFlag:"
   puts -nonewline "$sonarEna:"
   puts -nonewline "$lineEna:"
   puts -nonewline "$rtcEna:"
   puts -nonewline "$accelEna:"
   puts -nonewline "$motorType:"
   puts -nonewline "$Kp:"
   puts -nonewline "$Ki:"
   puts -nonewline "$Kd:"
   puts -nonewline "$samplePeriod:"
   puts -nonewline "$wheelDiam:"
   puts -nonewline "$turnRad:"
   puts -nonewline "$ticsPerRev:"
   puts -nonewline "$M1Ena:"
   puts -nonewline "$M2Ena:"
   puts -nonewline "$M3Ena:"
   puts -nonewline "$M4Ena:"
   puts -nonewline "$PWMresMode:"
   puts -nonewline "$Kp_sp:"
   puts -nonewline "$Ki_sp:"
   puts -nonewline "$Kd_sp:"
   puts -nonewline "$max_delta:"
   puts -nonewline "$velPIDscale:"
   puts -nonewline "$spPIDscale"
   puts ""
}
# Procedure to toggle color of "apply" button
proc buttonColor {{x 0}} {
```

```
  global applyPressed
  if {$applyPressed} {
#     puts "Apply pressed"
     set  applyPressed 0
     .b_run config -bg green -activebackground green
  } else {
     .b_run config -bg red -activebackground red
  }
}
set   cnt    0
set   cmd    buttonColor
# Flags
set   applyPressed    0
set   exitFlag        0
# Chose a font
set    font   {Arial 14}
set    bw      5
# Give the window a title
wm title . "SIUE IEEE Robot Project"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Describe the header area
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
frame .headerarea -borderwidth 5 -relief groove
pack .headerarea -fill x
label .l_title   -text  "SIUE IEEE Robot" -font "Arial 36"
pack  .l_title -in .headerarea
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Describe the button area
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# ^^^^^^^^^^^^^^^^  Features Enabled ^^^^^^^^^^^^^^^^^^^^^^^^
frame .buttonarea -borderwidth $bw -height 500
pack .buttonarea -fill x
label .l_feature_ena  -text "Feature enabled:  " -font $font
set sonarEna  1
set lineEna   1
set rtcEna    1
set accelEna  1
checkbutton .b_sonar   -text "   Sonars    " -variable sonarEna -justify right
    -background yellow \
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_line    -text "   Servos    " -variable lineEna -justify right
    -background yellow\
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_rtc     -text "     RTC     " -variable rtcEna -justify right -background
     yellow \
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_accel   -text " Acclerometer " -variable accelEna -justify right
    -background yellow \
        -font $font -fg black -relief raised -height 1 \
        -command $cmd
grid config .l_feature_ena   -in .buttonarea  -column 0 -row 1 -sticky "w"
grid config .b_sonar         -in .buttonarea  -column 1 -row 1 -sticky "e"
grid config .b_line          -in .buttonarea  -column 2 -row 1 -sticky "e"
grid config .b_rtc           -in .buttonarea  -column 3 -row 1 -sticky "e"
grid config .b_accel         -in .buttonarea  -column 4 -row 1 -sticky "e"
#  ^^^^^^^^^^^^^^^^^^^^^^^^^ Motor Type (DC or Stepper) ^^^^^^^^^^^^^^^^^^
set motorType  1
radiobutton .b_dc -variable motorType \
          -text "      DC     "  -value 1 -justify right -background white \
          -font $font -fg black -relief raised -height 1 \
```

```
          -command $cmd
radiobutton .b_stepper -variable motorType \
          -text "   Stepper   " -value 2 -justify right -background white \
          -font $font -fg black -relief raised -height 1 \
          -command $cmd
label .l_motor_type   -text "Select motor type:  " -font $font
grid config .l_motor_type   -in .buttonarea  -column 0 -row 2 -sticky "w"
grid config .b_dc           -in .buttonarea  -column 1 -row 2 -sticky "e"
grid config .b_stepper      -in .buttonarea  -column 2 -row 2 -sticky "e"
#  ^^^^^^^^^^^^^^^^^^^^^^^^^ PWM resolution ^^^^^^^^^^^^^^^^^^
# default is 8 bit
set PWMresMode  1
label .l_PWMres_mode   -text  "Select PWM resolution:  " -font $font
radiobutton .b_8bit -variable PWMresMode \
          -text "   8 bits    "  -value 1 -justify right -background violet \
          -font $font -fg black -relief raised -height 1 \
          -command $cmd
radiobutton .b_10bit -variable PWMresMode \
          -text "  10 bits    " -value 2 -justify right -background violet \
          -font $font -fg black -relief raised -height 1 \
          -command $cmd
radiobutton .b_12bit -variable PWMresMode \
          -text "  12 bits    " -value 3 -justify right -background violet \
          -font $font -fg black -relief raised -height 1 \
          -command $cmd
grid config .l_PWMres_mode  -in .buttonarea  -column 0 -row 3 -sticky "w"
grid config .b_8bit         -in .buttonarea  -column 1 -row 3 -sticky "e"
grid config .b_10bit        -in .buttonarea  -column 2 -row 3 -sticky "e"
grid config .b_12bit        -in .buttonarea  -column 3 -row 3 -sticky "e"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Motors Enabled ^^^^^^^^^^^^^^^^^^^^^^^
label .l_motor_ena  -text "Motors enabled:  " -font $font
set M1Ena  1
set M2Ena  1
set M3Ena  0
set M4Ena  0
checkbutton .b_M1   -text  "      M1      " -variable M1Ena -justify right -background
    yellow \
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_M2    -text "     M2      " -variable M2Ena -justify right -background
    yellow \
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_M3     -text "    M3      " -variable M3Ena -justify right -background
    yellow \
         -font $font -fg black -relief raised -height 1 \
         -command $cmd
checkbutton .b_M4     -text "    M4      " -variable M4Ena -justify right -background
    yellow \
        -font $font -fg black -relief raised -height 1 \
        -command $cmd
grid config .l_motor_ena     -in .buttonarea  -column 0 -row 4 -sticky "w"
grid config .b_M1            -in .buttonarea  -column 1 -row 4 -sticky "e"
grid config .b_M2            -in .buttonarea  -column 2 -row 4 -sticky "e"
grid config .b_M3            -in .buttonarea  -column 3 -row 4 -sticky "e"
grid config .b_M4            -in .buttonarea  -column 4 -row 4 -sticky "e"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Describe the velocity PID slider area
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
frame .sliderarea_vel_pid  -borderwidth $bw -relief ridge
pack .sliderarea_vel_pid  -fill x -expand true
set Kp  40
set Ki  20
```

```
set Kd  0
label .l_Kp -text "Velocity PID Kp parameter          " -font $font
scale .s_Kp -from 0 -to 100 -length 500 -variable Kp \
       -tickinterval 20 -orient horizontal -troughcolor red \
       -label "                  PID Control" -font $font \
       -showvalue false -command $cmd
label .l_Ki -text "Velocity PID Ki parameter:         " -font $font
scale .s_Ki -from 0 -to 100 -length 500  -variable Ki \
       -tickinterval 20 -orient horizontal -troughcolor green \
       -font $font -showvalue false -command $cmd
label .l_Kd -text "Velocity PID Kd parameter:         " -font $font
scale .s_Kd -from 0 -to 100 -length 500 -variable Kd \
       -tickinterval 20 -orient horizontal -troughcolor blue \
       -font $font -showvalue false  -command $cmd
grid config .l_Kp -in .sliderarea_vel_pid -column 0 -row 1 -sticky "w"
grid config .s_Kp -in .sliderarea_vel_pid -column 1 -row 1 -sticky "e"
grid config .l_Ki -in .sliderarea_vel_pid -column 0 -row 2 -sticky "w"
grid config .s_Ki -in .sliderarea_vel_pid -column 1 -row 2 -sticky "e"
grid config .l_Kd -in .sliderarea_vel_pid -column 0 -row 3 -sticky "w"
grid config .s_Kd -in .sliderarea_vel_pid -column 1 -row 3 -sticky "e"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Describe the setpoint PID slider area
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
frame .sliderarea_sp_pid  -borderwidth $bw -relief ridge
pack .sliderarea_sp_pid  -fill x -expand true
set Kp_sp  8
set Ki_sp  12
set Kd_sp  0.0
label .l_Kp_sp -text "Setpoint PID Kp parameter          " -font $font
scale .s_Kp_sp -from 0 -to 100 -length 500 -variable Kp_sp \
       -tickinterval 20 -orient horizontal -troughcolor red \
       -label "               Setpoint PID Control" -font $font \
       -showvalue false -command $cmd
label .l_Ki_sp -text "Setpoint PID Ki parameter:         " -font $font
scale .s_Ki_sp -from 0 -to 100 -length 500  -variable Ki_sp \
       -tickinterval 20 -orient horizontal -troughcolor green \
       -font $font -showvalue false -command $cmd
label .l_Kd_sp -text "Setpoint PID Kd parameter:         " -font $font
scale .s_Kd_sp -from 0 -to 100 -length 500 -variable Kd_sp \
       -tickinterval 20 -orient horizontal -troughcolor blue \
       -font $font -showvalue false  -command $cmd
grid config .l_Kp_sp -in .sliderarea_sp_pid -column 0 -row 1 -sticky "w"
grid config .s_Kp_sp -in .sliderarea_sp_pid -column 1 -row 1 -sticky "e"
grid config .l_Ki_sp -in .sliderarea_sp_pid -column 0 -row 2 -sticky "w"
grid config .s_Ki_sp -in .sliderarea_sp_pid -column 1 -row 2 -sticky "e"
grid config .l_Kd_sp -in .sliderarea_sp_pid -column 0 -row 3 -sticky "w"
grid config .s_Kd_sp -in .sliderarea_sp_pid -column 1 -row 3 -sticky "e"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Describe Text area where we can enter sine and corner freq
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
frame .textarea -borderwidth 5 -relief groove
pack .textarea -fill x -expand true
label .l_sample_period   -text  "PID sampling period (ms):  " -font $font
label .l_wheel_diam      -text  "Wheel diameter (in):  " -font $font
label .l_turn_rad        -text  "Turn radius (in):  " -font $font
label .l_tics_per_rev    -text  "Tics per revolution:  " -font $font
label .l_max_delta       -text  "Maximum delta in velocity PID loop:  " -font $font
label .l_velPIDscale     -text  "Scale factor for velocity PID parameters:  " -font
    $font
label .l_spPIDscale      -text  "Scale factor for setpoint PID parameters:  " -font
    $font
# Sample period in ms
set samplePeriod    10
```

```
set wheelDiam       2.625
set turnRad         3.51
set ticsPerRev      2400
set max_delta       50
set velPIDscale     1.0
set spPIDscale      0.01
entry .e_sample_period  -textvariable   samplePeriod  -width 15 -font $font
entry .e_wheel_diam     -textvariable   wheelDiam     -width 15 -font $font
entry .e_turn_rad       -textvariable   turnRad       -width 15 -font $font
entry .e_tics_per_rev   -textvariable   ticsPerRev    -width 15 -font $font
entry .e_max_delta      -textvariable   max_delta     -width 15 -font $font
entry .e_velPIDscale    -textvariable   velPIDscale   -width 15 -font $font
entry .e_spPIDscale     -textvariable   spPIDscale    -width 15 -font $font
bind .e_sample_period    <Button-1>     $cmd
bind .e_wheel_diam       <Button-1>     $cmd
bind .e_turn_rad         <Button-1>     $cmd
bind .e_tics_per_rev     <Button-1>     $cmd
bind .e_max_delta        <Button-1>     $cmd
bind .e_velPIDscale      <Button-1>     $cmd
bind .e_spPIDscale       <Button-1>     $cmd
grid config .l_sample_period  -in .textarea  -column 0 -row 0 -sticky "w"
grid config .l_wheel_diam     -in .textarea  -column 0 -row 1 -sticky "w"
grid config .l_turn_rad       -in .textarea  -column 0 -row 2 -sticky "w"
grid config .l_tics_per_rev   -in .textarea  -column 0 -row 3 -sticky "w"
grid config .l_max_delta      -in .textarea  -column 0 -row 4 -sticky "w"
grid config .l_velPIDscale    -in .textarea  -column 0 -row 5 -sticky "w"
grid config .l_spPIDscale     -in .textarea  -column 0 -row 6 -sticky "w"
grid config .e_sample_period  -in .textarea  -column 1 -row 0 -sticky "e"
grid config .e_wheel_diam     -in .textarea  -column 1 -row 1 -sticky "e"
grid config .e_turn_rad       -in .textarea  -column 1 -row 2 -sticky "e"
grid config .e_tics_per_rev   -in .textarea  -column 1 -row 3 -sticky "e"
grid config .e_max_delta      -in .textarea  -column 1 -row 4 -sticky "e"
grid config .e_velPIDscale    -in .textarea  -column 1 -row 5 -sticky "e"
grid config .e_spPIDscale     -in .textarea  -column 1 -row 6 -sticky "e"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Need a "Apply" and "Exit" button
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
frame .footerarea -borderwidth 5 -relief groove
pack  .footerarea
button .b_run -text "Apply" -font $font -fg black -relief raised -height 1 \
       -command {reportState ; set applyPressed 1 ; $cmd }
pack .b_run -in .footerarea  -side left
.b_run config -bg red
button .b_exit -text "Exit" -font $font -fg black -relief raised -height 1 \
        -command {set exitFlag 1; reportState ; exit}
pack .b_exit -in .footerarea
#
# Set up grid for resizing.
#
#grid columnconfigure . 1 -weight 1
#grid columnconfigure . 3 -weight 1
```

APPENDIX G

Compile/Assemble Scripts

G.1 Build Script

```
#!/usr/bin/env sh
# These two variables ar defined and exported from .bashrc file!
# export PRU_SDK_DIR=/root/pru_sdk
# export PRU_CGT_DIR=/root/pru_2.0.0B2
# compile support library without optimization
# enabled to keep argument passing convention
#
# Following option turns optimization on -O3
# Does a great job of optimizing out a a delay loop!
#
#
PRU=./pru
INC=./
echo ""
echo "==> Compiling pru0Lib ..."
$PRU_CGT_DIR/bin/clpru \
--silicon_version=2 \
--hardware_mac=on \
-i$PRU_CGT_DIR/include \
-i$INC/include \
-i$PRU_CGT_DIR/lib \
-c \
$PRU/pru0Lib.c
echo "==> Compiling motorLib ..."
$PRU_CGT_DIR/bin/clpru \
--silicon_version=2 \
--hardware_mac=on \
-i$PRU_CGT_DIR/include \
-i$INC/include \
-i$PRU_CGT_DIR/lib \
-c \
$PRU/motorLib.c
echo "==> Compiling pru0 ..."
$PRU_CGT_DIR/bin/clpru \
--silicon_version=2 \
--hardware_mac=on \
-i$PRU_CGT_DIR/include \
-i$INC/include \
-i$PRU_CGT_DIR/lib \
-c \
$PRU/pru0.c
# compile and link
echo "==> Linking PRU code ..."
$PRU_CGT_DIR/bin/clpru \
--silicon_version=2 \
--hardware_mac=on \
-i$PRU_CGT_DIR/include \
-i$INC/include \
-i$PRU_CGT_DIR/lib \
-z \
pru0Lib.obj \
motorLib.obj \
```

```
pru0.obj \
-llibc.a \
-m pru0.map \
-o pru0.elf \
./pru0.cmd
# convert ELF to binary file pru_main.bin
echo "==> Converting .elf to .bin format"
$PRU_CGT_DIR/bin/hexpru \
./bin.cmd \
./pru0.elf
# build host program
echo ""
echo "==> Calling make to build host program and to assemble any .p files"
echo ""
make clean
make START_ADDR=`./get_start_addr.sh ./pru0.elf`
```

G.2 Makefile

```
# Makefile
# List the C src code files for ARM
ARM := ./arm
C_FILES := $(ARM)/beaglebot.c
C_FILES += $(ARM)/mio.c
C_FILES += $(ARM)/child.c
C_FILES += $(ARM)/PRUlib.c
C_FILES += $(ARM)/servo_driver.c
C_FILES += $(ARM)/srf02.c
C_FILES += $(ARM)/color_sensor.c
C_FILES += $(ARM)/accel.c
C_FILES += $(ARM)/rtc.c
C_FILES += $(ARM)/bbbLib.c
C_FILES += $(ARM)/robotLib.c
# List the assembler files for the PRUs
PRU := ./pru
P_FILES := $(PRU)/pru1.p
# List the dts files that need tBBo be compiles
DTS_FILES := ./dts/BB-PRUPID-MOTOR-00A0.dts
# Start address for PRU C object
C_FLAGS := -DSTART_ADDR=$(START_ADDR)
# PRU development directory and cross tool
#CROSSTOOL = ""
# Compiler
CC := $(CROSS_COMPILE)gcc
# Linker
LD := $(CROSS_COMPILE)gcc
STRIP := $(CROSS_COMPILE)strip
# Important to have hard option!
C_FLAGS += -Wall -O2 -mtune=cortex-a8 -march=armv7-a  -mfloat-abi=hard
C_FLAGS += -I $(PRU_SDK_DIR)/include
C_FLAGS += -I ./include
#L_FLAGS += -L $(PRU_SDK_DIR)/lib
L_FLAGS += -L $(PRU_SDK_DIR)/lib
L_FLAGS += -mfloat-abi=hard
# Libraries
L_LIBS += -lprussdrv
L_LIBS += -lm
# Recipe for assembler files
BIN_FILES := $(P_FILES:.p=.bin)
# Recipe for C code
O_FILES := $(C_FILES:.c=.o)
# Recipe for device tree overlay files
DTBO_FILES := $(DTS_FILES:.dts=.dtbo)
# ------------------------------------------------
all:  main $(BIN_FILES) $(DTBO_FILES)
main: $(O_FILES)
   $(LD) -static -o $@ $(O_FILES) $(L_FLAGS) $(L_LIBS)
   $(STRIP) $@
%.bin : %.p
   $(PASM) -V2 -I$(PRU_SDK_DIR)/include -b $<
%.o : %.c
   $(CC) $(C_FLAGS) -c -o $@ $<
%.dtbo : %.dts
   $(DTC) -@ -O dtb -o $@ $<
.PHONY   : clean all
clean :
   -rm -f $(O_FILES)
   -rm -f $(BIN_FILES)
   -rm -f $(DTBO_FILES)
   -rm -f main
```

APPENDIX H

SIUE Robot Cape Pins

H.1 Device Tree Overlay File

```
/* DTO for the Beaglebone black PID motor board Rev-B
 * Ver - 2.0
 * Clayton Faber - 2016
 */
/dts-v1/;
/plugin/;
/ {
   compatible = "ti,beaglebone", "ti,beaglebone-black";
   part-number = "BB-PRUPID-MOTOR";
   version = "00A0";
   /* This overlay uses the following resources */
   exclusive-use =
        //   P9-Header  //
   "P9.17", // I2C-2 - SCL
   "P9.18", // I2C-2 - SDA
   "P9.22",        // Accel. GPIO Interrupt - IN2 gpio0[2]
   "P9.25", // Accel. PRU0 Interrupt - IN1 (r31.t7)
   "P9.27", // PRU0 r30.t5 - Buffers Enable
        //   P8-Header  //
   "P8.11", // PRU0 r30.t15 - PRU0 LED
   "P8.12", // GPIO1[14]    - GPIO LED
   "P8.15", // GPIO1[15]    - GPIO SWITCH
   "P8.16", // PRU0 r31.t14 - PRU0 SWITCH
   "P8.27", // PRU1 r31.t8  - ENC1
   "P8.28", // PRU1 r31.t10 - ENC3
   "P8.29", // PRU1 r31.t9  - ENC2
   "P8.30", // PRU1 r31.t11 - ENC4
   "P8.39", // PRU1 r30.t6  - M4-0
   "P8.40", // PRU1 r30.t7  - M4-1
   "P8.41", // PRU1 r30.t4  - M2-0
   "P8.42", // PRU1 r30.t5  - M2-1
   "P8.43", // PRU1 r30.t2  - M1-0
   "P8.44", // PRU1 r30.t3  - M1-1
   "P8.45", // PRU1 r30.t0  - M3-0
   "P8.46", // PRU1 r30.t1  - M3-1
   //   Devices   //
   "i2c1",         // Claim use of I2C-2
      //NOTE: Although this says i2c1 it mounts in /dev as I2C-2 and is labled on
      // Beaglebone Black schematics as I2C1. On the Motor Board it is labled
      // as I2C-2 (same as the /dev name) to avoid confusion
   "pru0",      // Claim use of the PRUs here
   "pru1";      // and here
   fragment@0 {
      target = <&am33xx_pinmux>;
      __overlay__ {
         gpio_pins: pinmux_gpio_pins {          // The GPIO pins
            pinctrl-single,pins = <
      0x150 0x27 // P9_22 MODE7 | INPUT  | GPIO Pull-Down  //Used for Accelerometer
          Interrupt (opt.)
                0x030 0x07 // P8_12 MODE7 | OUTPUT | GPIO Output LED //GPIO Status LED
      0x03c 0x27 // P8_15 MODE7 | INPUT  | GPIO pull-down  //GPIO Switch Input
            >;
```

```
      };
 pru_pru_pins: pinmux_pru_pru_pins {   // The PRU pin modes
    pinctrl-single,pins = <
   // P9 //
   0x1ac 0x26  // P9_25 pru0_r31.t7,  MODE6 | INPUT  | PULL DWN | PRU // Used for
       Accelerometer Interrupt (opt.)
   0x1a4 0x0D  // P9_27 pru0_r30.t5,  MODE5 | OUTPUT | PULL UP  | PRU // Used to
       enable buffers
   // P8 //
   0x034 0x06  // P8_11 pru0_r30.t15, MODE6 | OUTPUT | PULL UP  | PRU // Used for
       PRU0 LED
   0x038 0x26  // P8_16 pru0_r31.t14, MODE6 | INPUT  | PULL DWN | PRU // Used for
       PRU0 SWITCH
   0x0E0 0x26  // P8_27 pru1_r31.t8,  MODE6 | INPUT  | PULL DWN | PRU // ENC1 Input
   0x0E4 0x26  // P8_29 pru1_r31.t9,  MODE6 | INPUT  | PULL DWN | PRU // ENC2 Input
   0x0E8 0x26  // P8_28 pru1_r31.t10, MODE6 | INPUT  | PULL DWN | PRU // ENC3 Input
   0x0EC 0x26  // P8_30 pru1_r31.t11, MODE6 | INPUT  | PULL DWN | PRU // ENC4 Input
   0x0A0 0x0D  // P8_45 pru1_r30.t0,  MODE5 | OUTPUT | DISABLE  | PRU // M3-0 Output
   0x0A4 0x0D  // P8_46 pru1_r30.t1,  MODE5 | OUTPUT | DISABLE  | PRU // M3-1 Output
   0x0A8 0x0D  // P8_43 pru1_r30.t2,  MODE5 | OUTPUT | DISABLE  | PRU // M1-0 Output
   0x0AC 0x0D  // P8_44 pru1_r30.t3,  MODE5 | OUTPUT | DISABLE  | PRU // M1-1 Output
   0x0B0 0x0D  // P8_41 pru1_r30.t4,  MODE5 | OUTPUT | DISABLE  | PRU // M2-0 Output
   0x0B4 0x0D  // P8_42 pru1_r30.t5,  MODE5 | OUTPUT | DISABLE  | PRU // M2-1 Output
   0x0B8 0x0D  // P8_39 pru1_r30.t6,  MODE5 | OUTPUT | DISABLE  | PRU // M4-0 Output
   0x0BC 0x0D  // P8_40 pru1_r30.t7,  MODE5 | OUTPUT | DISABLE  | PRU // M4-1 Output
         >;
      };
 bb_i2c1_pins: pinmux_bb_i2c1_pins {
             pinctrl-single,pins = <
             0x158 0x72    // i2c1_sda,  SLEWCTRL_SLOW | IMPUT_PULLUP | MODE2
             0x15c 0x72    // i2c1_scl,  SLEWCTRL_SLOW | INPUT_PULLUP | MODE2
         >;
      };
   };
};
fragment@1 {         // Enable the PRUSS
   target = <&pruss>;
   __overlay__ {
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&pru_pru_pins>;
   };
};
fragment@2 {         // Enable the GPIOs
   target = <&ocp>;
   __overlay__ {
      gpio_helper {
         compatible = "gpio-of-helper";
         status = "okay";
         pinctrl-names = "default";
         pinctrl-0 = <&gpio_pins>;
      };
   };
};
fragment@3{
   target = <&i2c1>;
   __overlay__ {
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&bb_i2c1_pins>;
clock-frequency = <100000>;
#address-cells = <1>;
#size-cells = <0>;
```

```
        };
    };
};
```

H.2 Overlay Install Script

```
#!/bin/bash
#fist copy the dtbo to the FW folder
cp BB-PRUPID-MOTOR-00A0.dtbo /lib/firmware/
#next write the eeprom on the board
cat ./motor.eeprom > /sys/bus/i2c/devices/1-0054/eeprom
#make sure that the dtbo script is executable
chmod +x ./dtbo
#place it in the initramfs-tools/hooks folder so it will be ran when doing the update
    script
cp ./dtbo /etc/initramfs-tools/hooks/
#make a backup of the bootloader just incase
cp /boot/initrd.img-$(uname -r) /boot/initrd.img.bak
#run the update script
/opt/scripts/tools/developers/update_initrd.sh
#copy the new initrd to the boot directory and rename
cd /boot/uboot
mv initrd.img ../initrd.img-$(uname -r)
cd -
```

H.3 Quick Reference Guide

Documentation for Motor Controller Board

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ PRU #1 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Bit #	Name	Purpose	Header Pin
R30.t0	M3-0	Motor 3 control signal	P8_45
R30.t1	M3-1	Motor 3 control signal	P8_46
R30.t2	M1-0	Motor 1 control signal	P8_43
R30.t3	M1-1	Motor 1 control signal	P8_44
R30.t4	M2-0	Motor 2 control signal	P8_41
R30.t5	M2-1	Motor 2 control signal	P8_42
R30.t6	M4-0	Motor 4 control signal	P8_39
R30.t7	M4-1	Motor 4 control signal	P8_40

Bit #	Name	Purpose	Header Pin
R31.t7	PRU0_INT	Interrupt from accelerometer	P9_25
R31.t8	ENC1	Encoder 1 input	P8_27
R31.t9	ENC2	Encoder 2 input	P8_29
R31.t10	ENC3	Encoder 3 input	P8_28
R31.t11	ENC4	Encoder 4 input	P8_30

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ PRU #0 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Bit #	Name	Purpose	Header Pin
R30.t5	BUF_ENA	Enables tri-state buffers	P9_27
R30.t15	LED	PRU #0 LED	P8_11

Bit #	Name	Purpose	Header Pin
R31.t14	SWITCH	PRU #0 Switch	P8_16

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ GPIO ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

GPIO #	Name	Purpose	Header Pin
44	GPIO1[12]	GPIO LED	P8_12
47	GPIO1[15]	GPIO Switch	P8_15
2	GPIO0[2]	Accelerometer GPIO interrupt	P9_22

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ I2C-2 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

I2C-2 SCLK	I2C Bus #2 serial clock	P9_17
I2C-2 SDA	I2C Bus #2 serial data	P9_18