2010

Hardware, Software, and Low-Level Control Scheme Development for a Real-Time Autonomous Rover

Brenton K. Wilburn
West Virginia University

Hardware, Software, and Low-Level Control Scheme Development for a Real-Time Autonomous Rover

Brenton K. Wilburn

Thesis Submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Mechanical Engineering

Larry E. Banta, Ph.D., Chair
Yu Gu, Ph.D.
Powsiri Klinkhachorn, Ph.D.

Department of Mechanical Engineering

Morgantown, West Virginia
2010

Keywords: Autonomous, Rover, Mobile, Robot, RTAI Linux, Real-Time

ABSTRACT

Hardware, Software, and Low-Level Control Scheme Development for a Real-Time Autonomous Rover

Brenton K. Wilburn

The objective of this research is to develop a low-cost autonomous rover platform for experiments in autonomous navigation. This thesis describes the design, development, and testing of an autonomous rover platform, based on the commercial, off-the-shelf Tamiya TXT-1 radio controlled vehicle. This vehicle is outfitted with an onboard computer based on the Mini-ITX architecture and an array of sensors for localization and obstacle avoidance, and programmed with Matlab/Simulink® Real-Time Workshop (RTW) utilizing the Linux Real-Time Application Interface (RTAI) operating system.

First, a kinematic model is developed and verified for the rover. Then a proportional-integral-derivative (PID) feedback controller is developed for translational and rotational velocity regulation. Finally, a hybrid navigation controller is developed combining a potential field controller and an obstacle avoidance controller for waypoint tracking.

Experiments are performed to verify the functionality of the kinematic model and the PID velocity controller, and to demonstrate the capabilities of the hybrid navigation controller. These experiments prove that the rover is capable of successfully navigating in an unknown indoor environment. Suggestions for future research include the integration of additional sensors for localization and creation of multiple platforms for autonomous coordination experiments.

ACKNOWLEDGEMENTS

First of all I would like to thank my research advisor, Dr. Larry Banta. I am extremely grateful for the guidance, support, and patience you have given me throughout this project. The lessons you have taught me are invaluable.

I would like to thank my committee members, Dr. Yu Gu and Dr. Powsiri Klinkhachorn for evaluating my project and for the input you have given me. Also, I would like to thank the WVU Flight Controls Research Lab for allowing me to use their resources and laboratory space throughout this project.

I would also like to express my appreciation to Dr. Mario Perhinschi. Even though you were not a part of my advising committee, you took the time out of your busy schedule to mentor me on my control scheme development.

I am extremely thankful to NASA EPSCoR and West Virginia Space Grant Consortium Graduate Fellowship programs through which this research was funded.

To my loving family and friends who have been a constant source of encouragement throughout this project. Thank you for all your support.

Last, but certainly not least, I would like to thank my love, Jennifer Davis. You have stood by me through everything.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMECLATURE

Variables

Symbol	Description	Units
	English	
d	Distance from IR sensor	m
D	Wheel diameter	m
$\boldsymbol{e}$	Simulation error vector	-
$\vec{\boldsymbol{F}}$	Virtual force	N
k	Intensity factor	-
$\boldsymbol{K}$	Controller gain	-
$\boldsymbol{l}$	Distance between front and rear wheels	m
q	Encoder ticks	ticks
$\boldsymbol{\Delta t}$	Time step size	s
$\boldsymbol{t}$	Time step	s
$\boldsymbol{u}$	Controller output	-
U	Potential	-
$\boldsymbol{v}$	Translational velocity	m/s
$\boldsymbol{v_1}$	Translational velocity	m/s
$\boldsymbol{v_2}$	Rotational velocity	rad/s
$w_{1\text{-}6}$	IR sensor weights	-
W	Effective width of vehicle	m
$\boldsymbol{x}$	Position coordinate in the direction of the x-axis	m

$\dot{x}$	Velocity of the rover in the direction of the x-axis	m/s
y	Position coordinate in the direction of the y-axis	m
$\dot{y}$	Velocity of the rover in the direction of the y-axis	m/s
	Greek	
ϕ	Steering angle	rad
$\dot{\phi}$	Rate of change of steering angle	rad/s
θ	Orientation	rad
$\dot{\theta}$	Angular velocity	rad/s
ω	Rotational velocity	rad/s
ζ	Encoder distance factor	m/tick
	Subscripts	
att	Attractive	-
c	Commanded	-
D	Derivative	-
$drive$	Drive motors	-
f	Front wheel	-
g	Goal	-
I	Integral	-
k	Current step	-

L	Left side	-
m	Measured	-
OA	Obstacle Avoidance	-
P	Proportional	-
PFC	Potential Field Controller	-
r	Rear wheel	-
R	Right side	-
steer	Rotational	-
track	Translational velocity during obstacle avoidance	-
$\boldsymbol{v}$	Translational velocity	-
x	In x-direction	-
y	In y-direction	-
$\boldsymbol{\omega}$	Rotational velocity	-

ACRONYMS

Symbol	Description	Units
API	Application Programming Interface	-
CD	Compact Disc	-
CAD	Computer-aided Design	-
COTS	Commercial Off-the-shelf	-
DAQ	Data Acquisition	-
ESC	Electronic Speed Controller	-

GPS	Global Positioning System	-
GUI	Graphical User Interface	-
IP	Internet Protocol	-
IR	Infrared	-
NiCd	Nickel-Cadmium	-
NiMH	Nickel-metal Hydride	-
OS	Operating System	-
PCI	Peripheral Component Interconnect	-
PFC	Potential Field Controller	-
PID	Proportional Integral Derivative	-
RAM	Random-access Memory	-
RC	Radio Controlled	-
RTAI	Real-time Application Interface	-
RTOS	Real-time Operating System	-
RTW	Real-time Workshop	-
SATA	Serial Advanced Technology Attachment	-
UAV	Unmanned Aerial Vehicle	-
UGV	Unmanned Ground Vehicle	-
USB	Universal Serial Bus	-
WIFI	Refers to Wireless Local Area Network	-
WVU	West Virginia University	-

CHAPTER 1 - INTRODUCTION

1.1 MOTIVATION

The field of robotics is rapidly evolving to keep pace with the ever-increasing demands of society. Robots have made a profound impact on modern manufacturing industries. In just a few decades, robots have become the backbone to most manufacturing facilities. Factories now utilize robots to automate many of the jobs too mundane, time-consuming, or dangerous for humans. However, most of these robots have been employed in applications where their working conditions were structured or predictable.

In order for robotics to extend such a role into less predictable applications, robots must become more mobile and more intelligent. This is becoming increasingly possible due to the rise in computing power and decrease in computer size, power consumption, and costs. As the field matures, many new applications of robots are becoming possible including space and undersea exploration, search and rescue, security and surveillance, and many others. As such, the area of autonomous mobile robots is becoming an attractive field of study.

The design and control of an autonomous mobile robot is a complex task. Such a robot must be capable of navigating and performing tasks within its given real-world environment with minimal human interaction. The robot must be able to navigate in previously unknown surroundings which are often highly unstructured. As such, it must possess an assortment of sensors that can be used to construct an understanding of its environment. Once its surroundings have been assessed, the robot can plan a collision-free path to satisfy its goal. Once a plan has been formulated, the robot must then use an assortment of control schemes in order to execute this plan. Often, such a robot must work cooperatively with other robots to achieve a goal. Each of the tasks

described above can be accomplished in a wide variety of ways. As such, a great deal of effort is currently being put into developing new and improved methods for performing these tasks.

1.2 RESEARCH OBJECTIVES

The objective of this research is to develop a low-cost, versatile mobile robotics platform to serve as the framework for future autonomous navigation and multi-vehicle coordination research. In order to keep costs low, *commercial off-the-shelf* (COTS) components are to be used whenever possible. The extent of this project can be summarized by the following tasks:

1. Design and construct a mobile robotics platform for above mentioned objectives using a *Tamiya TXT-1* radio-controlled vehicle as a starting point.
2. Equip the robot with a powerful onboard computer and a variety of sensors pertinent for autonomous navigation.
3. Install and validate a real-time operating system on the onboard computer.
4. Develop a set of Simulink blocks for integration of the rover's sensors and actuators into control schemes utilizing the capabilities of the real-time operating system.
5. Develop a method for wireless data transmission, control scheme selection, monitoring, and uploading of programs.
6. Develop a mathematical model of the rover kinematics and verify using onboard sensory data.
7. Develop low-level control schemes for translational and rotational velocity regulation.
8. Conduct autonomous navigation experiments for validation of system functionality and utility.
9. Demonstrate the hardware and software in a laboratory setting.

1.3 THESIS OUTLINE

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of previous research efforts on developing autonomous mobile robot platforms, including hardware, software, control, and operating systems. Chapter 3 details the selection, design, and construction of the rover hardware. Chapter 4 describes the real-time operating system and software architecture used by the rover. In Chapter 5, the development and implementation of the rover's control schemes are discussed. The performance of these control schemes is presented in Chapter 6. Finally, Chapter 7 provides the conclusions of this research along with recommendations for future work.

CHAPTER 2 - LIETERATURE REVIEW

Much advancement has been made in the field of robotics over the past few years. At the onset of this project, an extensive literature review was performed in order to explore the existing applications, methods, and advancements in robotics. This chapter describes the findings of this review and is organized as follows. Section 2.1 describes some of the mobile robotic platforms implemented by other research teams. Section 2.2 covers some of the choices available for sensory input. Section 2.3 covers some of the methods used for controlling mobile robots. Finally, Section 2.4 discusses some of the operating system and software packages available for the programming of robots.

2.1 MOBILE ROBOTICS PLATFORMS

Mobile robots have been developed to perform a wide variety of tasks. No single platform is suited for all applications. For this reason, many types of robotic platforms have been developed, each geared toward the particular problem for which it was designed. In this section, many previously designed mobile robotics platforms will be described, as they relate to the research contained in this thesis.

Many robotic test-beds are unmanned aerial vehicles (UAVs) (1) (2) (3) (4) (5) (6) (7). An example of this category of robot is North Carolina State's Stingray UAV (1). This UAV was constructed from off-the-shelf components to research autonomous flight and navigation. Templeton et. al. (5) created a UAV based on a helicopter for the purpose of vision-based landing and terrain mapping.

West Virginia University utilized three UAVs for multiple aircraft coordination and formation flight (6) (7). Both the University of Pennsylvania (2) and Massachusetts Institute of

Technology (4) have created UAVs for cooperation with unmanned ground vehicles (UGVs). The team at the University of Pennsylvania used a fixed-wing UAV to coordinate with a ground-based, wheeled robot for surveillance experiments. The Massachusetts Institute of Technology group used both a traditional airplane style UAV and a blimp to coordinate with a UGV.

Another primary type of mobile robot is the UGV. Like UAVs, ground-based robotic platforms also come in many different configurations. These are typically categorized by their method of locomotion.

Several research groups have created mobile robot platforms based on hovercrafts (8) (9) (10). The California Institute of Technology (8) (9) created such a platform for performing multi-vehicle coordination experiments in an indoor environment. The University of Illinois (10) also created a very similar hovercraft robot platform. Both of these groups created multi-robot coordination systems which communicate over a wireless local area network (WLAN). Such a hovercraft platform does not actually move by making contact with the ground; instead it floats on a cushion of air and propels itself with ducted fans. Hovercraft based systems are capable of traversing uneven terrain or water. However, they require more advanced controllers to accommodate for their sensitive dynamic nature.

Possibly the most common form of UGV is the wheeled mobile robot (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (28) (29). Compared to other types of robot locomotion, a wheeled mobile robot has very good efficiency and is much simpler to control (30). Many mobile robots based on flying, hovering, or walking must work continuously just to stay in one place or balance. However, a wheeled robot typically does not have to worry about such things, so more attention can be focused on control, navigation, etc.

A common configuration for wheeled robots is to have two wheels and a caster which is used for balance (22). This type of wheeled robot is typically referred to as a differential drive.

With such a robot, the angular velocity of the vehicle and the curvature of its path are determined by the relative speeds of the two wheels. This type of robot is highly maneuverable since it is capable of turning in place.

Many wheeled mobile robots are designed for omni-directional movement (24) (25) (26) (27). Omni-directional robots can move in any direction regardless of their orientation (30). Omni-directional movement has been achieved using several different wheel types and configurations. Do Nascimento et. al. (24) and Kanjanawanishkul et. al. (25) used a configuration of three omni-directional wheels to achieve this type of robot movement. The wheels used on the robot are equipped with rollers around the perimeter of the wheel that allow it to move in any direction. This type of wheel is commonly referred to as a *Swedish* wheel. Zhu et. al. (26) achieved omni-directional movement by using four wheels capable of rotating fully around their vertical axes. Moore et. al. (27) used the same concept except with six wheels. Omni-directional robots are more maneuverable than the differential drive robots. However, they often have problems with ground clearance due to mechanical constraints of omni-directional wheels (30) and their drive mechanisms are generally more complex.

The final type of wheeled robot investigated was the car-like, or Ackerman, steering configuration. Such a robot has four wheels. Typically, two or four wheels are used for propelling the robot, and the front two wheels are used for steering. This type of vehicle normally has a minimum turning radius that is relatively large in comparison to the size of the vehicle, and it cannot move laterally without multiple maneuvers. However, it is much more stable than differential drive robots at higher speeds (30).

Several examples of car-like mobile robots can be found in (2) (14) (15) (17) (18) (19) (21) (23) (31) (32). Ferguson et. al. (21) created an all-terrain vehicle (ATV)-like robot for exploring and mapping abandoned underground mines. Hsu et. al. (32) adapted an ordinary passenger car for the

development of an automated parking system. The vehicle was equipped with sensors and actuators in order to autonomously perform parallel parking maneuvers.

A common approach to developing a mobile robot platform is to start with a COTS vehicle and modify it with sensors, actuators, and a computer system in order to create a robot. A very common off-the-shelf vehicle used in for this purpose is the Tamiya TXT-1 radio-controlled (R/C) 1/10th scale monster truck (2) (14) (15) (17) (19) (23) (31) (33). Researchers at Oklahoma State University (14) adapted the TXT-1 for multi-vehicle formation control, coordination, and mapping. The UGV from the University of Pennsylvania UAV-UGV coordination experiments (2) was based on the TXT-1 platform. Fierro et. al. (31) and Xiao et. al. (15) equipped the TXT-1 with an omni-directional camera for vision based navigation and obstacle avoidance. As will be discussed in Chapter 3, the TXT-1 is the basic foundation for the platform developed in this thesis.

2.2 SENSORY METHODS

Many of the robots studied in this literature review were equipped with sensors which were specific to the tasks being performed by the robot. However, such specialized sensors will not be discussed in this section. What will be covered here are common sensors which are useful for localization, navigation, obstacle avoidance, and control of wheeled mobile robots.

In robotics, localization refers to the robot's ability to determine its current position in its environment. One of the simplest ways a robot can accomplish this is to measure the velocity of its wheels and calculate its position based on geometry. Measuring the wheel velocities is typically done using optical encoders located on the wheel axes. Nearly every robot investigated at least partially used optical encoders for localization. As will be discussed later, one limitation of this method is the fact that wheel slipping often occurs and can introduce error into this estimation of position.

A common approach to mitigating this error is to introduce a second sensor to compare with odometric measurements. Accelerometers, gyroscopes, and electronic compasses are often used as secondary sensors to verify the position data calculated from wheel encoders. These types of sensors are utilized by many robotic platforms (1) (2) (4) (5) (14) (27).

Many robotic platforms (5) (10) (11) (13) (14) (15) (16) (19) (21) (31) utilize cameras to assist in localization, obstacle avoidance, and coordination tasks. Xiao et. al. (15) used a camera pointing to a convex mirror located directly above to robot to provide a 360° view of the robot's surroundings. Fierro et. al. (31) also used this type of camera for his navigation algorithms and formation control. This type of camera setup is commonly referred to as omni-directional vision.

Another common camera configuration found on mobile robots is stereoscopic vision. In this setup, two cameras facing the same direction are mounted a fixed width apart. When the images from the cameras are processed, approximate distances to *landmarks* or other relevant features may be triangulated. Cruz et. al. (14) used a stereoscopic camera to recognize identifying *nametags* on its fellow robots. Marshall et. al. (19) (23) used a very similar setup for locating other robots in formation control. Cameras are becoming much more popular since their price and size have dropped greatly over the past few years.

Cameras can provide a robot with a vast amount of information about its position and environment. However, processing the data into meaningful information is often very difficult. Colors, textures, and shadows can greatly interfere with the usability of data obtained from cameras. Also, the processing power needed for advanced image processing algorithms is often too great for the computers on small mobile robots.

Laser range finders provide an alternative or supplement to cameras. Laser range finders use lasers to accurately detect the distance to solid objects. Many of these sensors quickly scan an area in order to provide a 3-dimensional image of the environment. This ability makes this sensor a

popular choice for robot applications where mapping is a primary objective. Mano et. al. (13) and Zhu et. al. (26) used a laser range finder for navigation, mapping, and obstacle avoidance. Templeton et. al. (5) used a laser range finder on a UAV helicopter to map the ground and find acceptable landing spots.

Laser range finders are extremely versatile and powerful. However, they are currently too expensive for many robotic projects, especially coordination projects in which many identical robots must be constructed.

Global positioning systems (GPS) are another popular solution for localization. However, these systems are typically limited to outdoor use only. Cruz et. al. (14) utilized a GPS system to provide a secondary estimation of position. Templeton et. al. (5) also used a GPS to provide the position of an outdoor robot.

Infrared (IR) proximity sensors and ultrasonic range finders are commonly used for obstacle avoidance and mapping. These types of sensors give the distance to an object directly in front of them. The function of this sensor is similar to the laser range finder except they typically do not scan a region; they only measure a single distance in front of them. Regardless, simple clusters of IR or ultrasonic sensors are capable of cheaply detecting obstacles around a robot if properly implemented. Cruz et. al. (14) and Xiao et. al. (15) utilized IR sensors for obstacle avoidance. Ferreira et. al. (34) compared the performance of an array of ultrasonic range finders and a laser range finder for obstacle avoidance. It was found that the laser range finder provided slightly better results, but both provided acceptable performance.

IR sensors and ultrasonic range finders have a few limitations. First, both sensors have a limited effective range. Also, the sensors are designed to measure objects at a perpendicular angle. For this reason, angle variations are likely to affect the accuracy of their measurements. Finally, environmental factors also affect the accuracy of these sensors.

While not all of these sensors will initially be equipped on the rover being developed in this thesis, it is important to understand the options available for future expansion.

2.3 CONTROL METHODS AND PATH PLANNING

The control architectures of mobile robots are highly dependent on the purpose of the robot. Consequently, a multitude of control architectures are available for use on mobile robots. This section will briefly cover some of the most common methods and the ones used for this project. These methods will be discussed in greater detail in Chapter 5. This subsection will first discuss the different approaches to developing a mathematical model of a wheeled mobile robot. The second subsection will discuss the low-level motion control schemes to execute a given command. The last subsection will discuss the high-level behavioral architectures for generating suitable obstacle-free paths.

2.3.1 MODEL REPRESENTATION

The process of controlling a mobile robot begins with developing a mathematical model to describe the system. There are two different approaches to obtaining this model. The first, and most accurate, method involves modeling the mechanical dynamics of the system to derive the equations of motion. This is commonly referred to as the dynamic model. This approach has been used by (8) (9) (10) (19) (23) (35).

The dynamic equations of motion for a real-world robot are often very complex to model and control. For this reason, it is common to neglect the dynamic forces of the robot and consider only the geometric constraints of the chassis. This is referred to as kinematic modeling. This type of vehicle model is used by many mobile robotics efforts (14) (17) (22) (26) (28) (31). Kinematic control is generally sufficient in applications where dynamic forces are small, such as when the

robot is travelling at low speeds or friction forces are small (30). A kinematic model will be used for the mathematical model of the rover being developed in this thesis.

2.3.2 MOTION CONTROL

Once the mathematical model of a robot has been developed, its motion controllers can be designed. Three common motion control tasks are path following, trajectory tracking, and posture (or point) stabilization (36) (37). The design of the controller depends heavily on the objectives of the robot and the path planning algorithms utilized (discussed in the next section).

As paraphrased from (36), path following consists of driving a robot along a curvature in the coordinate space. Trajectory tracking encompasses path following, except the path is now time-dependent. Finally, with posture stabilization, the path and time are irrelevant so long as the robot eventually reaches its desired position and orientation. Each of these tasks requires a vastly different control scheme. Examples of these have been provided by (36) (37).

Each of the above mentioned motion control tasks require precise control of the velocities of the robot. This is frequently accomplished using a low-level controller. This controller compares the desired velocities as commanded by the above described controllers to the robot velocities measured using the odometric sensors. Cruz et. al. (14) used a proportional-integral-derivative (PID) controller programmed on a dedicated circuit board for this purpose, and Marshall et. al. (19) (23) used a proportional-integral (PI) controller. A similar controller will be developed for this project.

Control of a robot with an Ackerman style wheel configuration is more difficult than an omni-directional robot. This is due to the fact than an Ackerman style vehicle simply cannot move laterally or rotate in place. These maneuverability limitations are commonly referred to as nonholonomic constraints. With this type of steering mechanism, actions such as parallel parking

are greatly complicated since not all configurations and paths are always available. As a result, the complexity of controlling a car-like robot greatly depends on the level of control desired.

2.3.3 PATH PLANNING AND OBSTACLE AVOIDANCE

A standard task for a mobile robot is to move from a starting position to a goal. This is commonly referred to as waypoint tracking. This task generally consists of some type of *path planning* and *obstacle avoidance.* Path planning refers to calculating a route to allow the robot to reach its goal, and obstacle avoidance refers to adapting the robot's route to avoid collisions (30).

Various methods have been developed for path planning. One popular method is *road map* planning (30). This method uses a map of the area to determine open pathways between obstacles. The path planner then compares all of the combinations of pathways leading to the goal and calculates the shortest distance. One road map planner is called the *visibility graph*. It calculates the shortest path possible by finding the corners of the obstacles (assuming polygon obstacles) and creating a route which passes as close as possible to the obstacle. Li et. al. (11) used this method for a mobile robotic security team. One problem with this is that robots may accidentally pass too closely and collide with an obstacle. To solve this problem, a similar road map method, the *Voronoi diagram*, is often used. The Voronoi diagram method generates paths which are as far as possible from the obstacles. This method was used by (38) and (39).

Another popular method for path planning is the *potential-field controller* (PFC). This method uses artificial "forces" to guide the path of the robot. The goal is represented as an attractive force, while obstacles are represented as repulsive forces. The superposition of the forces acting on the rover produces a smooth path (30). A benefit of this planning method is that it not only produces a path for the robot, but also the control law for following the path. Consequently, this is relatively simpler to implement than most other algorithms. The downfall to this approach is

that the path may get stuck in complex obstacles, and it may be difficult for the robot to pass through some narrow spaces. This method was covered by (14) (30) (40) (41) (42) (43).

The path planning methods described above assume that a complete map of the area is initially available. However, this is often not the case. Many robotic platforms are used for exploratory purposes. Thus, the robot must *react* to obstacles as they are encountered. A way of overcoming this is to use a *hybrid* controller consisting of different *behaviors* or modes. Cruz et. al. (14) used a PFC controller consisting of only an attractive potential for the goal for a waypoint tracking behavior. While the robot is moving, IR sensors on the front of the robot watch for obstacles. If an obstacle is detected within a certain distance, the robot initiates an obstacle avoidance behavior which uses a separate controller to track the contour of the obstacle. Once the obstacle is no longer blocking the goal, the PFC controller is reengaged. A very similar approach will be used for this project.

2.4 OPERATING SYSTEMS AND SOFTWARE

Control of a mobile robot is typically accomplished with either an embedded microcontroller or a small form-factor computer such as a PC-104 system (1) (6) (10) (14).

The operating system refers to the most basic software required by the computer to operate. Often, the operating system on a robot's computer is very similar to that of a typical consumer computer. Examples of this include Windows, Linux, and Mac OS X. Control schemes are then written in some language compatible with the installed operating system. The Linux operating system has become a popular choice for mobile robots (5) (14) (19) (23) (28) (29) (26) (27). Its highly open-source nature allows researchers the ability to customize it to fit the needs of their particular application.

Most ordinary operating systems are designed to execute tasks in soft real-time (44). This means that the CPU will attempt to execute a given task at its earliest convenience; no strict timing constraints are available. This is usually acceptable for consumer applications. However, mobile robots need better timing guarantees for control purposes. An operating system capable of providing such timing constraints is typically referred to as a real-time operating system (RTOS). Real-time operating systems have been applied to many robotic applications (1) (26) (28) (29) (44).

There are many options available for real-time operating systems. Several popular options are *Windows CE, QNX Neutrino, VxWorks, RTLinux, Xenomai,* and *RTAI Linux*. Each of these real-time operating systems functions in slightly different ways, but each achieves real-time execution of tasks. Windows CE, QNX Neutrino, VxWorks, and RTLinux are all proprietary, commercial solutions. As a result, not only are they less flexible, but they are also costly. RTAI Linux and Xenomai are both open-source extensions of Linux that provide real-time operating conditions. Barbalace et. al. (45) compared VxWorks, the leading commercial RTOS, with RTAI Linux and Xenomai and found both to be worthy alternatives to the commercial software. It was also found that RTAI-Linux offers slightly better performance than Xenomai. Finally, RTAI-Linux includes a software suite called RTAI-Lab which allows for real-time code generation using Matlab® Simulink® Real-time Workshop (RTW). For this reason, RTAI Linux will be used in this project. This will be discussed in greater detail in Chapter 4.

CHAPTER 3 - HARDWARE

One of the primary aspects of this project was the design and construction of a versatile mobile robotic platform. Since no existing platform was available, one was selected, assembled, and modified to fulfill the needs of this project. In addition to the mechanical aspects of the platform, an onboard computer system and an array of sensors and actuators were required to allow the robot to navigate autonomously. This chapter will detail the design requirements and considerations used when selecting the various components for the system, and will describe the specifications, capabilities, and any installation modifications of the components that were chosen.

3.1 SYSTEM REQUIREMENTS

Before any decisions could be made regarding the selection of components for the platform, a general set of requirements was compiled to ensure that the final product would fully satisfy the needs of this research as well as provide a flexible platform for researchers to come.

It was necessary that the platform be capable of navigating indoors or outdoors, of operating autonomously, and be constructed from *commercial-off-the-shelf* (COTS) components in order to keep costs low. These requirements will be further developed in the following sections.

3.2 VEHICLE

3.2.1 REQUIREMENTS

The first set of requirements imposed on the platform involves maneuverability. The platform must have the ability to easily traverse typical indoor or outdoor terrain. It is important to specify what is meant by typical. The rover will generally operate in an indoor environment, which

includes hallways, corridors, and tight spaces. In such situations, the terrain is expected to be relatively smooth and level. This environment is expected to contain numerous obstacles such as walls, people, furniture, and other objects. In addition to navigating indoors, the rover must also be well-suited for outdoor terrain. This involves sloping and uneven terrain, as well as moderate environmental moisture. It is not expected that the rover will be capable of traversing all types of terrain, but moderate versatility is essential. The rover will not be expected to be water-proof, but general dampness and humidity should not damage the vehicle.

From these environmental criteria, three distinct requirements can be imposed upon the design of this platform. The rover must be small, have a tight turning radius, and be rugged. Since the average width of an interior door is approximately 30 inches, it is reasonable to state that the platform must be narrower than this width. No exact requirements are imposed on the length or height of the rover. However, consideration must be taken to keep the length small enough as to not impair turning radius and the height low enough as to not make the vehicle top-heavy. Turning radius should be kept to a minimum for maneuverability indoors. In addition to tight maneuvering, the rover must also be resilient enough to overcome reasonable environmental challenges. Since it is anticipated that the terrain may be rough and uneven, the vehicle should have shock absorbers to dampen vibrations and protect onboard equipment. The vehicle should be able to maintain traction on loose soil, damp conditions, and steep slopes. Finally, the platform should be durable enough to withstand minor rollover incidents and collisions.

As stated above, the platform must be capable of operating autonomously. The vehicle selected must therefore be able to house and support the weight of an onboard computer, sensors, and actuators. It was accepted that any COTS vehicle would require modification to house the onboard equipment, but such a vehicle should be readily modifiable.

The final requirement for the vehicle is that it must be capable of operating for an extended period of time without the need for refueling or recharging. This requirement is relatively flexible in definition, as run-time is to be maximized. It was decided that approximately one hour is the minimum amount of time that the platform should be able to operate continuously. Any less time would interfere with the range used in a typical experiment session.

3.2.2 SELECTION

When selecting a base vehicle for this platform, two primary options were available. The first option was to completely construct a vehicle from its most basic components. While this option would allow for the most customizability, the actual construction and debugging phase would likely be very time consuming. Also, since multi-robot coordination is to be left as an option for future research, a *from-scratch* construction would be less consistent than a commercial option for creating multiple robots. Finally, the cost of a vehicle built with this approach would likely be more expensive than a bundled package, especially if labor costs were encountered for any custom machining work. As such, the option of designing a completely custom vehicle platform was not considered a viable option.

Several commercial research robots were considered, but these were too costly and did not meet the requirements of this project. The next option explored was that of a radio-controlled (RC) vehicle aimed at hobbyists. Such a vehicle is readily available from multiple suppliers in various sizes and configurations in the forms of completely assembled vehicles as well as kits requiring assembly. Most of these RC vehicles are designed for outdoor use. Also, a wide range of accessory parts is available in industry standard sizes, thus simplifying the process of modifying a selected vehicle. Finally, the cost of these vehicles ranged around several hundred dollars, as opposed to several thousand for the other options explored, making it the most cost effective.

3.2.3 SPECIFICATIONS

The base vehicle chosen for this project was a Tamiya TXT-1 1/10th scale RC vehicle based on the design of a *monster truck*. It was sold as an unassembled kit vehicle for $419.99. The vehicle can be seen below in Figure 3.1 and its factory size specifications can be seen in Table 3.1.

Figure 3.1: TXT-1 R/C vehicle

Table 3.1: Stock specifications for the TXT-1

Dimension	Value
Length	510 mm
Width	385 mm
Height	297 mm
Minimum clearance	49 mm
Weight	5.010 kg
Wheel base	330 mm
Tire diameter	165 mm

Overall the design and construction of the vehicle make it well suited for indoor or outdoor navigation. This vehicle was classified under the *rock crawling* category of RC trucks. It is designed with twin, high torque motors, high ground clearance, independent suspension, and four-wheel

drive to allow it to maneuver rough terrain. The chassis is constructed from machined aluminum, and all of the drive-train components are covered by guards or seals to protect against dirt and moisture. Additionally, the size of the vehicle is neither too small to equip a wide range of sensors or too large to navigate easily indoors. As was discussed in Chapter 2, this vehicle was successfully used as a research platform by many different teams.

3.2.4 MODIFICATIONS

Several modifications were made to the vehicle in order to make it more suitable for this research. The first modification to the vehicle was to replace the original nickel-cadmium (NiCd) battery. This battery had a lifespan of about 15 minutes. The original battery was replaced by a DuraTrax DTX4200 nickel-metal hydride (NiMH) battery with a rating of 4200 mAh. With the new battery, run-times increased to approximately one hour. The exact life of the battery varies greatly based on the intensity of usage. Since the DuraTrax battery has the same form factor and connection as the original battery, installation was accomplished by simply removing the old battery and plugging in the new one.

Another modification to the rover was the addition of an electronic speed controller (ESC). This was required because the original speed controller did not have a high enough resolution to accurately control the speed of the vehicle. The ESC chosen to replace the original speed controller was the Novak Super Duty XR. This controller was chosen because it provides the vehicle with precise control of the motors in forward and reverse, has engine braking capabilities, and supports the dual motor configuration found on the vehicle. The ESC also helps to conserve battery life since it uses pulse width modulation to vary the speed of the motor instead of dissipating energy through a resistor as in the original speed controller. Finally, the ESC interfaces via a standard servo motor connection. The chosen motor controller can be seen in Figure 3.2 below.

Figure 3.2: Novak Super Duty XR electronic speed controller

The electronic speed controller was installed in the same location as the original motor controller. In order to mount the new controller, the supports for the original controller were cut away from the plastic harness. The original harness compared to its modified version can be seen in Figure 3.3. In addition to the physical modifications needed to mount the ESC, one of the two battery inputs was shorted out as outlined in the instructions for the ESC to enable the use of only one battery. Then three 0.1 μF ceramic capacitors were soldered across the motor terminals to help reduce electrical noise. Finally, the ESC was bolted to the plastic harness and the wires were arranged neatly. The completed installation can be seen in Figure 3.4.

.

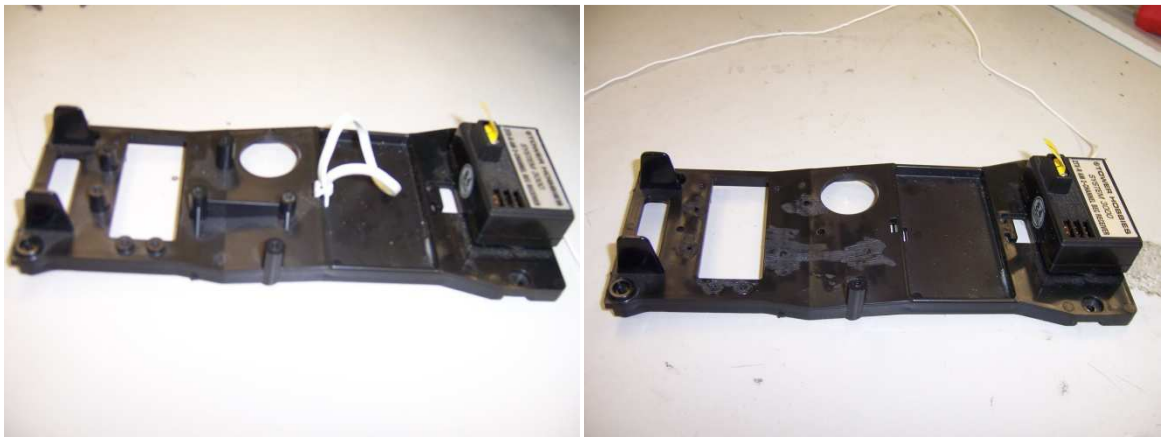

Figure 3.3: Comparison of harness before and after modification

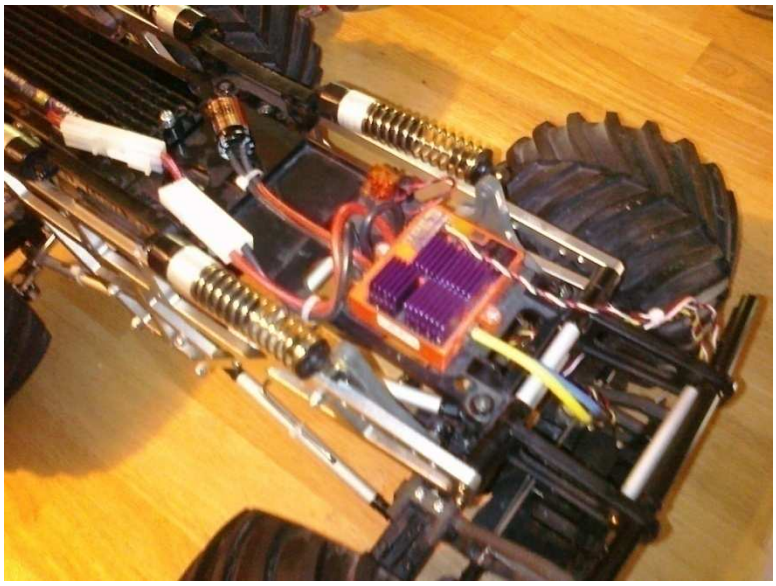

Figure 3.4: Completed ESC installation

One of the negatives to the vehicle chosen was that the original turning radius of approximately 0.95 m was very poor for indoor navigation. However, this was easily remedied by installing an additional steering servo on the vehicle's rear wheels. The additional steering servo was not included with the vehicle kit, but mounting brackets and linkages to facilitate the conversion to four-wheel steering were provided. To accomplish this conversion, another servo motor identical to the front steering servo was purchased and installed according to the kit instructions. This modification reduced the turning radius to 0.45 m which is more reasonable for indoor navigation. This modification can be seen in Figure 3.5.

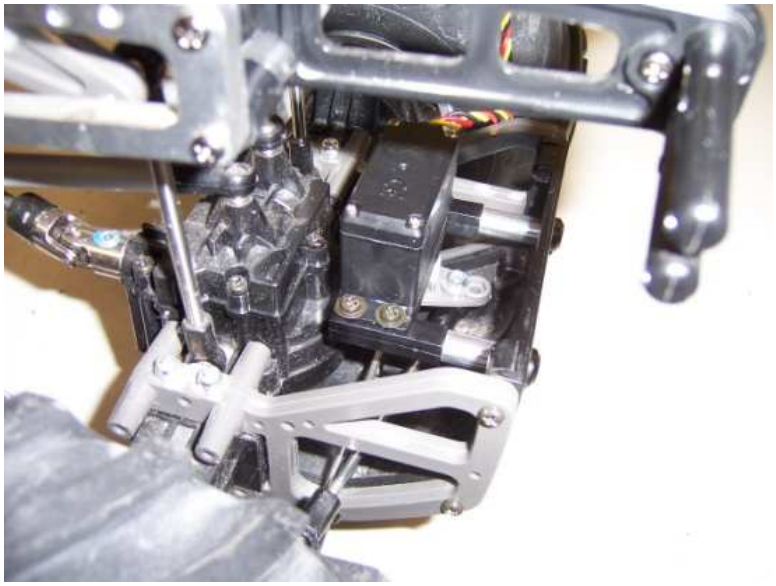

Figure 3.5: Additional servo installed for rear steering

The final two physical modifications made to the vehicle were to improve its suspension. Early tests proved that the vehicle was entirely capable of supporting and maneuvering with a simulated payload of 3 kg attached to the top of the vehicle. However, it was noticed that the suspension sagged under the weight of the payload. To help combat this problem, stiffer springs were installed in addition to spacer rings which were machined to further stiffen the effect of the springs. Also, the original *damping oil* in the shocks was replaced with a more viscous oil to help the vehicle compensate for the added weight of the payload.

3.3 COMPUTER SYSTEM

3.3.1 REQUIREMENTS

The research platform must be equipped with some type of computer system in order to operate autonomously. Since the computer system for the rover is to be included onboard, it must be small enough as to not overburden the capacity of the vehicle. The tested capacity of the vehicle

is approximately 3 kg, so the computer system should weigh around 2.5 kg or less so as to leave room for any additional sensors or actuators which may be needed later.

The onboard computer must be powerful enough to handle a wide variety of tasks. Many of the future experiments anticipated for the rover will involve processor intensive mapping and vision control algorithms. As such, the processor for the computer must be strong enough to handle these tasks without impairing the real-time execution of the motor and speed control.

Another key requirement of the rover's onboard computer is that it must be expandable. Careful consideration will be taken initially to predict the future computing needs of the project, but it is possible that future researchers may need to expand the platform. This said, the number and types of ports on the onboard computer should not only satisfy the needs of the current sensors and actuators, but it should also provide a range of common expansion ports to provide maximum flexibility. The system must also possess wireless networking capabilities to allow communication with a *base station* in order to receive mission directives and to send back status and collected data.

Finally, the onboard computer must operate from its own power supply. The NiMH battery installed on the rover is for powering the drive motors and steering servos only. Due to the nature of the drive train motors, large back currents could be sent through electrical system and seriously damage the onboard computer system. Also, the battery life of the drive battery is already at the low end of its acceptable limit; any extra load would destroy the charge range of the vehicle. The onboard computer must have at least the same charge range, one hour, as the vehicle. Extra battery life would obviously be a benefit to the overall usability of the system.

3.3.2 SELECTION

Several options were considered when selecting an appropriate on-board computer system for this project. Among the types of computers considered for the rover's onboard computer were a PC-104, a netbook, and a mini-ITX.

The first type of computer system considered for the rover was an embedded computer based around PC/104 components. PC/104 defines a set of industry standards for component form-factor, connection, and communication. Components adhering to the PC/104 standard have a small form factor of 96mm by 90mm. A PC/104 system is constructed by stacking desired components on top of one another using standoffs on the components' corner mounting holes. The standard's small form-factor, as well as its stacking capabilities, allow for the computer system to be very small, lightweight, and portable. The stacking ability also allows the system to be more rugged than a traditional system.

In addition to the small footprint provided by the PC/104, the nature of its design allows for a highly modularized approach to constructing a system. When building a system based on PC/104 components, one must first select a motherboard which will act as the central processor and hardware controller for the other accessory boards. This board will contain the actual CPU for the system as well as the connectors and interfaces for common peripherals such as a keyboard, mouse, and display. This board must also interface any additional boards desired. Once the main board is selected, peripheral boards may be obtained to provide a vast array of functions such as data acquisition, network communication, motor control, and GPS.

The small package size of the PC/104, as well as the high level of customizability afforded by available peripheral boards, makes PC/104 systems a popular choice for embedded systems in industrial and robotic applications. However, there were several reasons why it was eliminated as a

candidate for this particular application. The first reason was that the processors integrated into the motherboards were relatively slow when compared to those featured in traditional laptops and other possible solutions; most of the processors found on the PC/104 motherboards had no multi-threading capabilities and were at least an order of magnitude slower than their competing solutions. It was feared that the processors equipped on the PC/104 motherboards may not be capable of some of the more intense tasks such as video processing which may be required of the rover. Another reason why a PC/104 based solution was not chosen was that the cost of such a system was considerably more expensive than the other available options.

The second option explored for the rover's computer system was an ordinary laptop. The size, power, and capabilities of laptops vary greatly by manufacturer and price range. Size was the first consideration when selecting a possible laptop to serve as the rover's onboard computer system since it was imperative that the computer be small enough to be attached to the rover and not overburden the suspension or drive motors. This requirement limited the possible computers to a small laptop or netbook. The option of using a laptop for the onboard computer presented several strong advantages over a PC/104 system as well as a few strong disadvantages.

One major advantage provided by using a laptop is that the computer comes nearly ready to use out-of-the-box. Modifications would still have to be made to the computer in order to make it suitable for the rover, but these changes would be limited mostly to the operating system and attaching accessories via available ports. Another strong advantage of using a laptop is that the keyboard, mouse, and display are all integrated. Thus, modifications to programming or control variables could be as easy as opening the lid of the laptop and making the desired changes. Even if programming changes were not made *on-the-fly* via the laptop, the convenience provided by not having to continually reconnect the computer to an external keyboard, mouse, and display would

greatly increase the efficiency at which the rover's controllers could be tuned. Finally, the price of a suitable laptop was much cheaper than that of an equivalent PC/104 system.

A small laptop would provide several large advantages over a PC/104 system; however, it would also have several strong disadvantages to overcome. The processing speeds of the possible laptops selected were much quicker than the PC/104 systems compared. However, the smaller laptops and netbooks compared were still considerably underpowered when compared to larger laptops and desktops. Although this would likely not be an issue for most of the control schemes planned in the scope of this project, it was desired to keep the onboard computer as powerful as practical in order to facilitate future computationally intensive experiments. The largest disadvantage to using a laptop was the issue of expandability. When compared to the other options available, it was by far the least expandable. At this point in the selection phase, a decision could not be made between the laptop option and the Mini-ITX option to be discussed next. A search was conducted for the most suitable laptop that will later be compared the most suitable Mini-ITX option.

The final option considered for the onboard computer was that of a Mini-ITX based system. Like PC/104, Mini-ITX simply refers to an industry standard form-factor. This compact form-factor is also very versatile and is used in a wide variety of embedded systems. However, there are several distinct differences between the two standards. The first major difference is the size of the boards. The Mini-ITX has outer dimensions of 170mm by 170mm as compared to PC/104's outer dimensions of 96mm by 90mm. Also, whereas PC/104 is a standard for a whole variety of different boards, Mini-ITX applies only to the motherboard. A Mini-ITX system does not layer components as in a PC/104 system. Instead the other parts and accessories for a Mini-ITX system attach via other standard ports and mount in various configurations.

For this application, the Mini-ITX form-factor has many strong advantages over the PC/104 form-factor and the laptop options. First, the Mini-ITX is rather larger than the PC/104 board, but it is still small enough to fit on the rover. This extra size allows the Mini-ITX to contain considerably more processing power and features. On PC/104 systems, and most ultra-light laptops, the processor is integrated into the motherboard and cannot be changed. However, on a Mini-ITX motherboard, the processor is not included. Instead, there is an empty socket into which any processor matching the socket, and architecture type, may be connected. Also, in relation to available ports and interfaces, the Mini-ITX is very similar to a traditional desktop computer's motherboard. Instead of header pins and ribbon cables that connect PC/104 components together, ordinary connections are present. This means that instead of having to use specialized adapters for all of the connections, unmodified versions can be used. This same concept is applicable to onboard ports; most Mini-ITX motherboards contain traditional PCI, PCI Express, or other common ports. Using this type of board, it is possible to get near-desktop performance out of a mobile package.

The option of using a Mini-ITX system also presents several disadvantages. Since such a system must be assembled from individual components, a custom enclosure must be constructed. Also, no power supply or battery is integrated into the system. Thus, a separate battery and power adapter must be installed in order to power the system. This step is not a problem, but it does add to the overall complexity of developing the system. With a laptop, it is possible to charge to computer while using it; however, a special circuit would be needed to accomplish this using the Mini-ITX system. Finally, there are no input or output devices integrated into the Mini-ITX. This is not a problem for autonomous experiments, but does present a mild inconvenience for making quick adjustments to the programming.

When selecting the best option for the onboard computer, the PC/104 option was easily eliminated due to its inferior processing power and expensive price tag. However, the choice

between an ultra-light laptop and a Mini-ITX system was not as simple. To help decide between the two, a system was selected for each case according to the above mentioned requirements. These candidates were then compared side by side to select the best option. Table 3.2 provides an overview of the features available with the best option found for each of the options.

Table 3.2: Comparison of the selected Mini-ITX computer and ultralight laptop

	Mini-ITX Computer	HP Mini-Note
CPU	2.4 GHz Core 2 Duo Mobile 800 MHz, 3 MB L2 Cache	1.60 GHz VIA C7 800 MHz, 128 KB L2 Cache
Storage	80 GB Hard Drive	120 GB Hard Drive
Memory	4 GB DDR2 667	2 GB DDR2 667
Video Card	Integrated	VIA Chrome 9
Video Memory	Integrated	Shared
Operating System	None	Windows Vista Business
Serial Ports	2	2 (with ExpressCard addon)
Parallel Ports	1	0
USB	8	2
GPIO	Yes	No
ExpressCard	0	1
PCI	1	0
PCI Express	1	0
Mini PCI-E	1	0
Bootable Compact Flash	Yes	No
CD/DVD Drive	No	Yes
LAN	Yes	Yes
WLAN	Yes (with add-on card)	Yes
Bluetooth	No	Yes
Audio	Yes	Yes
Webcam	No	Yes
Display	None	8.9” 1280x768
Price	**$823.97**	**$1008.99**

In the end, the Mini-ITX option was chosen for the rover’s onboard computer. This was because it afforded the most processing power and expandability for the money.

3.3.3 COMPONENTS

3.3.3.1 MINI-ITX MOTHERBOARD

The Mini-ITX motherboard selected for the rover's onboard computer was the MSI MS-9803. This motherboard accepts a standard Intel mobile dual-core processor. Also, it features multiple ports for future expansion. This motherboard can be seen in Figure 3.6 below.

Figure 3.6: MSI MS-9803 motherboard

3.3.3.2 STANDARD COMPONENTS

Since the computer system was chosen to be built around a Mini-ITX motherboard, ordinary desktop PC components could be used.

The processor chosen was an Intel T8300 Core 2 Duo. This processor features two processing cores and a clock speed of 2.4 GHz. The dual processing cores allows for tasks to be parallelized for greater efficiency. At the time of purchasing this processor, it was at the *knee* of the mobile processor price-to-performance curve; the only processors available for the chosen

motherboard that were faster than the one selected were considerably more expensive but only marginally faster.

Several options were explored when choosing internal storage for the system. The two main concerns when choosing a storage option where speed and durability. The initial thought was to use a bootable compact flash card, which has no moving parts. This is a great feature from a durability standpoint. However, compact flash storage has a limited number of read/write cycles. This is normally not an issue for use in devices such as digital cameras since the number of cycles is generally small, but an operating system writes to a disk much more frequently and this could present a problem. The other most viable option was to use an ordinary internal hard drive. The storage capacities available in this type of drive are much greater than those found in other equally priced solid-state drives. In addition, disc access times are much quicker than their solid-state alternatives. The only downfall to this type of storage is that it contains moving parts which could be damaged by physical shock.

The storage option chosen was a 60 GB 2.5" SATA hard drive. This is the same type of drive commonly used in laptop computers. As such, it is designed to be more rugged and consume less power than a larger desktop hard drive. One should note that since the original purchasing of the computer, solid-state hard drives have become significantly cheaper and more advanced.

Other components selected for the computer system include 4 GB of RAM and an Edimax PCI Wireless networking card. The amount of RAM chosen was the maximum supported by the motherboard.

3.3.3.3 POWER SUPPLY

The power supply chosen was the *Pico PSU-120 Watt Wide-Input Voltage DC to DC Convertor.* This power supply is built into a small circuit that is located on a standard 20-pin ATX

connector. This power supply has connections available for the motherboard, one SATA device, and one 4-pin Molex which can be used to supply enclosure fans, disc drives, or any other device requiring five or twelve volts. It accepts a DC voltage source between 12V to 32V, thus providing the flexibility to accept many different types of battery sources at various levels of charge. This power supply can be below in Figure 3.7.

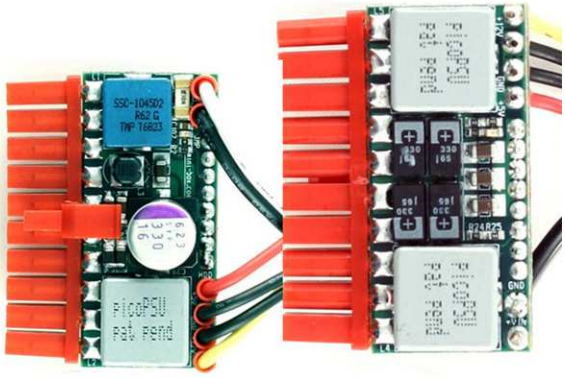

Figure 3.7: Pico PSU-120 watt wide-input voltage DC to DC converter

3.3.3.4 BATTERY PACK

A Powerizer 14.8V 5700 mAh lithium-ion battery was selected for the onboard computer. A lithium-ion battery was selected because they have a high energy density and do not experience memory effects. The chosen battery can be seen in Figure 3.8.

Figure 3.8: Powerizer 14.8V 5700 mAh lithium-ion battery

3.3.3.5 ENCLOSURE

A Mini-ITX enclosure that was compact enough to be mounted on the rover, while still having room for the computer, battery, and controller boards, could not be found when searching for components for the rover's computer system. As such, a custom enclosure was fabricated to meet this need.

The enclosure was created from aluminum sheet metal. Two 40mm case fans were installed in the enclosure to provide cooling for the various components. These fans were wired into the main supply for the computer so that they will run any time the computer is powered on. Also, power and reset buttons were installed on the front of the enclosure, and a set of charging ports were installed on the side of the enclosure. Finally, a switch was installed on the rear of the enclosure to allow for the external power to either power the computer directly or charge its battery. Figure 3.9 provides an overview of the internal layout of the system. Figure 3.10 shows the charging ports and power switch. Figure 3.11 provides an overall view of the completed enclosure with the computer system installed.

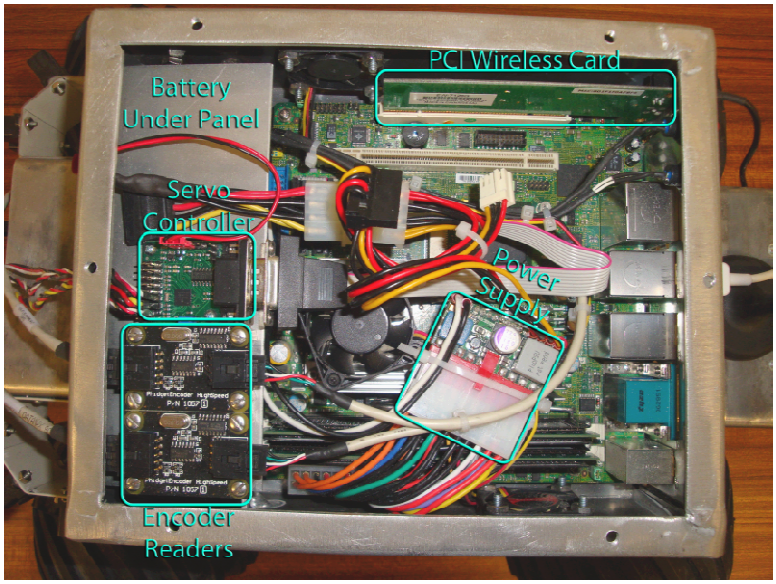

Figure 3.9: Overview of internal component layout

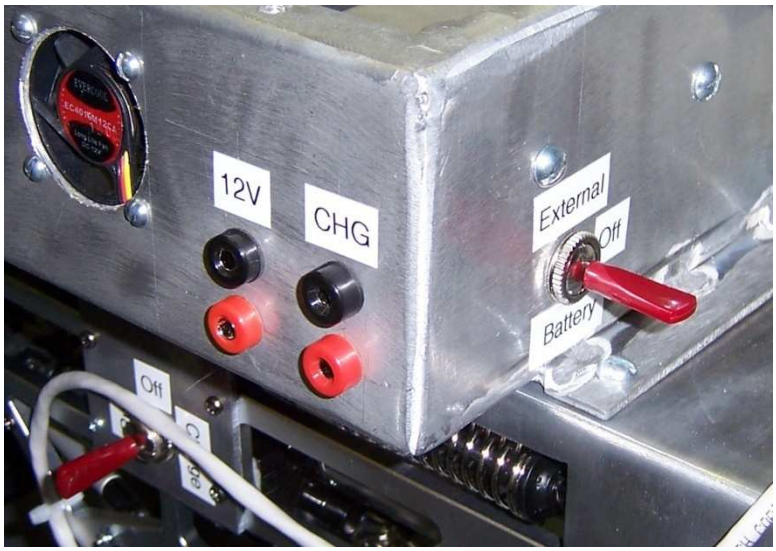

Figure 3.10: Charging port and power selector switch

Figure 3.11: Completed enclosure with computer installed

3.3.4 MOUNTING

The onboard computer was mounted on top of the existing rover frame. An aluminum plate was constructed and attached to the rover in order to provide a flat surface for mounting the

computer. Then the computer and its wireless antennae were bolted to the plate. The completed installation of the computer can be seen in Figure 3.12.

Figure 3.12: Computer enclosure mounted to rover platform

One issue created by the installation of the mounting plate is that directly accessing the drive battery plug for charging the battery or disconnecting it when not in use became impossible. To remedy this problem, a port and a three-position switch were installed on the mounting plate to allow the battery charger to be connected to the battery or the battery to be disconnected without the need for disassembly. This modification can be seen in Figure 3.13.

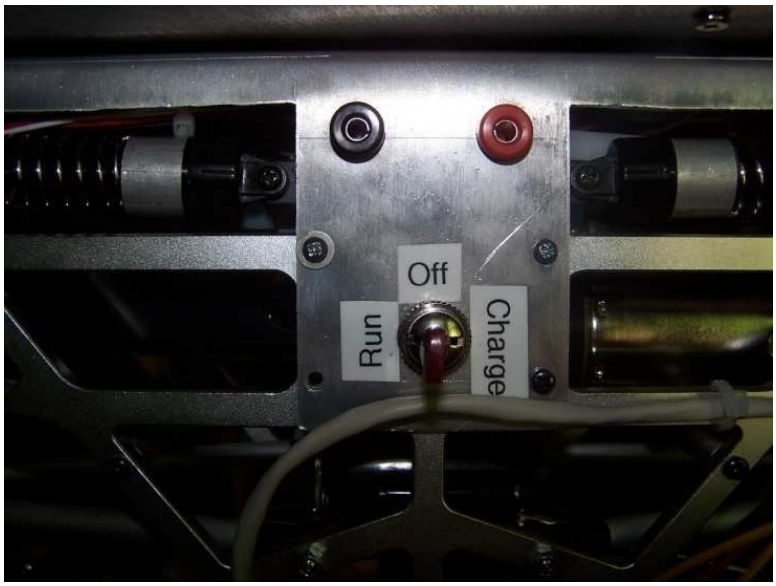

Figure 3.13: Charging port for drive battery

3.4 SENSORS AND CONTROL BOARDS

Several different sensors and control boards were installed on the rover to facilitate autonomous navigation. A block diagram of the system layout can be seen in Figure 3.14. The individual components will be discussed in the remainder of this section.

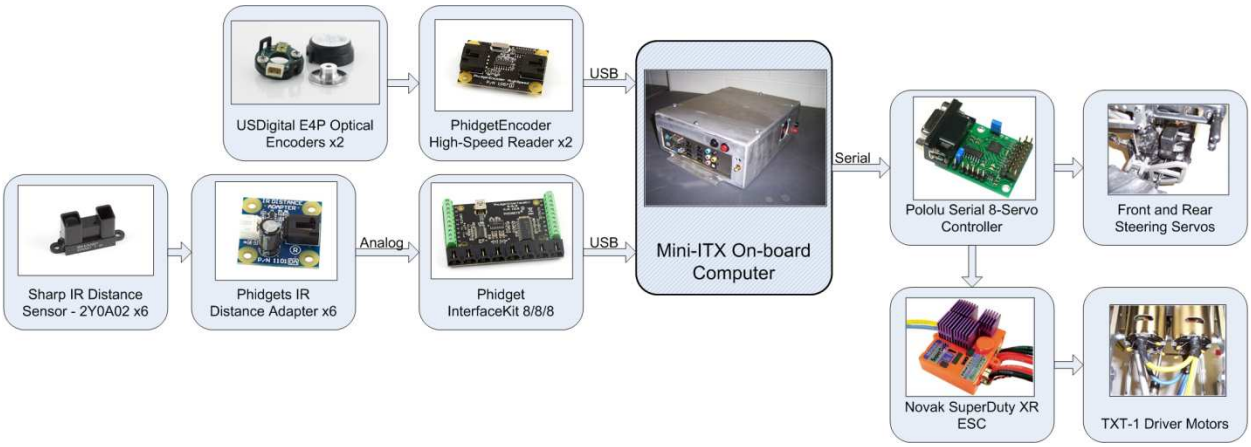

Figure 3.14: Block diagram of rover sensor and actuator system

3.4.1 ENCODERS

3.4.1.1 SELECTION

The rover must be capable of estimating its position, orientation, and translational and rotational velocities in order to operate autonomously. One way of accomplishing this is to monitor the angular velocities of the wheels and calculate the required values based upon this information. The state of the vehicle can be calculated using the angular velocities of either the front or rear set of wheels. For this application, it was chosen to monitor the angular velocity of the front wheels.

The most practical way of measuring the angular velocity of the wheels was to install optical encoders. An optical encoder uses a light source and an array of detectors to measure the position of a rotating shaft, which it then converts to a digital or analog signal. Quadrature type encoders were selected because they allow both the absolute angular position and the direction of motion of the drive shafts to be measured. By using one quadrature encoder for each of the front wheels, their angular velocities can be calculated which can then be used to calculate the position and velocity state of the rover.

When selecting the encoders for the rover, the main problem was finding a set of encoders small enough to be mounted in the space constraints of the rover. The front and rear drive axles are covered by the enclosure for the differentials, thus the only location available for mounting the encoders was inside the wheel hubs. However, the wheel hubs had very limited space. As such, the encoders were required to be very compact in order to fit the space constraints. Figure 3.15 provides a view of the mounting space available within the unmodified wheel hub assembly.

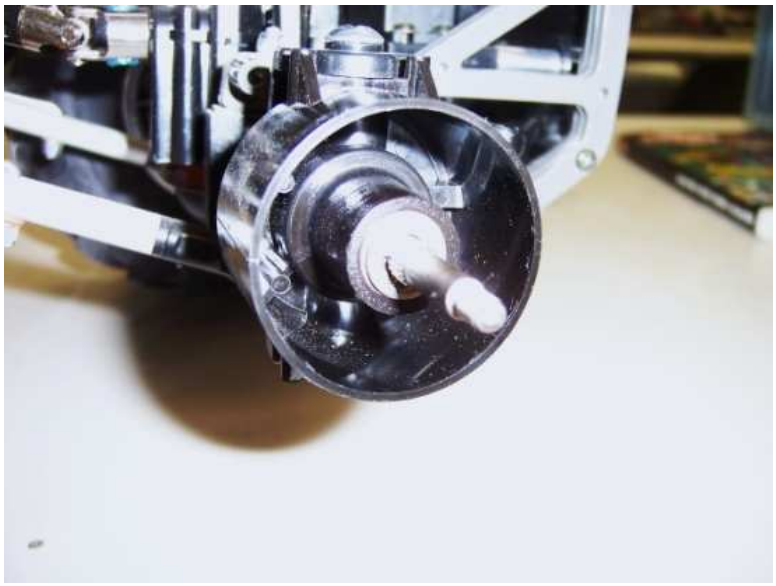

Figure 3.15: Original wheel hub configuration

This space constraint proved much more difficult than originally anticipated as most encoders available commercially in small quantities are much too large for the space available. After extensive searching, the E4P Miniature Shaft-Mount Optical Encoder by USDigital was selected. The size of this encoder still was not perfect for the application but could work with modification to the wheel hub assembly. This encoder features an encoder disk with 256 divisions, but since the encoder is quadrature in type, the actual resolution is four times this, or 1024 counts per revolution. The chosen encoder can be seen in Figure 3.16.

Figure 3.16: USDigital E4P miniature shaft-mount optical encoder

3.4.1.2 MOUNTING

The wheel hub was modified and a mounting bracket was constructed in order to mount the encoders. First, the original bearing supports and surrounding material were removed from the wheel hub. Then a mount was machined from aluminum. This mount consists of a cavity in which the encoder can rest as well as replacement bearings for the ones that were removed to make room for the encoder. These modifications can be summarized by the CAD drawing in Figure 3.17.

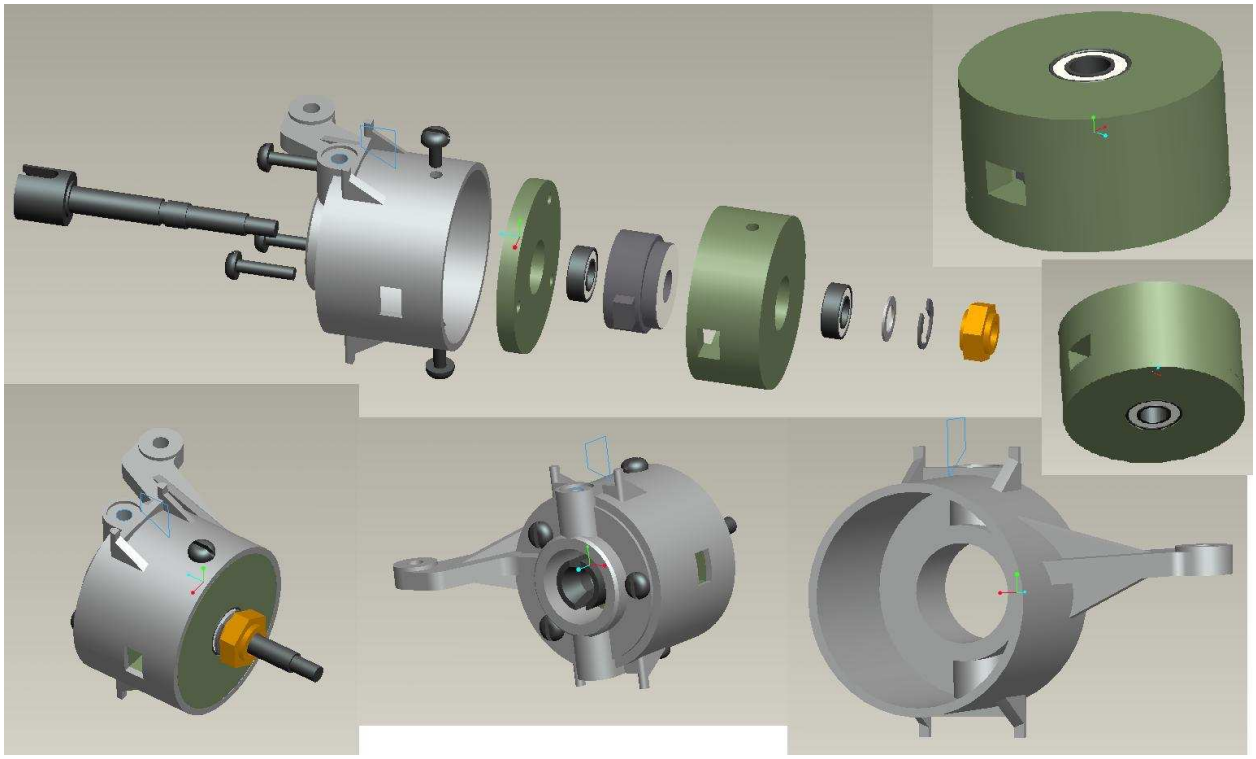

Figure 3.17: Summarized drawing of wheel hub modification for encoder installation

The completed mounts can be seen in Figure 3.18 and Figure 3.19. The wheel hub assembly with the encoders and their mounts installed can be seen in Figure 3.20. Finally, the completed assemblies were mounted back onto the vehicle without any further modifications.

Figure 3.18: Bottom replacement bearing mount and encoder support structure

Figure 3.19: Top replacement bearing mount and encoder support structure

Figure 3.20: Completed wheel hub and encoder assembly

3.4.1.3 ENCODER CONTROLLER

The signals from the encoders must be translated before meaningful position or speed data can be read by the computer. A set of *PhidgetEncoder High Speed USB Encoder Readers* were chosen for this task. These encoder controllers are small, lightweight, and connect to the computer using a standard USB port. Finally, they interface easily with either Windows or Linux using the API provided by the manufacturer. These encoder controllers were mounted inside the computer enclosure and attached using a USB header on the motherboard. The chosen encoder controller can be seen in Figure 3.21.

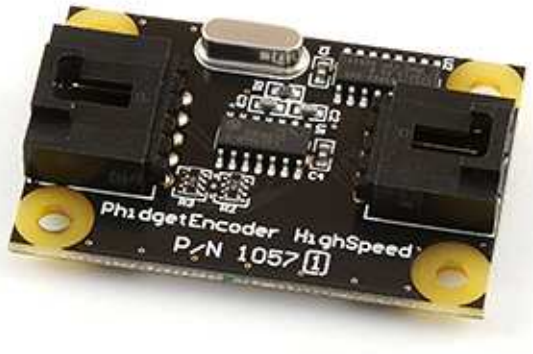

Figure 3.21: PhidgetEncoder high speed USB encoder reader

3.4.2 INFRARED PROXIMITY SENSORS

Infrared (IR) proximity sensors were installed on the rover to facilitate obstacle avoidance. The sensor chosen for this task was the *Sharp Distance Sensor 2Y0A02*. This sensor is capable of measuring distances between 0.2 m and 1.5 m. It produces an analog voltage inversely proportional to the distance measured. Six of these sensors are mounted to the front of the rover in two arrays of three which are offset by 45°. The mounting arrangement will be discussed in further detail in Chapter 5. This sensor can be seen in Figure 3.22. The array arrangement of the mounted sensors can be seen in Figure 3.23.

Figure 3.22: Sharp distance sensor 2Y0A02

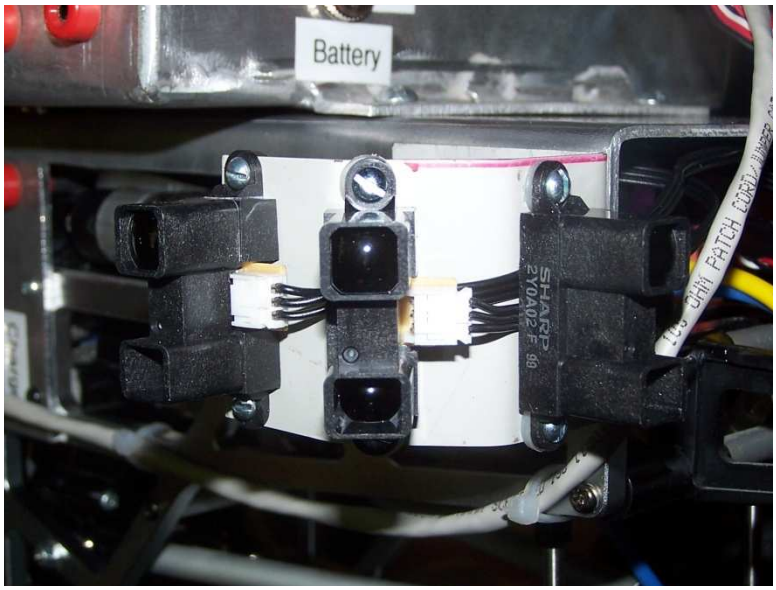

Figure 3.23: Array of three IR proximity sensors mounted to rover

The output from the distance sensor can be read directly by a typical data acquisition system. However, *Phidget IR Distance Adapters* are used to help condition the signal. This board helps to protect the data acquisition board, which will be discussed in the next section, from excessive current draw from the six distance sensors. These boards are mounted under the computer mounting panel. This adapter can be seen in Figure 3.24.

Figure 3.24: Phidgets IR distance adapter

3.4.3 DATA ACQUISITION

A data acquisition (DAQ) board with at least six analog inputs was needed to read the rover's IR proximity sensors. The one used on the rover is the *PhidgetInterfaceKit 8/8/8*. This DAQ board features eight 10-bit analog inputs, eight digital inputs, and eight digital outputs. This satisfies the requirement for the IR proximity sensors and allows for some future expansion. Additionally, it is manufactured by the same company as the rover's encoder readers. As such, it also features a well-equipped API that can be used in either Windows or Linux operating systems. This simplified interfacing the control boards with the on-board computer since very similar code could be used for both types. This DAQ board can be seen in Figure 3.25.

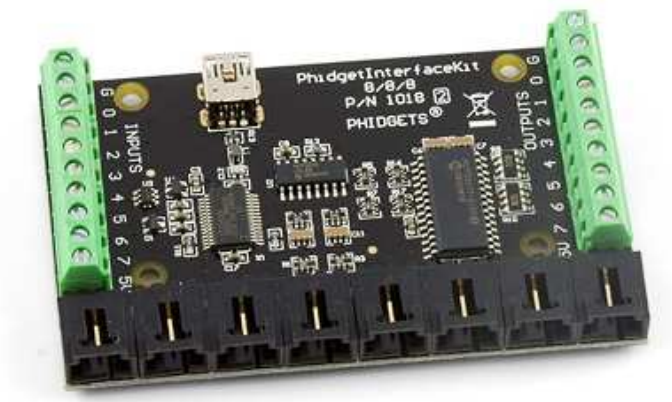

Figure 3.25: PhidgetInterfaceKit 8/8/8

The DAQ board is mounted under the computer mounting panel along with the IR distance adapters. This mounting panel can be seen in Figure 3.26 .

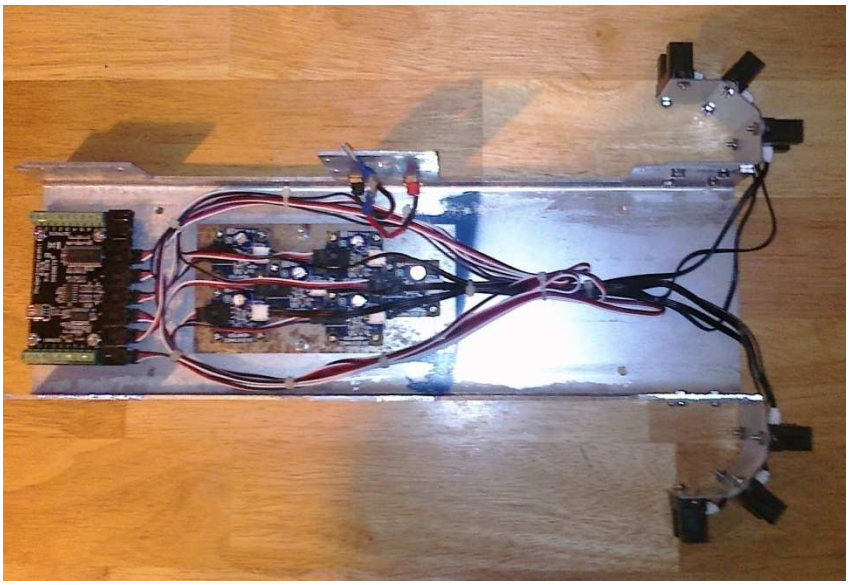

Figure 3.26: Bottom view of mounting panel with DAQ board, IR distance adapters, and IR distance sensors

3.4.4 SERVO CONTROLLER

The rover was equipped with a *Pololu Serial 8-Servo Controller* in order to interface the rover's steering servos and electronic speed controller. This servo controller connects to the computer using a standard RS-232 serial port and is capable of controlling eight RC servos simultaneously. Since the controller uses a serial port to communicate with the computer, it can work under any operating system. The controller was mounted inside the computer enclosure next to the encoder readers and attached using a serial header on the motherboard. The servo controller can be seen in Figure 3.27. A complete list of hardware sources and prices can be found in Appendix A.

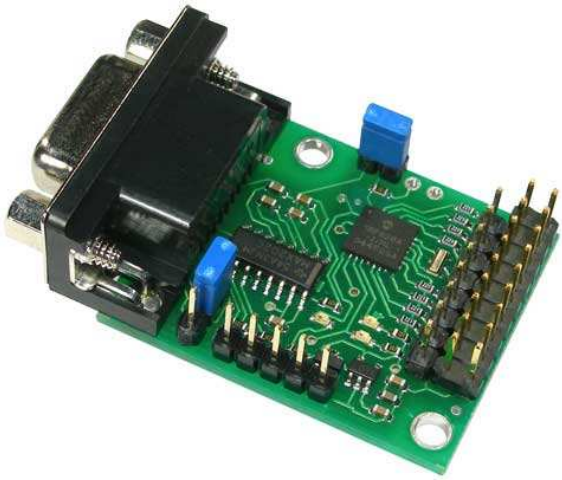

Figure 3.27: Pololu serial 8-Servo controller

CHAPTER 4 - SOFTWARE

In addition to the hardware aspect of this project, a complete software suite was developed to interact with the rover. This software suite allows control laws to be executed in real-time on the rover's various hardware systems. At the bottom-most layer of the software suite is the real-time operating system (OS). Control schemes are built to run in this real-time OS using Matlab® Simulink® Real-Time Workshop (RTW). A set of Simulink® S-function blocks were developed using the C programming language to interact with the system's sensors and actuators. This chapter describes the RTOS installation and calibration, and the custom Simulink® block-set used for real-time code-generation.

4.1 OPERATING SYSTEM

4.1.1 UBUNTU 9.04

The onboard computer did not have an OS pre-installed, so the entire software suite was developed, starting with the installation of an operating system. As mentioned previously, the computer system should execute its control schemes in real-time. Thus, several different operating systems capable of real-time code execution were available. The chosen OS for this project was RTAI Linux. This was discussed in Section 2.4 and will be briefly discussed in the next section.

The chosen real-time environment, RTAI Linux, is not actually a complete OS. It consists of a patch for a Linux kernel as well as a library of functions and services that enable real-time operation. Consequently, nearly any Linux distribution can be used as a base for RTAI.

The base Linux OS chosen for the rover was Ubuntu 9.04 32-bit. Ubuntu is currently one of the most popular Linux distributions. It offers most of the same functionalities as a typical

Windows® OS such as a graphical window manager, standard hardware drivers, network support, and a file system. In terms of Linux distributions, it is one of the more *full-featured* varieties. Much of the functionality provided by Ubuntu will not be directly needed by the rover's real-time control schemes. However, for testing and development purposes, this distribution affords the user with many conveniences. One such convenience is the ability to generate real-time control schemes on the rover's on-board computer. Many real-time systems are built on very *lean* OS's and require control schemes to be written on a separate machine (they do not possess the graphical capabilities required to use Simulink® or RTW). A leaner OS may need to be installed in the future as control schemes become more complex and computational needs become greater, but for the scope of this project, Ubuntu performed well.

Installation of Ubuntu was straight-forward. An installation disk was downloaded from Ubuntu's website and burned to a CD. A CD drive was temporarily connected to the on-board computer. Then the computer was booted with the installation CD, and the on-screen instructions were followed until the installation was complete. Installation may also be performed from a USB flash drive to avoid the necessity of temporarily connecting a CD drive to the rover.

4.1.2 RTAI

The RTAI environment allows specially-written code to be executed in real-time. It consists of a Linux kernel patch which introduces a real-time hardware abstraction layer and a set of modules and libraries needed to write and execute real-time code.

4.1.2.1 INSTALLATION

The installation procedure provided here is a simplified and customized version of the procedure provided by (46). The installation procedure is intentionally provided in detail since, at

the time of this project, most of the instructions found on the various RTAI community websites were outdated or unreliable.

Once Ubuntu was installed, the built-in update manager was run to ensure that the base OS was up-to-date. Next, a small number of *packages* needed for compiling the custom Linux kernel, RTAI environment, and QRtaiLab (which will be discussed in detail in Section 4.2.1) were installed. The following command was executed in a Linux terminal (command line):

```
# sudo apt-get install cvs subversion build-essential kernel-package linux-source libncurses5-dev libtool automake libqt4-dev libqwt5-qt4-dev
```

Next, the source codes for a *vanilla* Linux kernel, RTAI, and QRtaiLab were downloaded and extracted. A vanilla Linux kernel is essentially a generic kernel without any distribution specific customization. The following commands were executed:

```
# cd /usr/src
# sudo wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.28.7.tar.bz2
# sudo tar xjvf linux-2.6.28.7.tar.bz2
# sudo ln -s linux-2.6.28.7 linux
# sudo wget --no-check-certificate https://www.rtai.org/RTAI/rtai-3.7.tar.bz2
# sudo tar xjvf rtai-3.7.tar.bz2
# sudo ln -s rtai-3.7 rtai
# cd /opt
# sudo wget http://downloads.sourceforge.net/qrtailab/QRtaiLab-0.1.12.tar.gz
# sudo tar xvzf QRtaiLab-0.1.12.tar.gz
```

After the relevant source codes were downloaded and extracted to their proper locations, the Linux kernel was patched. This was accomplished by using the following commands:

```
# cd /usr/src/linux
# sudo su
# patch -p1 < /usr/src/rtai/base/arch/x86/patches/hal-linux-2.6.28.7-x86-2.2-06.patch
# sudo make menuconfig
```

This will bring up a menu for configuring the custom kernel. Under *General setup, Local version - append to kernel release* was set to *-rtai*. Under *Processor type and features*, *Processor family* was set to *Core 2/newer Xeon*, *Maximum number of CPUs* was set to 2, *Symmetric multi-processing support* was enabled, *Preemption Model* was set to *Preemptible Kernel (Low-Latency Desktop)*, and *Timer Frequency* was set to 100 Hz. Under *Power management and ACPI options*, *Power Management Support* was disabled. Then the kernel configuration was saved, compiled, and installed. This step took approximately two hours. This was done by executing the following commands:

```
#  cd /usr/src/linux
#  sudo make-kpkg clean
#  sudo make-kpkg --initrd kernel_image kernel_headers
#  cd /usr/src
#  sudo dpkg -i linux-headers-2.6.28.7-rtai_2.6.28.9-rtai-10.00.Custom_i386.deb
#  sudo dpkg -i linux-image-2.6.28.7-rtai_2.6.28.9-rtai-10.00.Custom_i386.deb
```

Once the installation was complete, the computer was rebooted into the custom kernel. This was accomplished by simply rebooting the computer and selecting the kernel name with *–rtai* appended to the name in the boot menu. Next the RTAI environment was installed using the following commands:

```
#  cd /usr/src/rtai
#  sudo make menuconfig
```

This will once again bring up a configuration menu. The installation directory should be listed as /usr/realtime and kernel source should be listed as /usr/src/linux. Also, the correct number of CPU's should be selected under the *Machine* menu. The rover's on-board computer CPU has two cores; as such, two was entered here. Then RTAI was compiled and installed as follows:

```
#  sudo make
#  sudo make install
#  sudo sed -i 's/\(PATH=\"\)/\1\/usr\/realtime\/bin:/' /etc/environment
```

Once again, the computer was rebooted. Finally, QRtaiLab was installed. This was accomplished by executing the following command:

```
#  cd /opt/qrtailab
```

It was necessary to uncomment the line containing "#DEFINES += _RTAI_3_7_", in the file *qrtailab.config*. Then the following commands were executed:

```
#  sudo qmake-qt4
#  sudo make
#  sudo make install
```

4.1.2.2 VERIFICATION AND CALIBRATION

Once RTAI was installed, it was tested to verify that it was operating in real-time. This was accomplished by running the six tests in the *testsuite* that is included as part of RTAI. These tests are located in /usr/realtime/testsuite/. These tests were performed by navigating to the bottom-most directories and executing the *run* scripts. It was found by running these tests that the system had a maximum latency of approximately 890 nanoseconds. Since this latency was well within the acceptable range, no additional calibration was needed.

4.1.2.3 MATLAB/SIMULINK RTAI INTEGRATION

After the RTAI environment was installed and calibrated, Matlab® was installed with Simulink® and RTW by following the manufacturer's instructions. Once Matlab® was installed, RTW was configured to work with the RTAI installation. This was done by copying the rtai-lab folder

from the RTAI source directory into the RTW folder located in the Matlab® root directory (usr/local/matlab in this case). The following commands were executed:

```
#  cd /usr/local/matlab/rtw/c
#  mkdir rtai
#  cp -R /usr/src/rtai/rtai-lab/matlab ./
```

Next, Matlab® was configured to use the system's compiler which was installed during the RTAI installation. This was accomplished by issuing the following command in the Matlab® command window and following the instructions:

```
≫ mex -setup
```

With the compiler setup complete, the integration of RTAI into RTW was performed. This was accomplished by navigating to /usr/local/matlab/rtw/c/rtai and executing the setup script from within Matlab®. This step adds the correct RTAI target to the RTW dialog in Simulink®. It also adds *RTAI Devices* to Simulink®, which are used by the graphical portion of RTAI Lab. This will be discussed in further detail in Section 4.2.1.

At this point, the integration of RTAI and RTW was complete. This installation was verified by compiling the test model provided with RTAI by using RTW. To compile a real-time control scheme, the Simulink® model is first constructed or loaded. From within the configuration parameters dialog, the model's time step should be set to a fixed-step with no continuous states. Also, the sampling rate should be specified. Under the *Real-Time Workshop* pane, *System target file* should be set to rtai.tlc. If no custom S-Functions are being used, the *Build* button is pressed and the executable is generated. Otherwise, the *Generate code only* option should be checked and the *Generate code* button is pressed. If only code is generated, the S-Function files must be moved into the _rtai directory as described in Section 4.2.2 and the following command must be executed from within that directory:

```
#  make –f projectname.mk
```

RTAI contains a set of *loadable kernel modules* which must be loaded before any real-time code can be executed. These modules contain the libraries which extend the functionality of the kernel to operate in real-time. Through the use of loadable kernel modules, functionality can be added or removed from a kernel which is already running, thus allowing memory to be freed when the functionality is not needed. The modules needed for real-time code execution are loaded by using the following commands:

```
#  sync
#  sudo insmod /usr/realtime/modules/rtai_hal.ko
#  sudo insmod /usr/realtime/modules/rtai_lxrt.ko
#  sudo insmod /usr/realtime/modules/rtai_fifos.ko
#  sudo insmod /usr/realtime/modules/rtai_sem.ko
#  sudo insmod /usr/realtime/modules/rtai_mbx.ko
#  sudo insmod /usr/realtime/modules/rtai_msg.ko
#  sudo insmod /usr/realtime/modules/rtai_serial.ko
#  sudo insmod /usr/realtime/modules/rtai_netrpc.ko ThisNode="127.0.0.1"
#  sync
```

Finally, the control scheme can be executed from within its directory. The program can be run with the *–u* option to display the various run-time options available. Typical options used are: *–v,* which displays *verbose* output; *-f,* which, when followed by a number, runs the control scheme for that period of time (in seconds); *-w,* which initializes the program, but does not run it until instructed to do so by an RTAI Lab graphical user interface (GUI); and *–o,* which runs the control scheme in *one-shot* mode (discussed in the next section).

4.1.3 LIMITATIONS AND CONSIDERATIONS

On a real-time computer system, strict timing constraints are given to tasks to indicate when they should execute and how long they may take. This is referred to as scheduling. In the

RTAI environment, such tasks may be scheduled in two ways. The first method is *periodic* mode. A periodic tasks is scheduled to execute at a predefined interval. The computer's built-in timers are used to automatically reload the timer to the same value each time it is triggered. A periodic task is ideal for something such as the control schemes implemented on the rover since discrete time-steps are used by the Simulink® models. In RTAI, periodic tasks are limited by frequency of the Linux timer. The Linux timer may be configured to 1000 Hz, 250 Hz, or 100 Hz during the compilation of the custom kernel, and the periodic tasks should be set to a multiple of the Linux timer. However, the rover's motor controller and encoder readers are only capable of sampling at a rate of 50 Hz. Thus the rover's control schemes should be executed at a rate of 50 Hz or lower (at least for the sampling of these two control boards). Attempts to schedule a periodic task lower than the 100 Hz will result in unstable operating conditions.

The second method for real-time task scheduling, *one-shot* mode, provides a solution to this problem. During one-shot mode, tasks are scheduled to run only once. For applications where periodic timing is desired, such as in the control schemes developed in this project, the task must be rescheduled during each time step. Fortunately, this can be performed automatically at run-time for any control scheme created with RTW by simply executing the program with the *-o* command line option as discussed above. Thus, to the user, this method functions like the periodic mode. However, this results in about 15-20% more computational overhead than periodic mode(47). This could possibly present an issue if the rover's control schemes become greatly more complex, but for the scope of the control schemes presented here, this was not an issue.

4.2 SOFTWARE

4.2.1 RTAI LAB

One of the more valuable aspects of the RTAI environment is the RTAI Lab graphical interface. It is available in a few slightly different varieties. The one included with RTAI is xrtailab which is built on the eFTLK graphics library. Another is jrtailab which is built using Java. Finally, QRtaiLab is built using the Qt graphics library. All three serve the same basic function. However, QRtaiLab was the only one found to be reliable enough for this project.

The purpose of the RTAI Lab graphical interface is to provide the user with a method for observing and tuning RTAI control schemes while they are executing. RTAI Lab provides a virtual oscilloscope, meter, and LED for observing a running control scheme. These are implemented in control schemes by using the Simulink® blocks included under *RTAI Devices*. Also, parameters (variables), such as controller gains, can be adjusted in real-time without the need to stop and recompile the control scheme. As will be discussed later, this proved very useful for tuning all of the rover's controllers. A screenshot of QRtaiLab can be seen below in Figure 4.1.

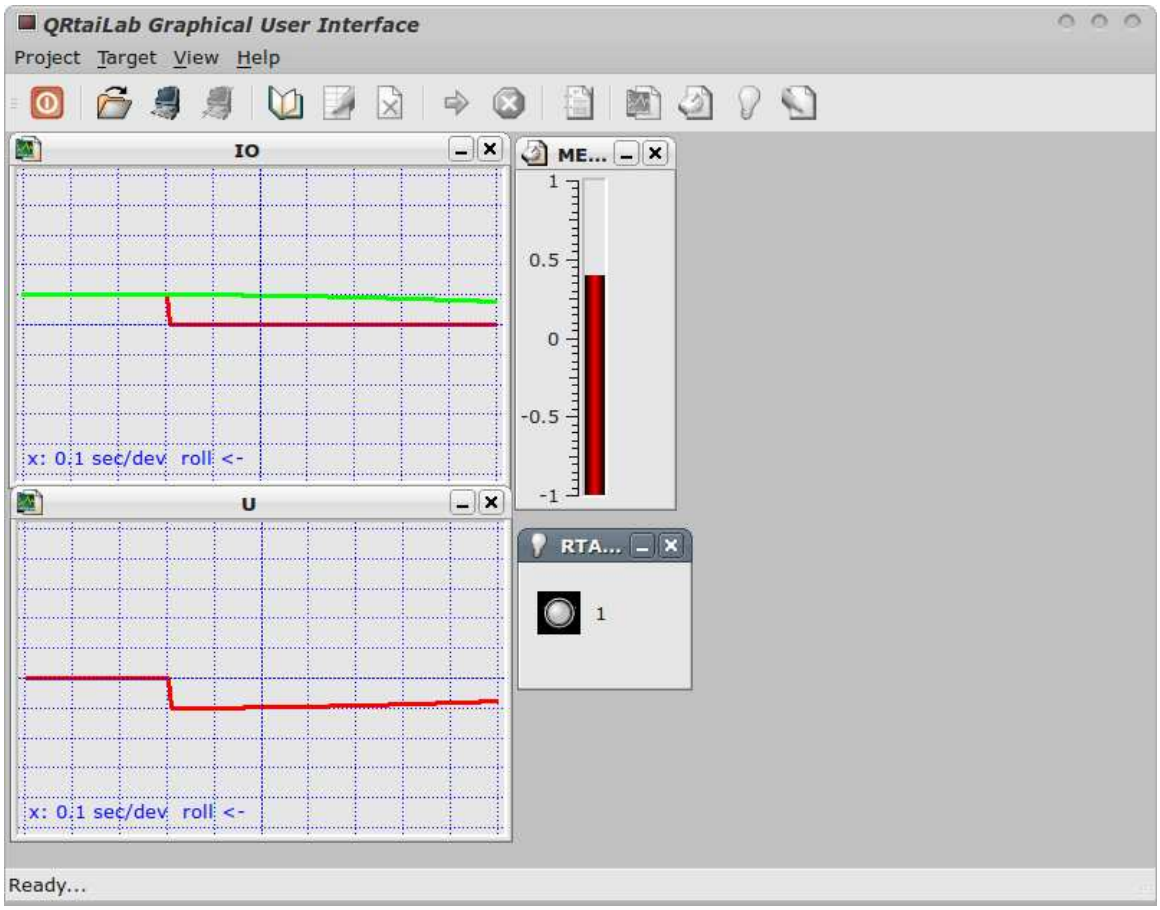

Figure 4.1: QRtaiLab running two scopes, a meter, and an LED

One promising feature of RTAI Lab is the ability to perform tuning and observation tasks over a local network. This functionality could be useful for tuning controllers in situations where the RTAI computer cannot be connected to a monitor, keyboard, and mouse (such as navigation experiments with the rover) or on a *leaner* implementation of RTAI (one without a graphical window manager and thus no RTAI Lab). However, this capability was never achieved in the systems tested for this project. Once a connection was established between the onboard computer and another computer running RTAI Lab, it would immediately lock up the onboard computer. Significant effort was put into remedying this problem, but it was never solved. This problem was mitigated by using the remote desktop interface built into Ubuntu, as discussed in Section 4.2.3.

4.2.2 CUSTOM SIMULINK BLOCKS

A set of custom Simulink® blocks were created in order to interface the rover's sensors and actuators with control schemes. These blocks were created using the C programming language. A template is provided by Matlab® that must be followed in order for the blocks to work in Simulink®. Programs adhering to this template are referred to by Matlab® as S-Functions. Once an S-Function has been created, it must be compiled in the Matlab® command line using the *mex* command. This creates a *.mexglx file that must be kept in the current model's working directory. After the S-Function has been compiled, it can be used in Simulink® by inserting the S-Function block under *Simulink/User-Defined Function* in the blocks menu and specifying the relevant filename. This block can then be *masked* to include a description, parameter input fields, a title, and descriptive port labels. Once the blocks were created, they were added to a master source directory and the Matlab® path variable for easier inclusion in models. The custom block-set can be opened in Matlab® by issuing the *roverblocks* command.

When compiling a real-time control scheme that incorporates S-Function blocks, the source files (*.c) for the blocks must be placed in the _rtai directory for the current working directory. For example, if you are creating a Simulink® model called *test.mdl,* when you click "Generate Code" under the configuration parameters dialog, two directories will be created. One directory will be *test_rtai* and the other will be *slprj.* Copy any relative *.c files from the working directory (the one that contains *test.mdl*) into *test_rtai*. Then compile the real-time executable as usual.

When designing these custom blocks, care was taken to ensure maximum modularity. The blocks were created to be as generic as possible to allow for future expansion to the rover or for use with other applications.

4.2.2.1 ROVER SERVO CONTROLLER

The first custom block created is used to interact with the *Pololu Serial 8-Servo Controller.* The servo controller does not require any special drivers. It simply accepts a signal from an RS-232 serial port. Instructions are given to the controller in three byte signals of the form: Byte 1 should be set to 255. This instructs the servo controller that a command is being sent. Byte 2 serves as an index to tell the controller which servo to move. This value can be between 0 and 7; 0 is the first servo and 7 is the last one. Finally, byte 3 gives the desired position. This value can be between 0 and 254, with 127 corresponding to neutral. For example, the command {255, 1, 127} would move the second servo connected to the controller to its neutral position.

The block is initialized by setting the baud rate to 9600 and a mode of 8N1, which corresponds to eight data bits, no parity bit, and one stop bit. Then a command is sent out to zero all servos to the neutral position. For every time-step, the block reads an eight dimensional (or fewer if desired) input vector, which corresponds to the desired output positions of the eight possible servos on the controller board, and sets the positions of the servos using the above described command signal format. Finally, when the program terminates, the servos are returned to their neutral positions, and the serial port is closed.

One special consideration must be taken with this block. The ESC for the drive motors is equipped with a "smart" braking feature. This prevents the drive motors from going directly from a forward direction to a reverse direction. To engage reverse, the ESC must be set to reverse, neutral, then back into reverse. As such, the first servo position is reserved for the ESC. If such a condition is found, the block will automatically engage the reverse maneuver over the course of a few samples. For this reason, two different blocks are provided. The *Rover Servos* block accounts for the smart braking feature on the first servo port, while the *Pololu Serial Controller* block does not.

An example model using this block can be seen in Figure 4.2. The example model sets the drive motors to a value of 150 and varies the steering servos in a sinusoidal fashion. Figure 4.3 provides a view of the mask that was created for the block. The source code for the block's S-Function can be found in Appendix B.1.

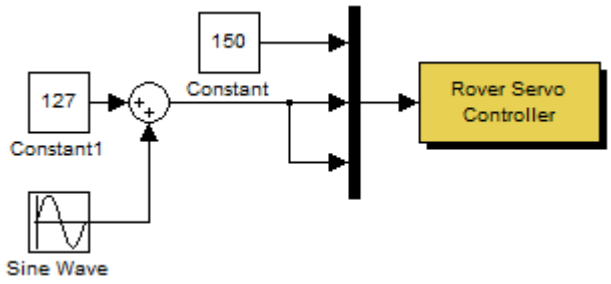

Figure 4.2: Example of the rover servos block

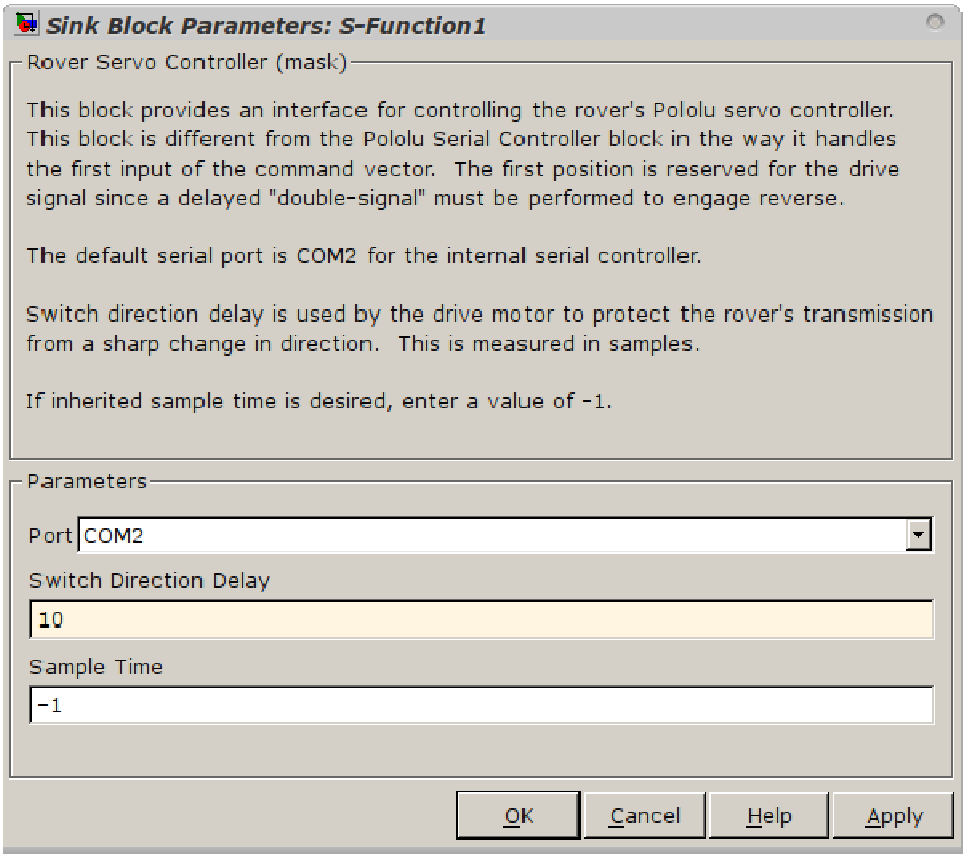

Figure 4.3: Rover servos block mask

4.2.2.2 PHIDGET ENCODER READER

The second custom block created is used to interface a *PhidgetEncoder High-Speed USB Encoder Reader*. The rover is equipped with two of these controller boards. This block, however, was designed to control only one encoder reader. Multiple instances of this block are used in this project by utilizing the *serial number* parameter of the block to differentiate between the encoder readers. This was done in order to make the blocks more modular in the event that more encoder readers are added in the future.

In order to use the Phidgets controllers, the appropriate drivers must be installed. Special Linux drivers are found on the manufacturer's website. This package was installed using the included instructions. This driver package includes the API needed to interface the control boards. The header file, *phidgets21.h*, must be included in any C program using the API. Any program using this must use the *–lphidget21* compiler flag when compiling.

The block is initialized by creating a *handle* pointer to the Phidgets object. This handle is then used to open the connection to the Phidget controller using its serial number. If the serial number is not known, a value of -1 may be entered, which will cause the program to use the first available Phidgets controller. Once the connection to the controller has been initiated, the *waitForAttachment* method is called to verify that the Phidget has opened correctly. If so, a confirmation message along with the device's serial number is printed to the terminal. The handle is then passed on to the S-Function's *work variable* since global variables are not allowed in a Simlink® S-Function.

During each time-step, the handle pointer is accessed through the *work-variable* and is used to retrieve the current encoder position into a temporary variable. This variable is compared to the position retrieved last time step. This provides the change in position since the last time step. The

change in position is output through the first port, and the absolute position of the encoder since the beginning of the simulation is output through the second port.

Once the simulation has terminated, the Phidgets device is closed and its handle is freed. An example model utilizing this block can be seen in Figure 4.4 below. This model drives the rover straight at a constant value and outputs the ticks per sample of each wheel encoder to a pair of RTAI Lab scopes. The custom mask for this block can be seen in Figure 4.5. The source code for this block can be found in Appendix B.2.

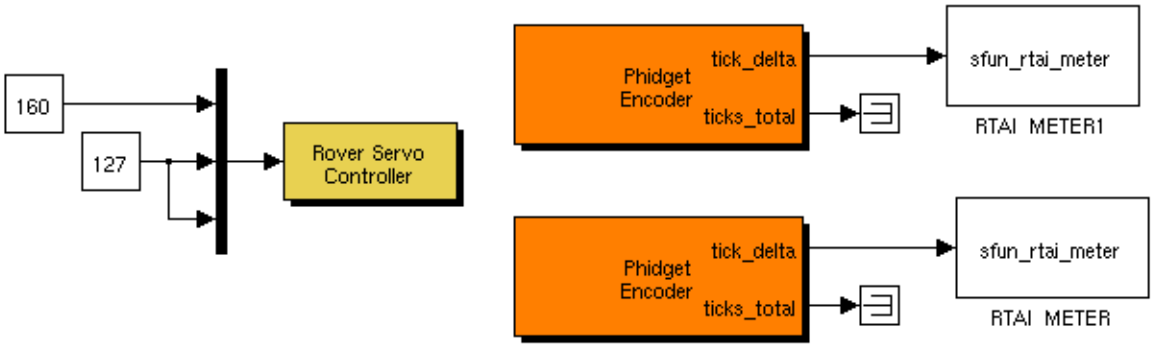

Figure 4.4: Example of the Phidget encoder reader block

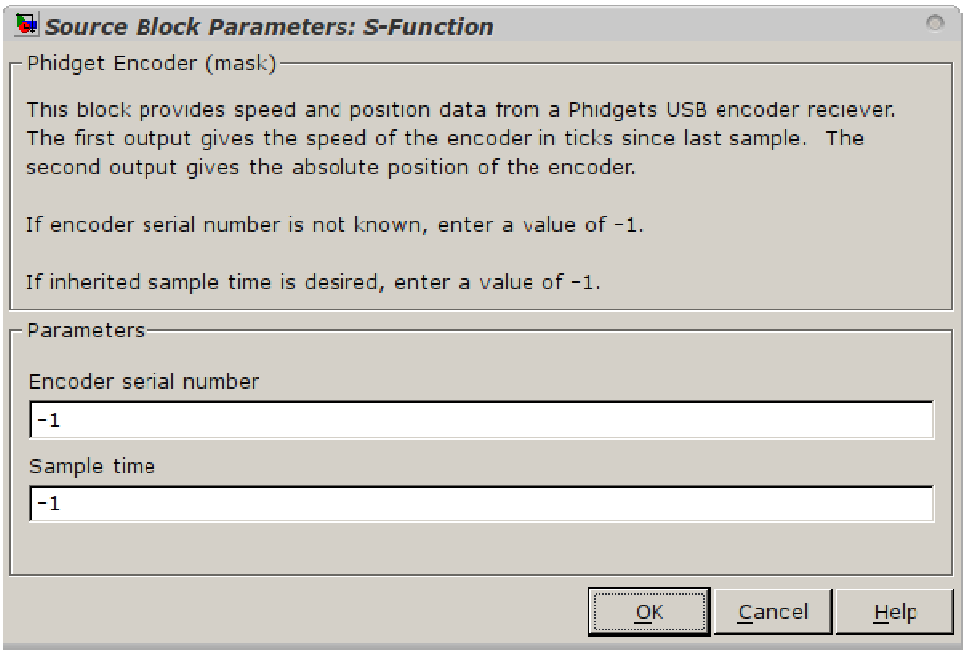

Figure 4.5: Phidget Encoder reader mask

4.2.2.3 PHIDGET INTERFACE KIT

Another custom block was created to interface the rover's *PhidgetIntefaceKit 8/8/8*. This block was created using nearly the exact same principles as the previous block. The initialization and termination of the block are identical to the encoder controller block except *CPhidgetInterfaceKitHandle* is used instead of *CPhidgetEncoderHandle*. The main difference lies in the output phase of the s-function. The *PhidgetInterfaceKit* controller contains eight analog inputs, eight digital inputs, and eight digital outputs. Thus, the Simulink® block contains two output ports to return the values from the analog inputs and digital inputs, and one input port to send values to the digital outputs. Each of the ports sends or retrieves an eight-dimensional vector to correspond to the eight inputs or outputs. During the output phase of the block, each of these ports is addressed to set or get the desired states. The digital outputs are assigned using *setOutputState*, the analog inputs are read using *getSensorValue*, and the digital inputs are read using *getInputState*.

An example model applying this block can be seen in Figure 4.6. This model reads the raw analog values from the six IR sensors connected to the interface kit and saves their values to a file. It also sets the interface kit's digital outputs all to zero since they are not needed. Finally, the values obtained from the digital inputs are sent into a terminator block since they also are not needed. Note that while the digital inputs and outputs are not used for this project, functionality for interfacing them is provided for possible future research. The block's mask can be seen in Figure 4.7. The source code for this block can be found in Appendix B.3.

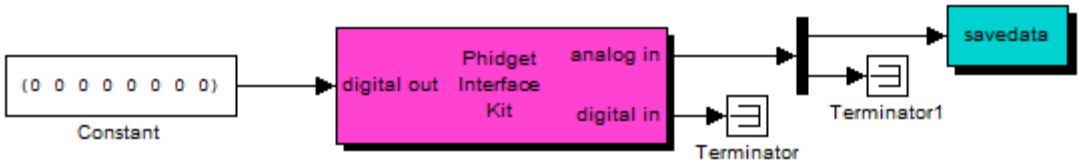

Figure 4.6: Example of the Phidget interface kit block

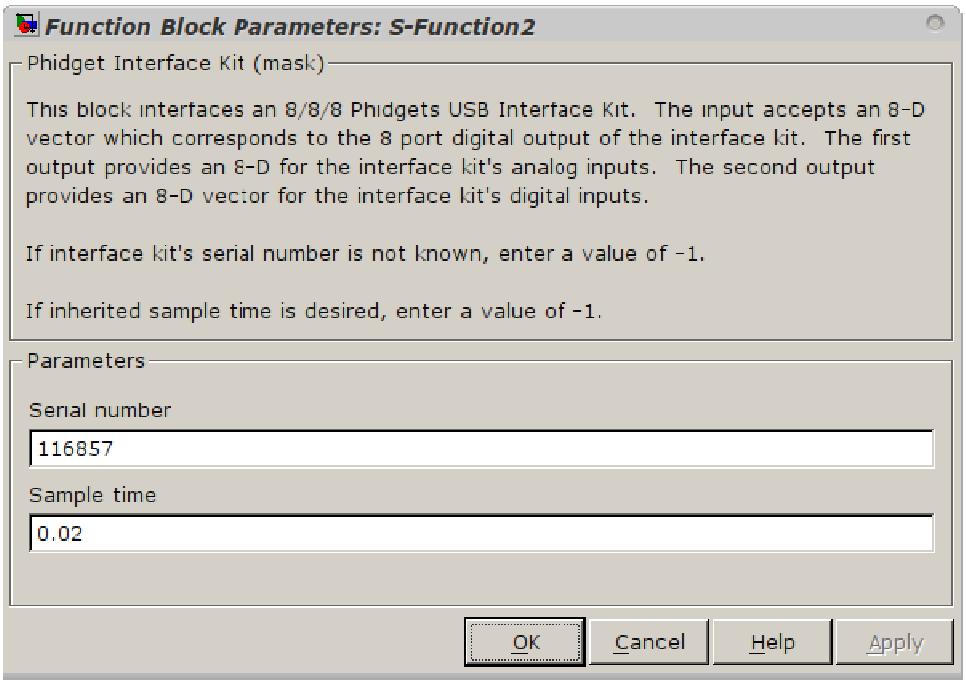

Figure 4.7: Phidget interface kit mask

4.2.2.4 SAVEDATA

The *savedata* Simulink® block, developed by the WVU Flight Controls Systems Group, is used to save signal data from a real-time control scheme. The block accepts a vector of signals that will be saved at the end of the simulation. The number of channels, filename, time range to save, and sampling rate are configurable via the block's mask. While Simulink® comes with a block used for saving data, it cannot work with RTAI. This is because the block provided by Simulink® writes to the hard disk while the simulation is running, and disk operations are too *expensive* to carry out in real-time. The *savedata* block used in this project stores the desired signals in the computer's RAM and writes the data to a file only after the simulation has ended. An example of this block can be seen in Figure 4.6 above. The block's mask can be seen in Figure 4.8. The source code for this block can be found in Appendix B.4.

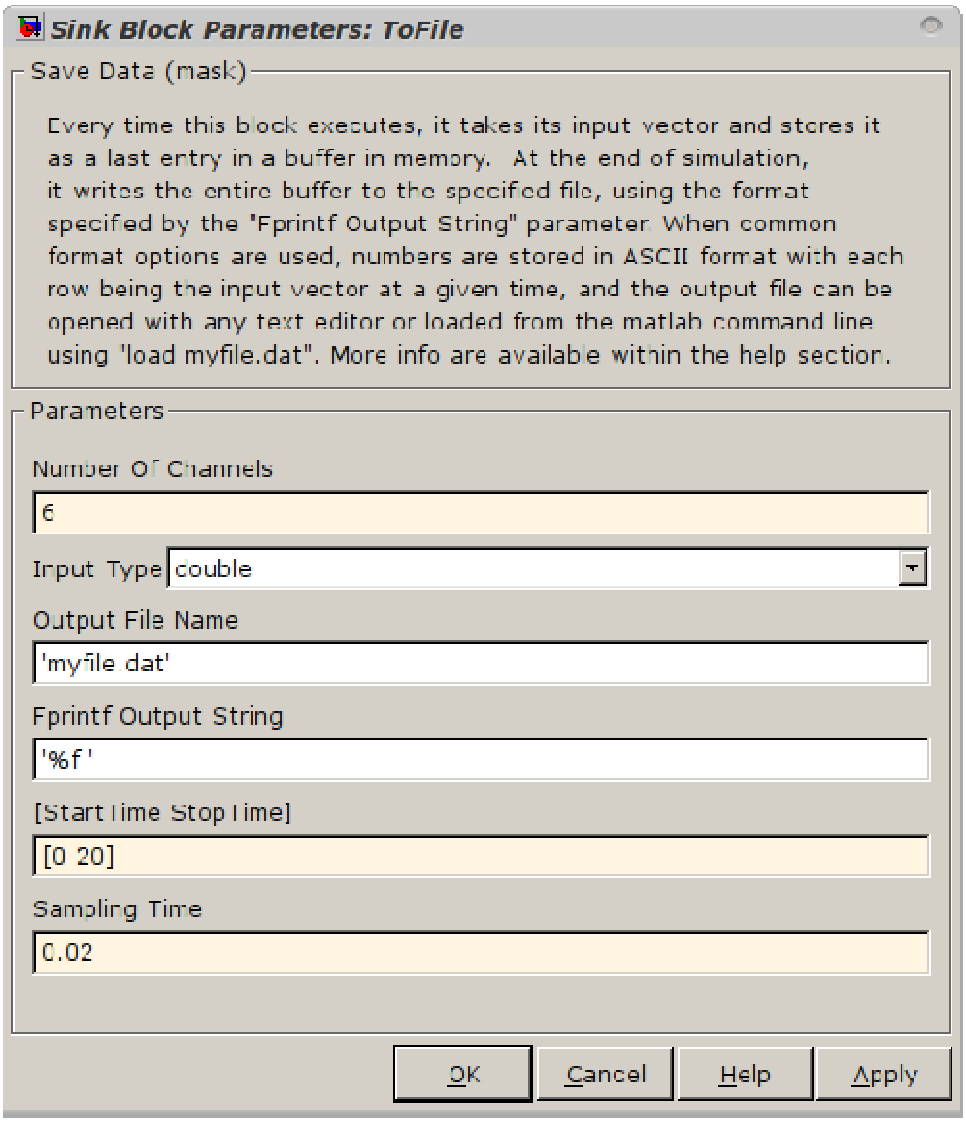

Figure 4.8: Savedata mask

4.2.2.5 SHARP IR SCALING

Another block that was created was the *Sharp IR Scaling* block. This simple block converts the analog signal of one of the IR proximity sensors into its corresponding distance in meters. This block was created as a typical subsystem without the need for a custom S-Function. It was included in the *roverblocks* library for convenience since several instances of it are often needed. This block

is the Simulink® implementation of Equation 4.1. The expanded subsystem can be seen in Figure 4.9.

$$\boldsymbol{Distance(m) = \frac{94.62}{Analog\ Value - 16.92}} \qquad \boldsymbol{(4.1)}$$

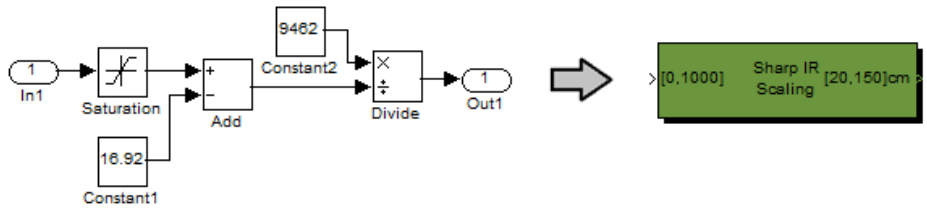

Figure 4.9: Sharp IR scaling subsystem

4.2.3 WIRELESS TUNING AND COMMUNICATION

The rover must be disconnected from its monitor, keyboard, and mouse during navigation experiments. Consequently, methods are needed to remotely start, stop, and monitor simulations, tune control scheme parameters, and give orders for high-level behaviors such as goal-seeking. It should be noted here that some of these tasks could be accomplished through the use of timers and delays, or by repeatedly reconnecting the computer to its input/output peripherals. However, these methods are largely inefficient and will not be used.

The rover's onboard computer is equipped with a wireless network adapter which can be used to communicate with other computers on the local network. As discussed earlier, RTAI Lab should be capable of providing the desired functionalities over a local network. However, the network capabilities of this program were never successfully implemented. Therefore, different network protocols must be used.

The solution to this was to use a simple remote desktop interface to communicate with the rover's onboard computer. Remote desktop enables a *client* computer to access the display and

input devices of a *host* machine over a network. This enables the *host*, the onboard computer, to be controlled as easily as if it were still connected to its own input peripherals. The wireless network connection introduces some latency into the interaction with the onboard computer, but this is not a problem for issuing simple terminal commands and parameter adjustments. RTAI Lab can still be utilized to tune control scheme parameters with this method since it is running on the onboard computer and not over the network.

CHAPTER 5 - CONTROL

A basic set of controllers was designed for the purpose of navigating the rover to goals while avoiding obstacles. To accomplish this, a mathematical model of the rover was developed and verified. Next, a PID controller was created and tuned to control the translational and rotational velocities of the rover. Once this was created, a potential field controller (PFC) was used to perform basic goal-seeking. Finally, the IR sensor arrays located on the front of the rover were integrated into the PFC controller to perform the task of obstacle avoidance during the goal-seeking behavior. This chapter outlines the steps taken in creating and tuning the required controllers and subsystems used for this task.

5.1 KINEMATIC MODEL

The first step in controlling the movements of the rover is to develop a mathematical model which can be used to estimate its position. As discussed in Chapter 2, the most basic form of mathematical model used for estimating the position of the robot is referred to as the kinematic model. For a wheeled mobile robot, the kinematic model is derived by analyzing each individual wheel's contribution to the movement of the vehicle and the geometric constraints imposed by their mounting configuration on the vehicle's chassis. This model assumes that the wheels roll without slipping, and dynamic forces are negligible since the rover will be travelling at low speeds. The rover can be described by the kinematic equations given for the car-like model by (36) in Equation 5.1.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{\tan\phi}{l} \\ 0 \end{bmatrix} v_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} v_2 \qquad \textbf{(5.1)}$$

In the above equation, x and y are the coordinates of the center of the rear axle of the rover. The orientation of the rover is given by θ. The angle of the front wheels with respect to a central body axis is given by ϕ. The distance between the front and rear wheel axles is given by l. For the TXT-1, l is approximately 0.33m. Finally, v_1 and v_2 refer to the translational velocity and the angular steering velocity respectively. This coordinate system is summarized in Figure 5.1. The Simulink® implementation of this front-wheel steering model can be seen in Figure 5.2.

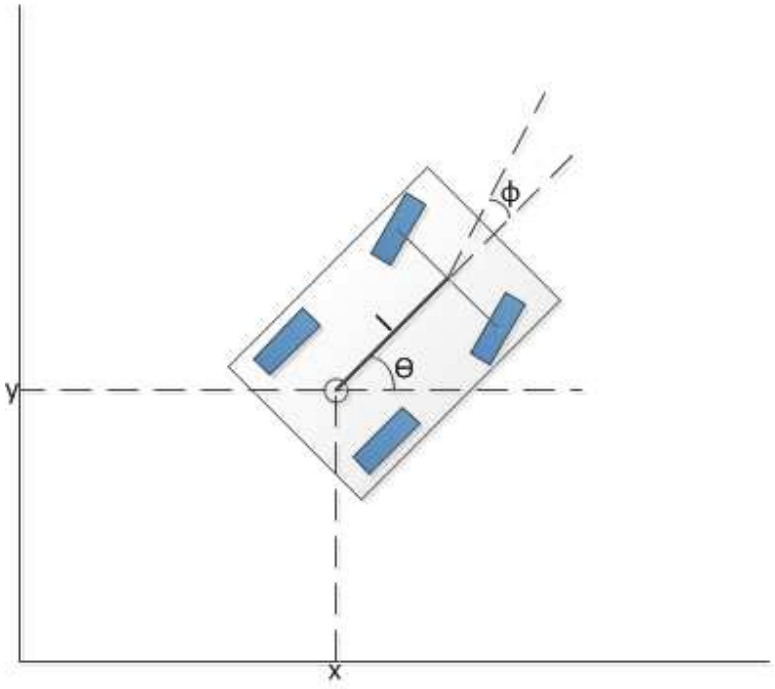

Figure 5.1: Coordinate system for front-wheel steering model

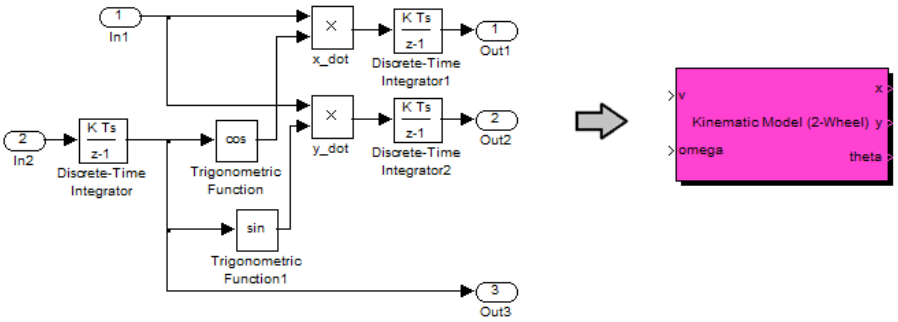

Figure 5.2: Simulink® implementation of front-wheel steering kinematics

The above equation describes the kinematic model for a car-like robot with front-wheel steering. However, the rover being developed here is capable of operating with either front-wheel steering or four-wheel steering. Thus a kinematic model is also needed for when four-wheel steering is being used. This variation of the kinematic model is referred to as the "body centered axis model" given by (23) (19). This can be seen below in Equation 5.2.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\phi \cos\theta \\ \cos\phi \sin\theta \\ \frac{\sin\phi}{l} \\ 0 \end{bmatrix} v_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} v_2 \qquad \textbf{(5.2)}$$

The above equation has a slightly different coordinate system than the one used in Equation 5.1. In this equation, the x, y coordinate is located midway between the front and rear axles. Also, l now refers to half of the distance between the front and rear axles. An updated diagram for this coordinate system can be seen in Figure 5.3.

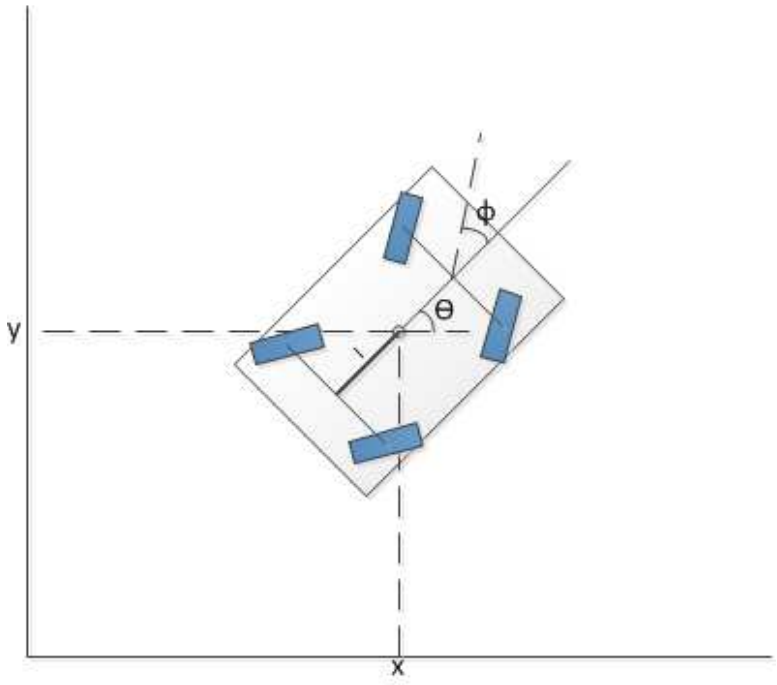

Figure 5.3: Coordinate system for four-wheel steering model

When operating in front-wheel steering mode, the rear steering servo input is locked to its neutral position. When using four-wheel steering, the rear steering servo angle, ϕ_r, is set to the negative value of the front steering servo angle, ϕ_f, such that $\phi_r = -\phi_f$. The Simulink® implementation of this four-wheel steering model can be seen in below in Figure 5.4.

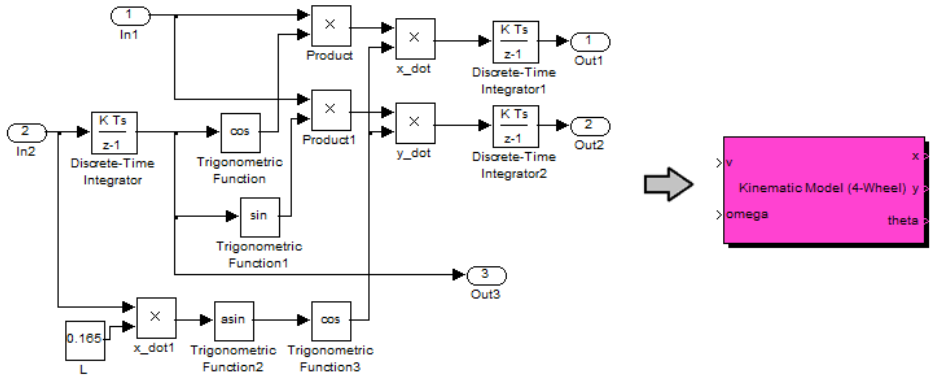

Figure 5.4: Simulink® implementation of four-wheel steering kinematics

In the above models, v_2 refers to the angular steering velocity. In this project, it is desirable to control the angular velocity of the vehicle rather than the angular steering velocity. As such, v_2 will not be considered in the design of this project's controllers. Henceforth, the translational velocity of the vehicle will be referred to as v, and the rotational velocity of the vehicle will be referred to as ω.

5.2 VELOCITY CONTROLLER

A PID controller is used for controlling the rover's translational and rotational velocities. This controller is used to guarantee that within a short time, the rover's actual velocities will correspond to the commanded velocities. Such a controller is necessary to compensate for battery fluctuations and unlevel terrain. This *low-level* velocity controller will later be utilized by the rover's *high-level* motion controllers.

5.2.1 ASSUMPTIONS

Several assumption and simplifications were made when designing the velocity controller. The first assumption is that the translational and rotational velocity controllers can be treated as uncoupled. This is not entirely accurate, since due to the nature of the vehicle, rotational velocity is not possible without translational velocity. However, the translational velocity of the rover will be bounded to a relatively small range of values during typical navigation maneuvers. Therefore, the rotational velocity controller will be tuned with the translational velocity set to this value. This approach was used successfully by (14).

Another assumption made is that the nonlinear components of the kinematic model are transient in nature and can be ignored in the design of the velocity controller. During most maneuvers, the steering angle or heading angle is held approximately constant. Therefore, the system is assumed linear around these points. As will be discussed in Section 6.3.1, these assumptions were found to be acceptable for the typical operating conditions encountered by the rover. Also, this general approach has been successfully used by several research groups. Marshall et. al.(19) (23) successfully implemented a PI controller for regulating velocity, and a PID controller was used for this purpose by (14) and (33).

5.2.2 VELOCITY ESTIMATION

The design of this controller begins with the calculation of the rover's translational and rotational velocities based upon the angular velocities of the front wheels. The angular velocities of the front wheels are calculated using their optical encoders. The translational and rotational velocities of the rover, as given by (14), are given below in Equations 5.3 and 5.4 respectively.

$$v = \frac{(q_R + q_L)\zeta}{2\Delta t} \quad \textbf{(5.3)}$$

$$\omega = \frac{(q_R - q_L)\zeta}{W\Delta t} \quad \textbf{(5.4)}$$

In the above equations, q_R and q_L refer to the number of ticks measured by the right and left wheel encoders respectively. The length of the simulation time step is given by Δt. For the control schemes developed in this project, this was always equal to 0.02 seconds since a sampling rate of 50 Hz is the maximum which can be used with the ESC and Phidget interface kit. The variable W represents the width between the rover's wheels. A value of 0.275 m was initially used for this. However, it was later determined experimentally that a value of 0.25 m provides more accurate velocity estimations for front-wheel steering and a value of 0.225 m provides a better estimation for four-wheel steering. Finally, ζ is a distance factor constant used to represent the distance travelled by the rover per encoder tick. This was approximated using the diameter of the wheels, $D = 0.159\,m$, and the resolution of the encoders, 1,024 ticks per revolution of the encoder. A theoretical value of $4.878 * 10^{-4}\ m/tick$ was calculated for ζ. However, due to slippage in the drive-train and between the wheels and ground, a value of $\zeta = 4.858 * 10^{-4}\ m/tick$ was determined experimentally for the rover on a flat tile surface. This was determined by simply recording and averaging the number of ticks measured by the encoders over a known distance. These two equations were programmed into two simple Simulink® blocks for easier inclusion in control schemes. This can be seen below in Figure 5.5.

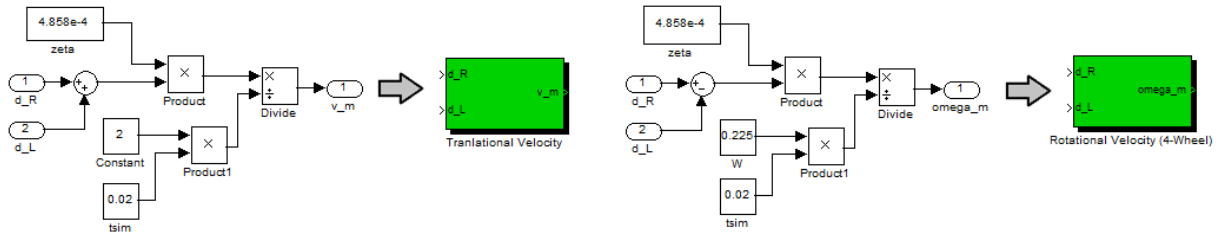

Figure 5.5: Simulink® representation of Equations 5.3 and 5.4

As can be seen in Figure 5.6, a significant amount of high-frequency noise was encountered in the angular velocity measurements obtained from the wheel encoders. Such noise would hinder the development of an effective velocity controller. To remedy this, a 3rd order, *low-pass* Butterworth filter was implemented to clean the signal. The coefficients for this filter were obtained using the Matlab® *butter* command. Figure 5.7 below presents an example of the designed filter's effectiveness when applied to the data from Figure 5.6. Finally the odometric velocity estimation system presented here was placed inside a Simulink® subsystem for inclusion in the rover's controllers. This can be seen in Figure 5.8.

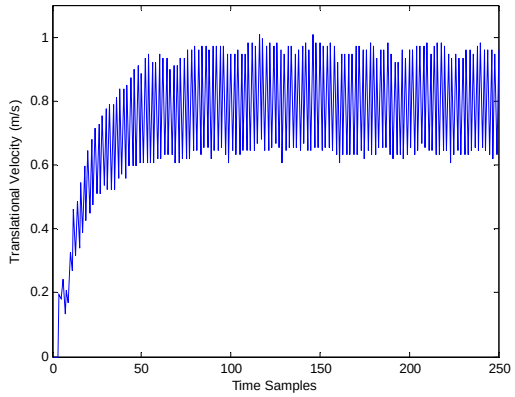

Figure 5.6: Raw velocity data

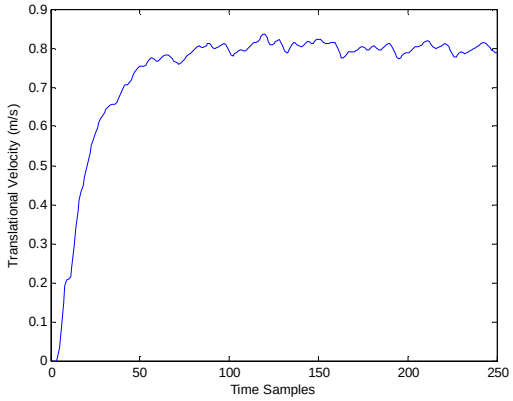

Figure 5.7: Filtered velocity data

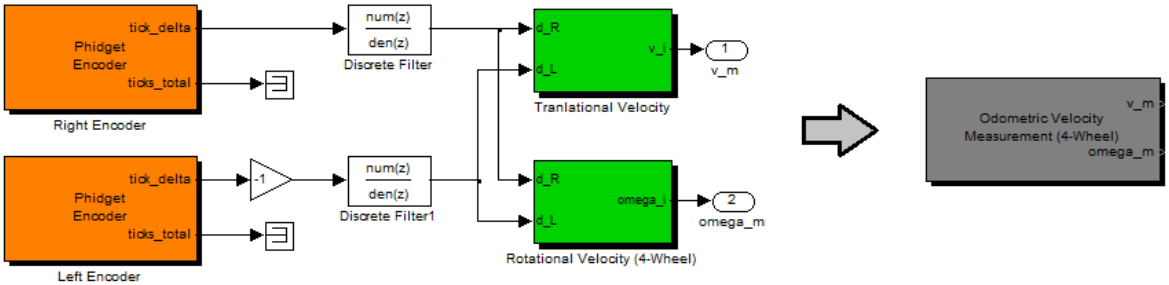

Figure 5.8: Odometric velocity estimation subsystem

5.2.3 PID CONTROLLER

Once a method for measuring the translational and rotational velocities of the rover was obtained, a discrete-time PID controller for velocity regulation was designed. This controller is used to ensure that after some short time, the error between the commanded velocities and measured velocities will be minimal. The error vector, e, at time-step k is given by Equation 5.5.

$$e(t_k) = \begin{bmatrix} v_c(t_k) - v_m(t_k) \\ \omega_c(t_k) - \omega_m(t_k) \end{bmatrix} \quad \textbf{(5.5)}$$

In the above equation, the subscript c denotes the command value, and the subscript m denotes the measured value.

The form of the discrete PID controller is given by:

$$\begin{bmatrix} u_{drive}(t_k) \\ u_{steer}(t_k) \end{bmatrix} = \begin{bmatrix} K_{P_v} & 0 \\ 0 & K_{P_\omega} \end{bmatrix} e(t_k) + \begin{bmatrix} K_{I_v} & 0 \\ 0 & K_{I_\omega} \end{bmatrix} \sum_{k=0}^{n} e(t_k) + \begin{bmatrix} K_{D_v} & 0 \\ 0 & K_{D_\omega} \end{bmatrix} [e(t_k) - e(t_{k-1})] \quad \textbf{(5.6)}$$

In the above equation, the controller outputs, u_{drive} and u_{steer} are used to command the servo controller. The gains for the proportional, integral, and derivative components of the controller are given by K_P, K_I, and K_D for the translational and rotational velocities. Since the equations in this controller are decoupled, the controller may be modeled as two independent controllers as seen in Figure 5.11 in Section 5.2.5.

One should note that controlling both the translational and rotational velocities also controls the current turning radius of the rover. Consequently, to achieve the minimum turning radius of the rover, the translational velocity must be decreased.

5.2.4 OUTPUT SCALING

This controller was designed such that its outputs are in the range $[-1,1]$. However, the drive and steering servos accept a range of [0,254]. For this reason, the PID controller outputs must be scaled to correspond to the specifications of the servo controller. A set of Simulink® blocks were created for this purpose.

The drive motors are controlled by the PID controller output, u_{drive}. For this output, a value of 1 corresponds to full speed forward, a value of 0 corresponds to stopped, and a value of -1 corresponds to full speed in reverse. For the drive motor servo command, 254 corresponds to full speed forward, and 0 corresponds to full speed reverse. However, there is an un-symmetrical *dead band* for the neutral position. This may be due to difference in forward and reverse drive train friction or inaccuracies in the ESC. It was determined experimentally that there is a dead band in the range of $[120, 150]$. The motor speed is assumed to vary linearly between the dead band and maximum. The Simulink® block for scaling the drive motors can be seen in below in Figure 5.9.

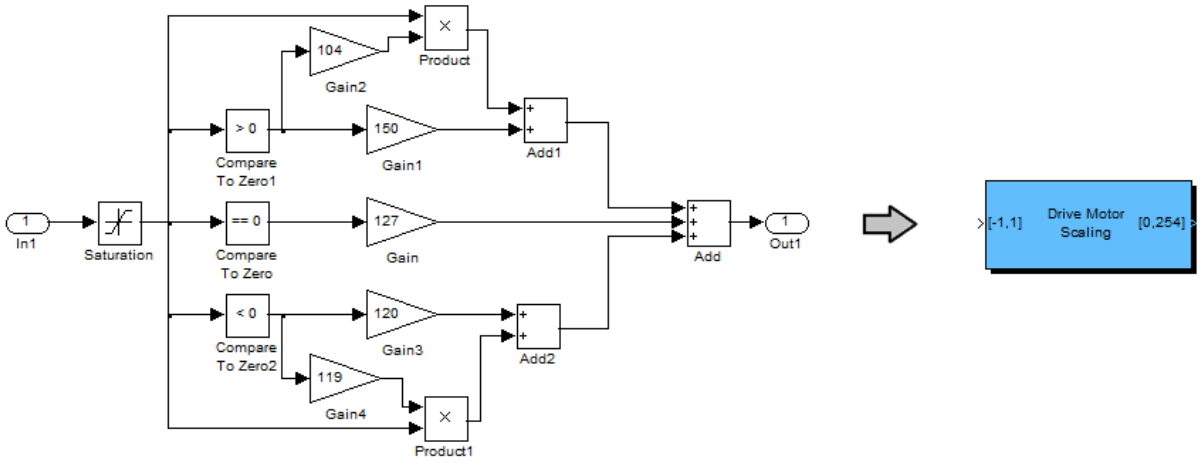

Figure 5.9: Drive motor scaling block

The steering servos are controlled by the PID controller output, u_{steer}. For this output, a value of 1 corresponds to the front wheels being turned completely to the right, a value of 0 corresponds to the wheels being centered, and a value of -1 corresponds to the wheel being turned completely to the left. Unlike the drive motors, the steering servos do not require a dead band. As such, the scaling of this output is much simpler than the drive motors. The steering servo mounting brackets may be adjusted in order to center the steering. The Simulink® block for scaling the steering servos can be seen in Figure 5.10.

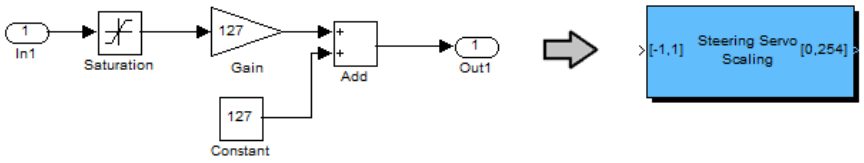

Figure 5.10: Steering servo scaling block

5.2.5 IMPLEMENTATION AND TUNING

Several minor difficulties were experienced during the physical implementation of the PID velocity controller. The first issue arose from the fact that the rover's drive-train experiences friction differently depending on whether it is in forward or reverse. This was anticipated when it was discovered that the drive motor dead band is not symmetrical. To resolve this issue, the translational velocity controller was broken into two separate controllers. Consequently, six gains must be tuned for the translational velocity controller instead of three. The controller output received by the servo controller is selected by the simulation based upon the sign of the commanded translational velocity. Another issue faced was the fact that the translational velocity

controller did not provide true zero velocity. To remedy this problem, another selection block was placed in Simulink model to override the translational velocity to zero. The Simulink® model used for tuning the PID controller can be seen in Figure 5.11 on the next page.

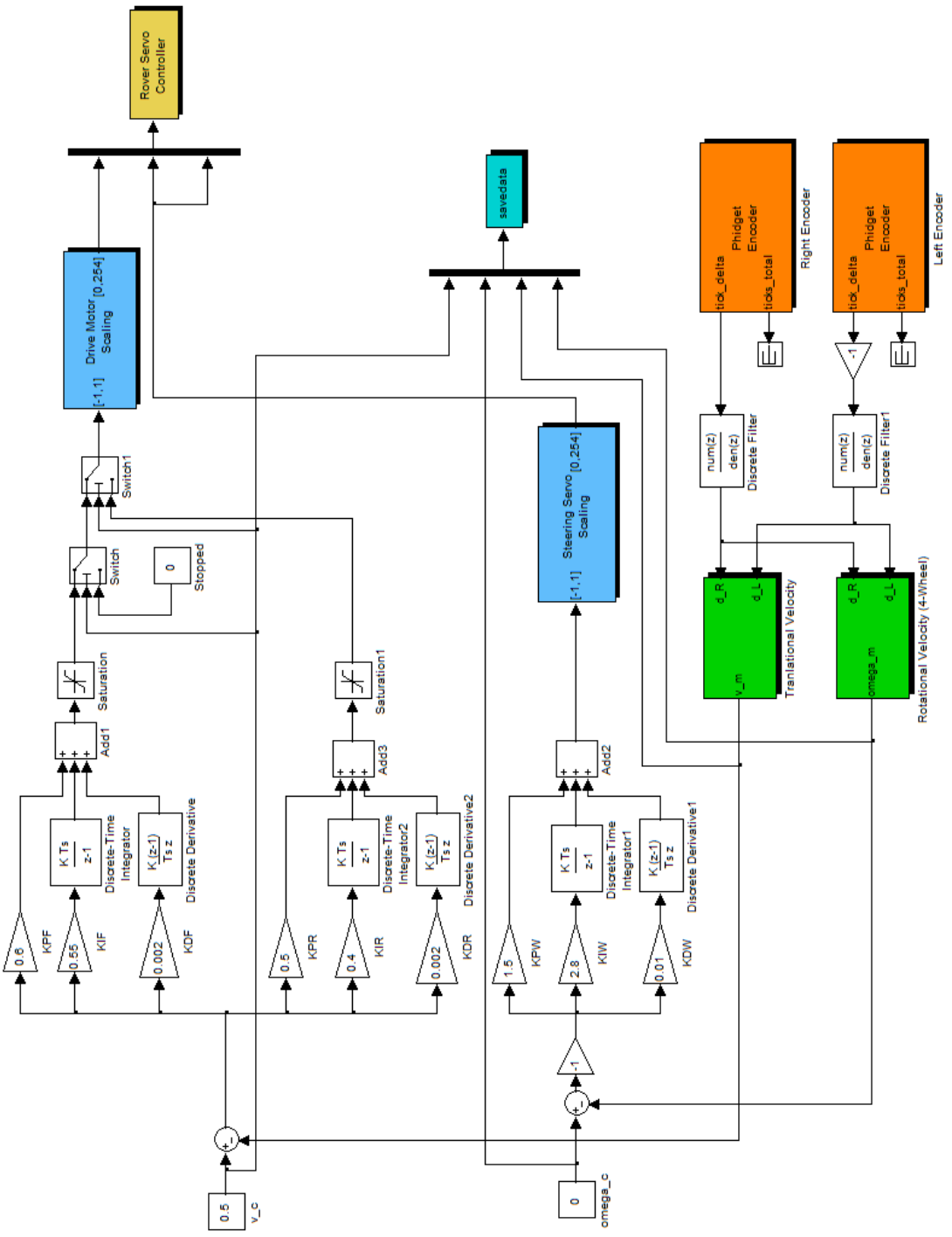

Figure 5.11: Simulink® model for PID controller (four-wheel steering)

The gains for the velocity controller were determined experimentally using online tuning. Each of the three controllers (forward translational, reverse translational and rotational velocities) were tuned independently. Since the translational and rotational velocities of the physical system are coupled, an iterative approach was taken to tuning the gains. First the translational velocity controllers were roughly tuned. Then the rotational velocity controller was tuned. This process was repeated until a suitable controller was obtained. The gains for the tuned controller can be found in Table 5.1. Once the tuning process was complete, the entire PID controller was placed inside a subsystem block for use in higher-level controllers. This can be seen in Figure 5.12.

Table 5.1: PID controller gains

	Front-wheel Steering	Four-wheel Steering
$K_{P_v}^{forward}$	0.6	0.6
$K_{I_v}^{forward}$	0.55	0.55
$K_{D_v}^{forward}$	0.002	0.002
$K_{P_v}^{reverse}$	0.5	0.5
$K_{I_v}^{reverse}$	0.4	0.4
$K_{D_v}^{reverse}$	0.002	0.002
K_{P_ω}	1.7	1.5
K_{I_ω}	2.5	2.8
K_{D_ω}	0.01	0.01

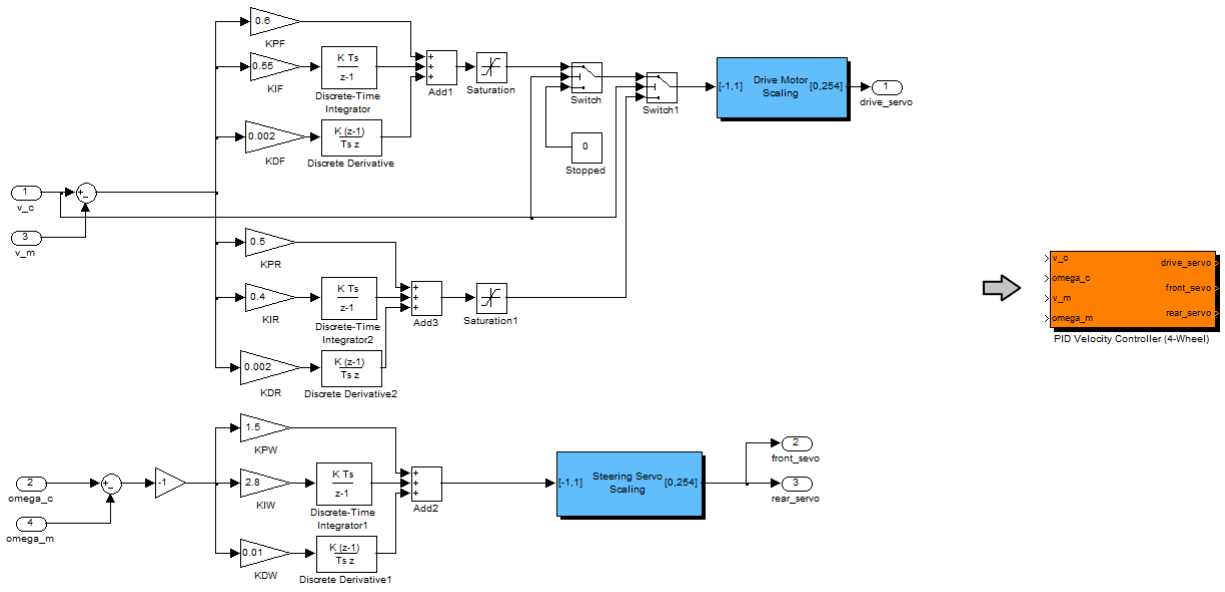

Figure 5.12: Simulink® velocity controller block

5.3 WAYPOINT TRACKING

A PFC controller is used on the rover for the purpose of path planning. This controller provides the rover with basic *waypoint tracking.* Typically, a PFC controller needs an *a priori* map of the area being navigated. However, since the rover will be used for exploring unknown, dynamic environments, this map will not be available. To counter this, obstacle information will be provided by the rover's IR sensors in real-time. Goal-seeking will be accomplished with a hybrid controller. This hybrid controller will be used to switch between a PFC-based waypoint tracking behavior and a contour tracking, obstacle avoidance (OA) behavior. This hybrid controller was inspired by (14). The obstacle avoidance controller will be discussed in Section 5.4.

5.3.1 POTENTIAL FIELD CONTROLLER

A PFC controller treats the robot as a point in a gradient, or potential, field (30). The goal acts an attractive force, and obstacles act as repulsive forces. As discussed earlier, obstacles will not

be handled by the PFC being developed in this project. A positive aspect of the PFC is that it can also serve as the motion control scheme. The attractive potential of the goal, U_{att}, is given by Equation 5.7 (14) (30).

$$U_{att} = \frac{1}{2} k_{att}[(x - x_g)^2 + (y - y_g)^2] \quad \textbf{(5.7)}$$

In the above equation, x and y are the current position of the rover as calculated by the kinematic equation using the measured velocities, and x_g and y_g represent the coordinates of the goal. The variable, k_{att} represents a positive factor for the intensity of the attractive force of the goal. The attractive force, $\vec{F}_{att}$, is given by the following equation (14):

$$\vec{F}_{att}(x, y) = -\nabla U_{att}(x, y) = -\begin{bmatrix} \frac{\partial U_{att}}{\partial x} \\ \frac{\partial U_{att}}{\partial y} \end{bmatrix} \quad \textbf{(5.8)}$$

which simplifies to (14):

$$\vec{F}_{att} = k_{att} \begin{bmatrix} x_g - x \\ y_g - y \end{bmatrix} \triangleq \begin{bmatrix} F_{att,x} \\ F_{att,y} \end{bmatrix} \quad \textbf{(5.9)}$$

This attractive gradient is used to control the motion of the rover. The rover is given a constant velocity, and the steering is calculated based on the force calculated above. The desired heading angle to the goal, θ_g, for the rover using the following equation (14):

$$\theta_g = arctan2(F_{att,y}, F_{att,x}) \quad \textbf{(5.10)}$$

In the above equation, $arctan2$ calculates the angle with respect to the correct quadrant. Once the desired angle is calculated, a basic proportional controller is used to track this angle. The controller developed for this purpose is a simple proportional controller of the following form:

$$u_{PFC_\omega}(t_k) = K_{PFC_\omega} e(t_k) \quad \textbf{(5.11)}$$

where

$$e(t_k) = \theta_g - \theta \tag{5.12}$$

In the above equations, $u_{PFC_\omega}(t_k)$ is the commanded angular velocity, and θ refers to current orientation of the rover obtained from the kinematic equations using the measured velocities. The proportional gain for the controller is given by K_{PFC_ω}. This gain is related to the maximum angular velocity of the rover. A suitable value for this was determined experimentally. For front-wheel steering, a value of 0.8 was used, and a value of 1.5 was used for four-wheel steering. A translational velocity of 0.5 m/s was used for typical waypoint tracking.

5.3.2 IMPLEMENTATION

When developing the PFC, Simulink® was used to model the *ideal* system using the kinematic equations and assuming perfect velocity control. In this model, the distance to the goal is checked after every step to see if the rover is close enough. This is to prevent it from overshooting the goal. In practice, the rover is stopped within 0.1m of the goal. This may be adjusted if needed. This test model can be seen below in Figure 5.13.

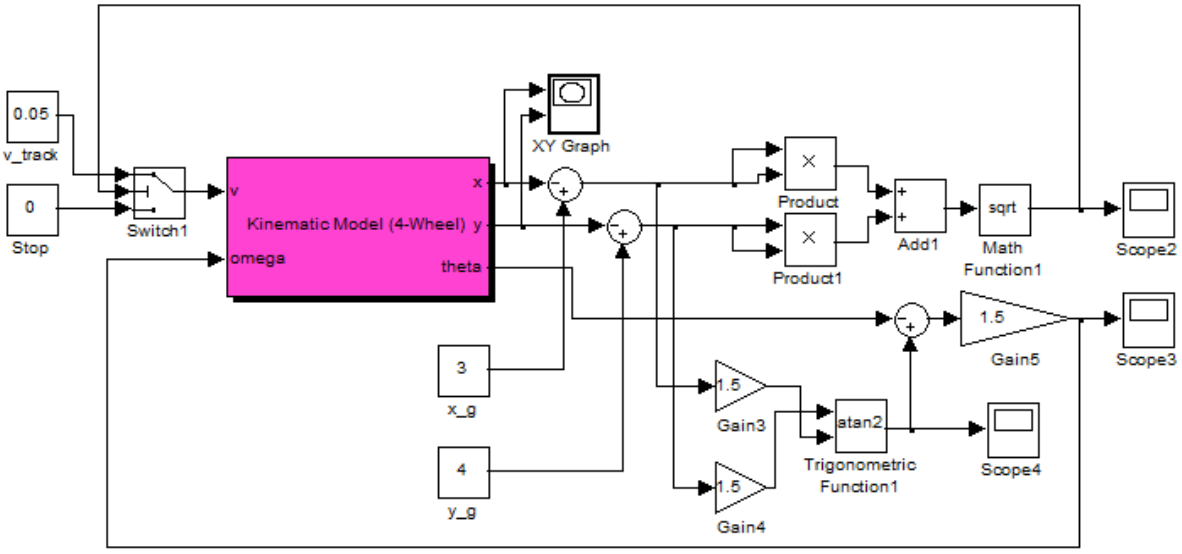

Figure 5.13: Simulink® model of ideal PFC

For the PFC implementation on the actual rover, the position of the rover is calculated using the method described in Section 5.2.1. The PFC provides the values for the commanded velocities used by the PID velocity controller. Position and velocity data from the simulation is saved so it may be compared or analyzed later. A simple controller was added to gradually slow the rover once it is less than a meter from the goal. The Simulink® model for the PFC implemented on the rover can be seen in Figure 5.14. As seen in Figure 5.15, this system was also added to a Simulink® subsystem for easier inclusion in control schemes.

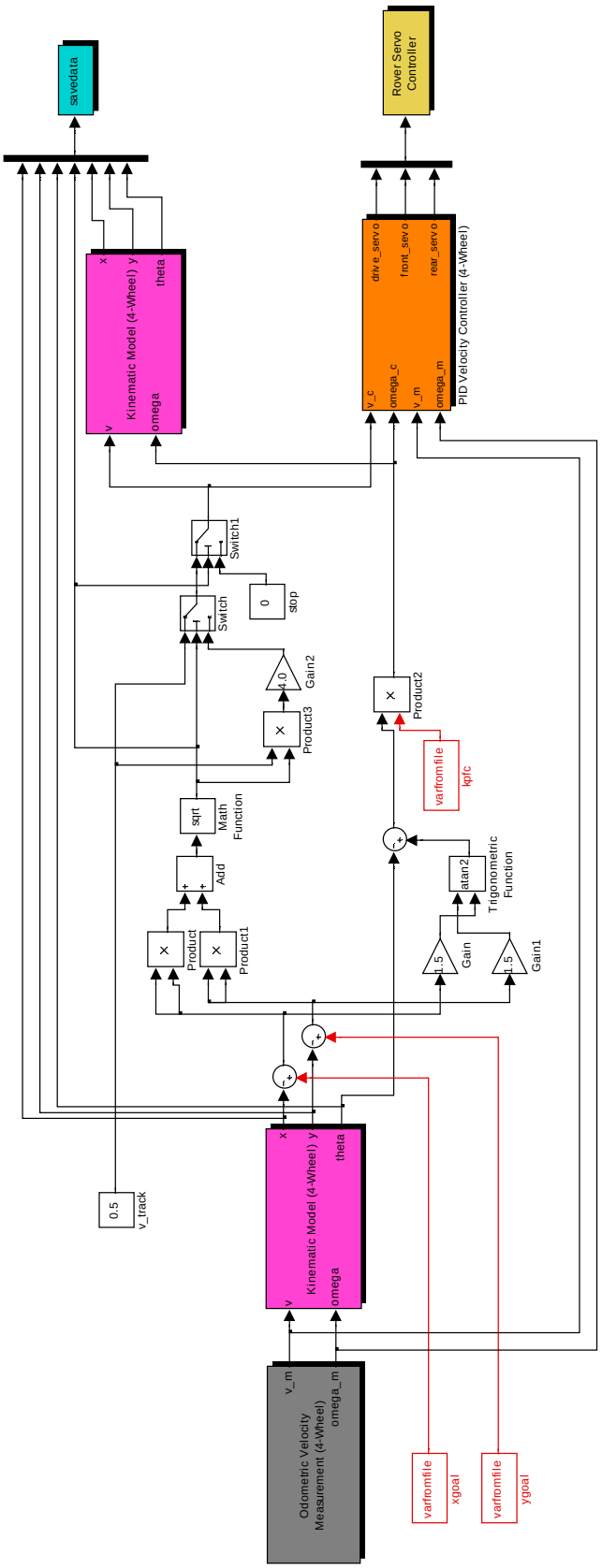

Figure 5.14: Simulink® model for actual PFC

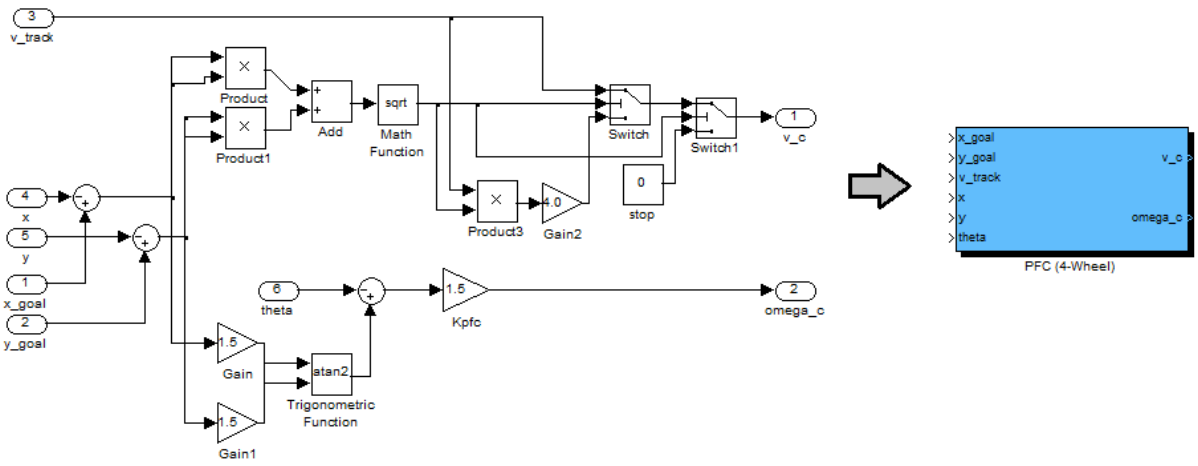

Figure 5.15: Simulink® block for PFC

5.3.3 LIMITATIONS

The PFC was used because it provides a relatively simple solution to the complex problem of controlling a nonholonomic vehicle. However, there are several limitations to using this type of controller. Primarily, in situations where obstacles are too close together, or convex in shape, the robot may become trapped. Also, in situations where the rover rotates 180°, the $arctan2$ function causes the rover to lose its path. This is due the fact that the angle the $arctan2$ function is bounded to the range $[-\pi, \pi]$. This could be remedied by using an incremental such as the one used in (40).

5.4 OBSTACLE AVOIDANCE CONTROLLER

It is assumed that the rover is operating in an unknown environment. As such, the PFC used by the rover cannot incorporate any type of obstacle avoidance. To remedy this, a separate obstacle avoidance controller was developed to dynamically maneuver past obstacles. The method described and used here is an adaptation of the method described by Cruz et. al. (14).

An array of six IR sensors provides the rover with distance data in order to perceive obstacles ahead of the vehicle. Each of these sensors is capable accurately measuring the distance to objects directly in from them. The effect range of these is 0.20 to 1.50 m. The arrangement and indices of the sensors can be seen in Figure 5.16.

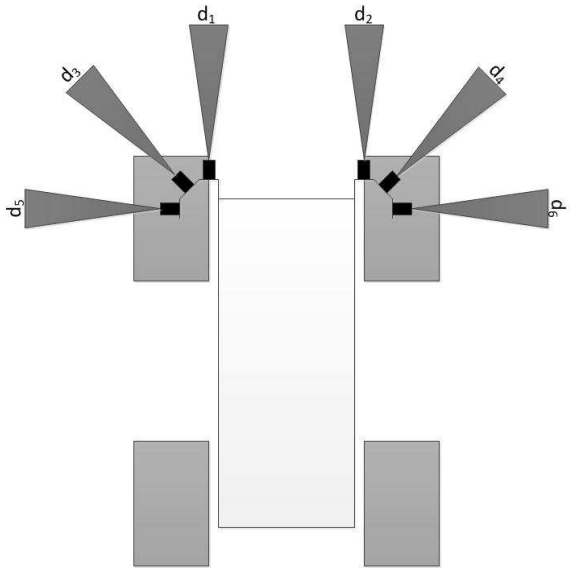

Figure 5.16: Numbering and layout of IR sensor array

The obstacle avoidance controller utilizes a set of virtual repulsive forces to steer the rover away from obstacles and to slow its translational velocity, such that a smaller turning radius is obtained. These virtual forces are described by the Equations 5.13 and 5.14 below.

$$F_L(t_k) = \sum_{n=1}^{3} \frac{w_{(2n-1)}}{d_{(2n-1)}(t_k)} \quad \textbf{(5.13)}$$

$$F_R(t_k) = \sum_{n=1}^{3} \frac{w_{2n}}{d_{2n}(t_k)} \quad \textbf{(5.14)}$$

In the above equations, $F_L(t_k)$ and $F_R(t_k)$ are the virtual forces. Each of the IR sensors carries a constant weight, w_i, to determine its influence. These values were experimentally determined and

are given by the vector $\vec{w} = [1.0\ 1.0\ 0.7\ 0.7\ 0.05\ 0.05]$. Finally, the distance reported by each sensor during a given time step is given by $d_i(t_k)$.

Using these virtual forces, a desired rotational velocity can be calculated using a proportional controller given by the following form:

$$u_{OA_\omega}(t_k) = K_{OA_\omega}[F_L(t_k) - F_R(t_k)] \quad \textbf{(5.15)}$$

In the above equation, the commanded angular velocity is given by $u_{OA_\omega}(t_k)$ and the proportional gain is given by K_{OA_ω}. A gain of -0.4 was experimentally determined to be suitable.

These virtual forces are also used to reduce the translational velocity of the rover as it gets closer to obstacles. This serves to reduce the turning radius and to ensure that the rover will come to a stop if it gets trapped in complex obstacles. The commanded velocity $u_{OA_v}(t_k)$ is calculated with a proportional controller of the following form:

$$u_{OA_v}(t_k) = v_{track} - K_{OA_v}[F_L + F_R] \quad \textbf{(5.16)}$$

The gain, K_{OA_v}, in the above equation was experimentally tuned to 0.045. The gains in this controller are valid for either front-wheel or four-wheel steering. The Simulink® block for this controller can be seen below in Figure 5.17.

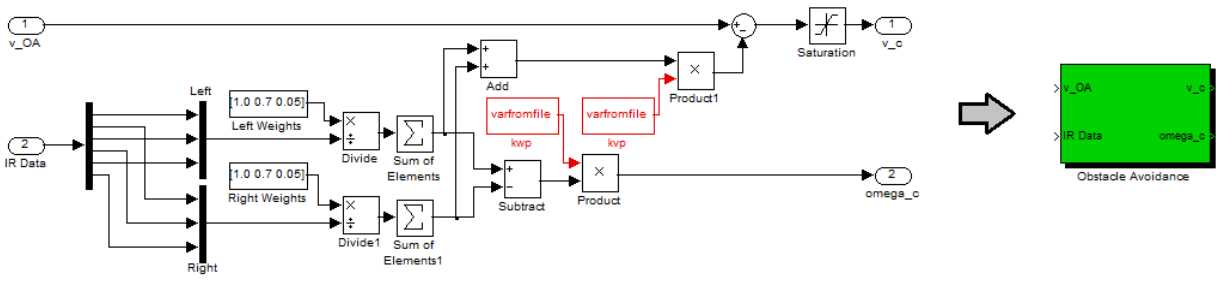

Figure 5.17: Simulink® block for OA controller

5.5 HYBRID CONTROL

As mentioned previously, waypoint tracking is accomplished using both a PFC and an obstacle avoidance controller. The PFC controller directs the rover when no obstacles are detected by the IR array. The obstacle avoidance controller is initiated any time IR sensors 1 through 4 detects an obstacle within 0.5 m, or a IR sensors 5 or 6 detect an obstacle within 0.3 m. The Simulink® implementation of this hybrid controller can be seen in Figure 5.18 on the next page.

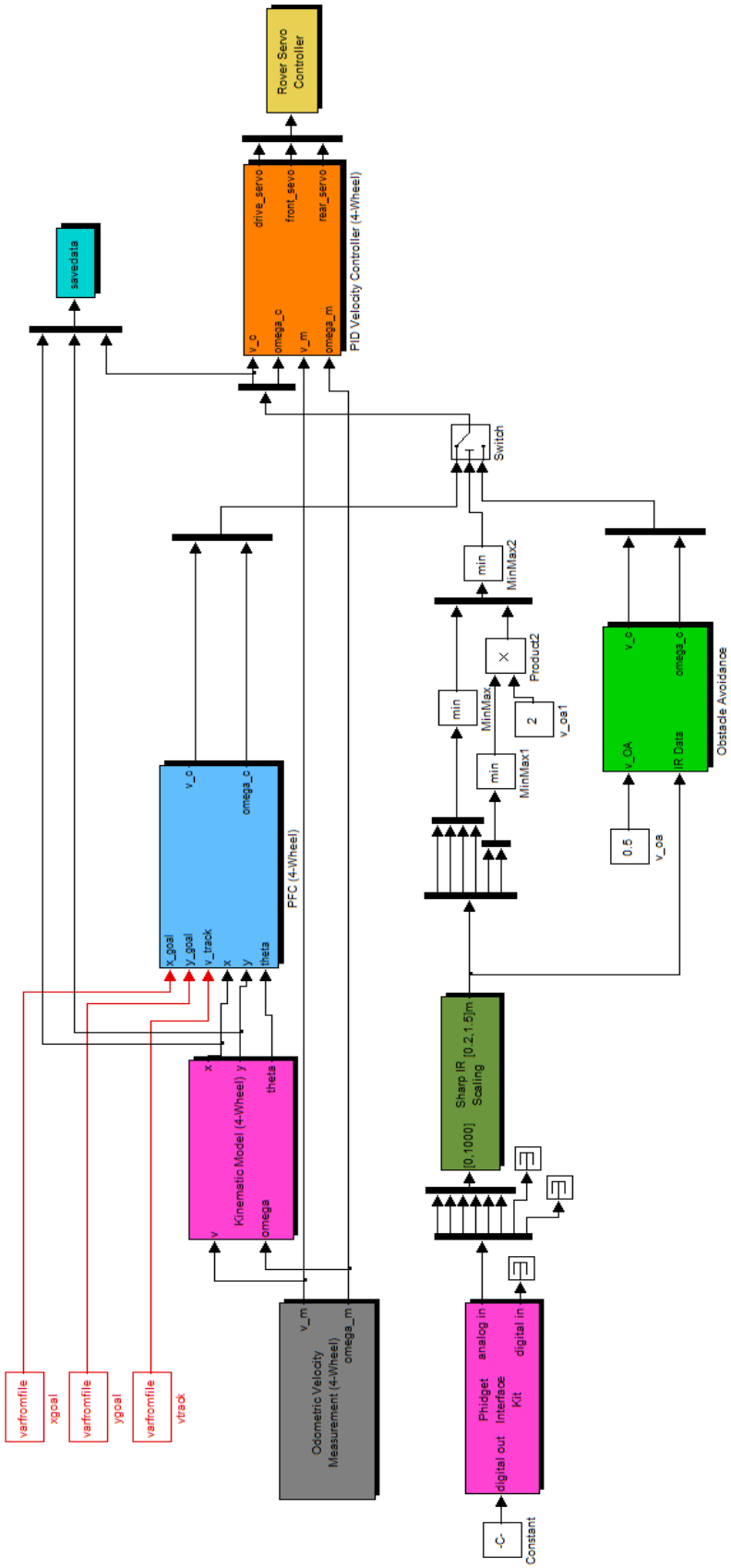

Figure 5.18: Hybrid waypoint tracking controller

CHAPTER 6 - RESULTS

The rover developed in this thesis performed as expected. This chapter will describe the experiments conducted with the rover and observations about its performance.

6.1 HARDWARE

The physical modifications to the rover platform served to better the suitability for indoor navigation. Throughout the course of these experiments, it was determined that four-wheel steering is much more conducive to the tight spaces encountered in a typical laboratory setting being that it allows for a tighter turning radius. Additionally, as will be discussed in Section 6.3.1, the rotational velocity controller for four-wheel steering exhibited a quicker response.

The addition of stiffer suspension components proved absolutely necessary. Without this measure, significant sag was observed in the drive-train, which would have inhibited accurate control. Over the course of experiments conducted, an even stiffer set of springs were installed to further reduce sag. With the addition of the stiffer springs and damper oil, the current payload of the rover with easily carried, insinuating that a heavier payload, including perhaps more sensors or the addition of cameras, could easily be accommodated with the rover's current configuration.

For indoor navigation, the maximum translational velocity of the rover was never fully utilized. It was estimated that the maximum translational velocity of the rover with the added computer and sensor payload was approximately 2.5 m/s while most of the tests performed with the rover were done so at 0.8 m/s or less.

The battery life of both the onboard computer and drive motors was found to be sufficient for typical tuning and testing sessions. A runtime of approximately four hours was consistently

obtained by the onboard computer. The driver motors were estimated to get approximately one hour of battery under moderate conditions. However, due to the nature of preparing experiments and collecting data between runs, the computer battery was always the limiting component.

The resolution of the quadrature encoders was sufficient for accurate distance measurements. However, significant noise was observed in their measurements. As was discussed in Section 5.2.1, suitable velocity estimation was obtained once the output of theses sensors was filtered. Finally, the IR sensors were found to be adequate for indoor obstacle avoidance. However, the sensors are practically worthless in direct sunlight. As such, they are not suitable for outdoor navigation.

The final rover configuration connected to a monitor and input peripherals can be seen below in Figure 6.1. In this type of setup, the rover is powered by an external power supply.

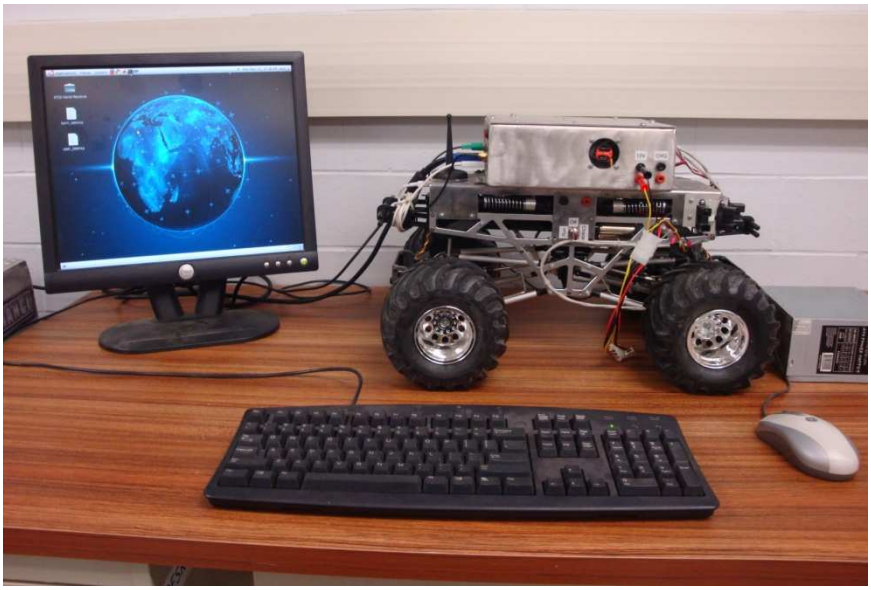

Figure 6.1: Rover connected to monitor, keyboard, and mouse

6.2 SOFTWARE

The speed of the onboard computer was adequate for the control schemes developed in this project. When combined with RTAI, control schemes executed accurately. The speed of the computer presented no limitation for the real-time computation of control schemes during the scope of these experiments.

A library of Simulink® blocks was developed to allow for rapid development of new control schemes. These blocks have been described throughout Chapters 4 and 5. The full library of blocks can be seen below in Figure 6.2. This library can be opened by executing *roverblocks* from the Matlab® command line.

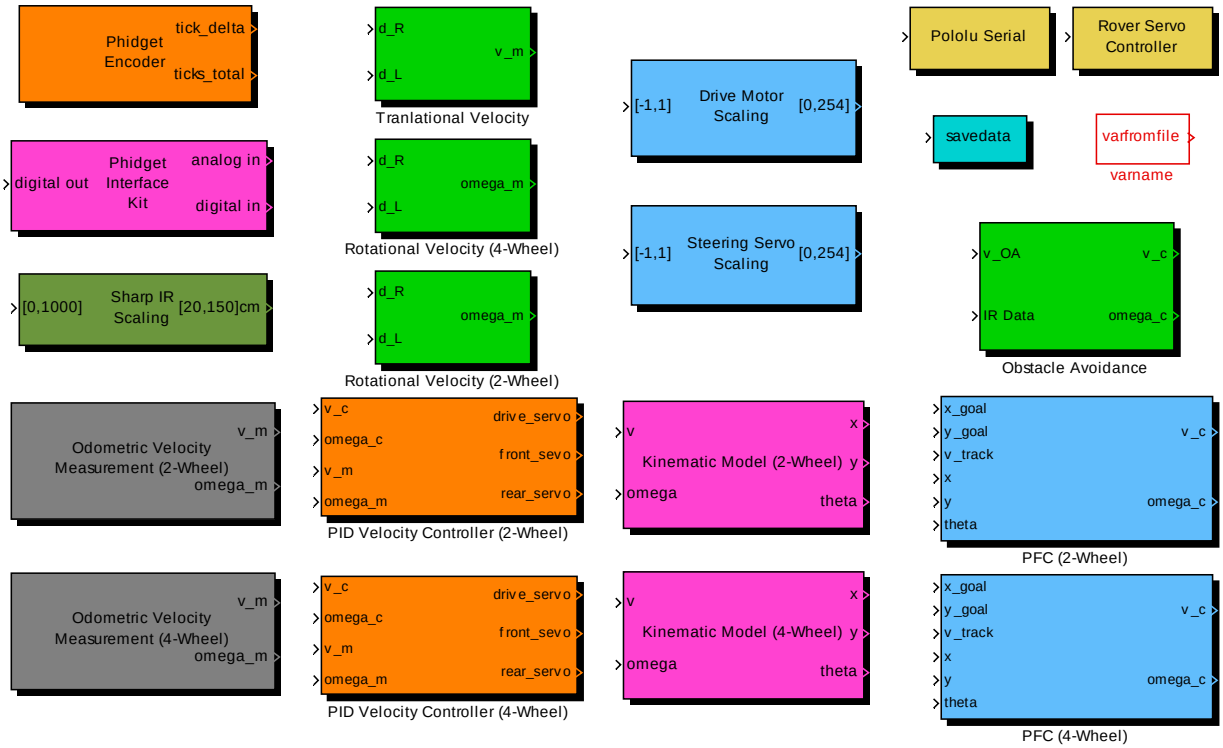

Figure 6.2: *Roverblocks* Simulink® library

6.3 CONTROL

6.3.1 PID VELOCITY CONTROLLER

The velocity estimation equations presented in Section 5.2.1 were verified experimentally prior to tuning the PID velocity controller. When tuning the gains for the PID controller, the translational and rotational velocities were tuned independently, with the translational velocity being tuned first. When tuning manually, the integral and derivative gains were initially set to zero. The proportional gain was slowly increased until the response started to oscillate. Then the proportional gain was reduced to approximately two-thirds of its oscillation value. Next, the integral gain was increased until the steady-state error of the response went to zero. Finally, the derivative gain was adjusted to provide a quicker response. Note that for the translational velocity controller, no over-shoot was allowed as over-shoot caused the rover to exhibit a jerky response. The response of the translational velocity controller to a series of step inputs can be seen in Figure 6.3.

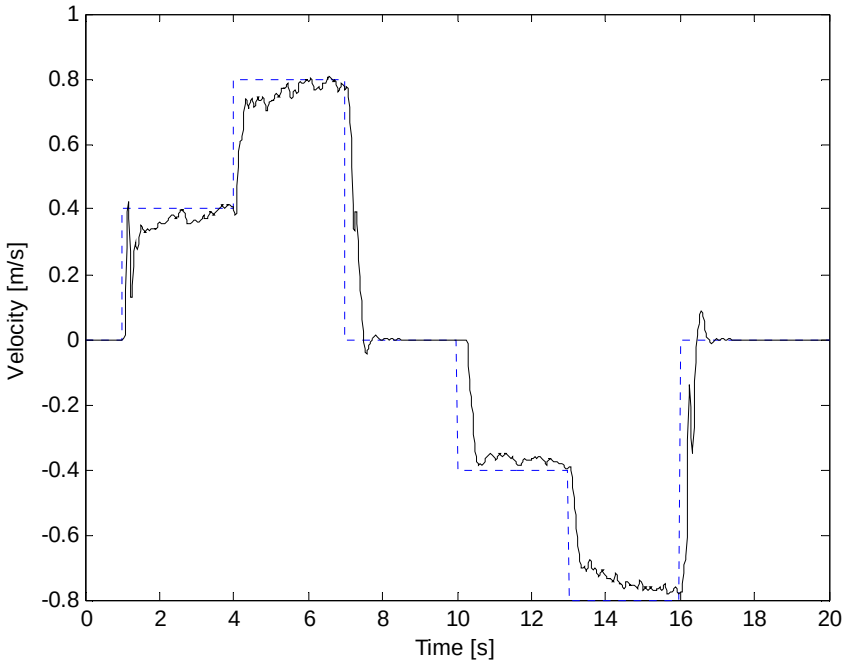

Figure 6.3: Translational velocity controller response

Two different rotational velocity controllers were developed, one for front-wheel steering and one for four-wheel steering. These were developed using the method described above. After extensive tuning, the response of the four-wheel steering controller was considerably better than the two-wheel steering controller. The response of the rotational velocity controllers to a series of step inputs can be seen in Figure 6.4 and Figure 6.5 for the front-wheel and four-wheel controllers respectively.

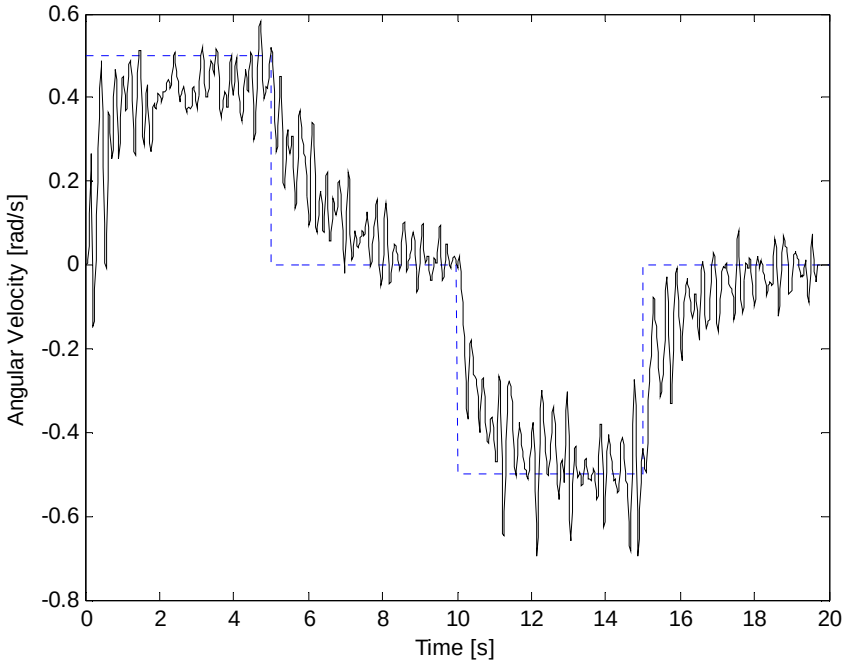

Figure 6.4: Two-wheel steering rotational velocity controller response

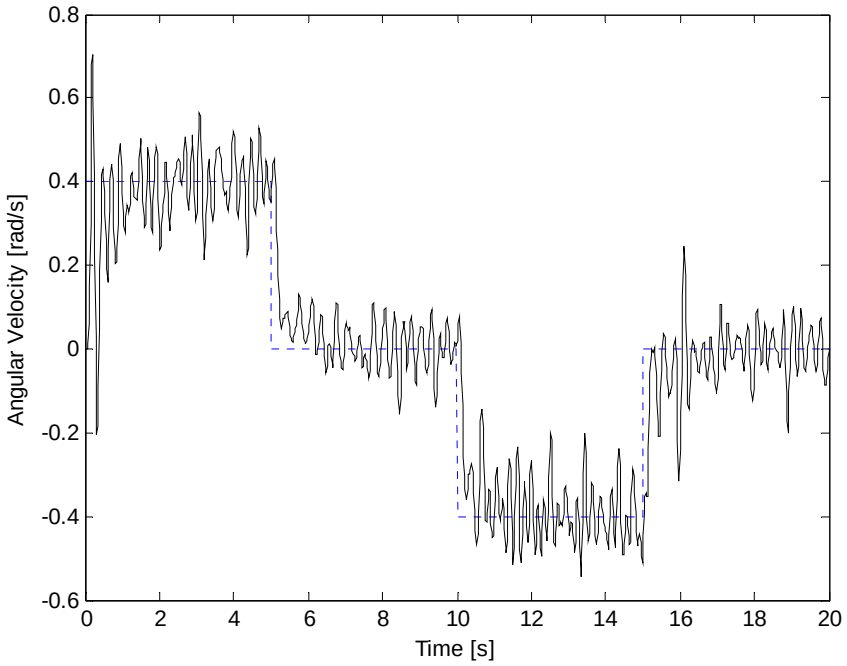

Figure 6.5: Four-wheel steering rotational velocity controller response

6.3.2 POTENTIAL FIELD CONTROLLER

After successfully tuning the PID velocity controllers, they could be integrated into the potential field controller. Separate PFCs were designed for each of two steering schemes, since the steering scheme limits the turning radius of the vehicle. The performance of the PFC was quite accurate, as can be seen in Figure 6.6. This example is for four-wheel steering, which performed better since it can obtain a tighter turning radius. Only examples for four-wheel steering will be illustrated from now on. The reference path given in this figure is calculated using the outputs from the PFC controller fed directly back into the kinematic model to give the output if the rover could perfectly track the commanded velocities. However, more error is introduced into to the final position of the rover due to localization errors from slippage and encoder sampling (i.e. even when the rover thinks it has reached the goal, it may still be some distance away). This experiment was conducted five times to show the rover's localization error. This error can be seen in Figure 6.7.

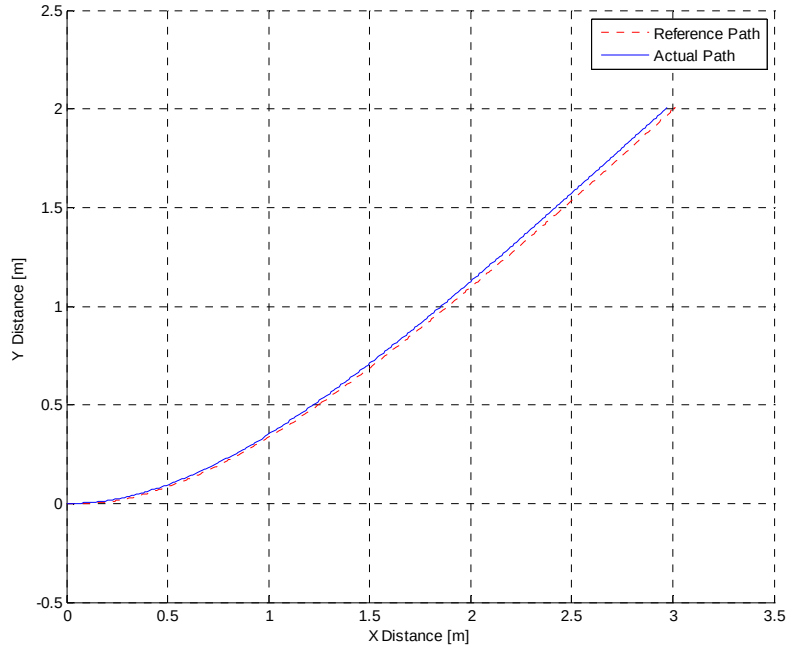

Figure 6.6: PFC response

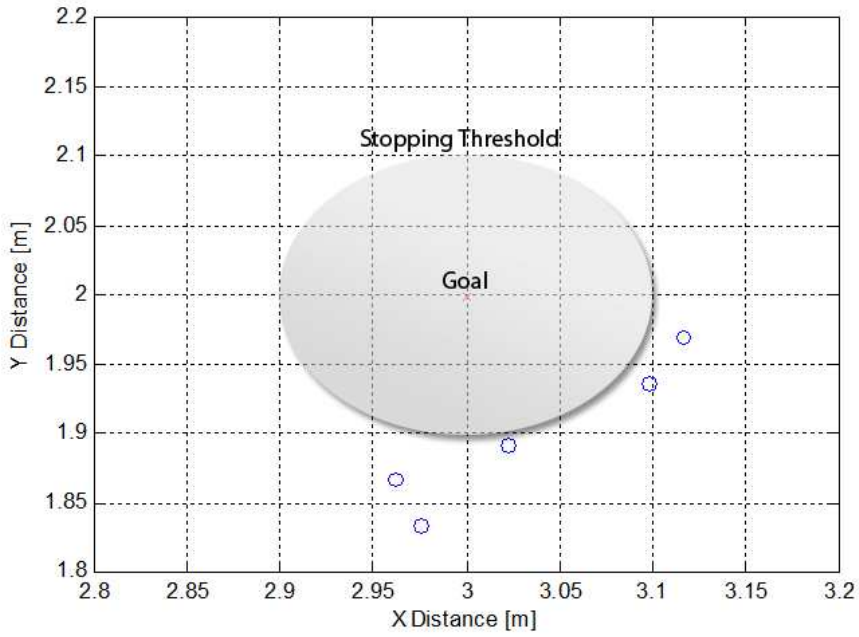

Figure 6.7: Localization error

6.3.3 HYBRID CONTOLLER

Once the PFC and OA controllers were designed, they were combined into a hybrid waypoint tracking controller. To test the functionality of this controller, an *obstacle course* was created. The rover was given a goal that was blocked by both a wall and a rectangular object. The rover was able to successfully navigate the course and find its goal. Figure 6.8 shows the path taken by the rover in the navigating to the goal. Figure 6.9 shows the distribution of control between the PFC and OA controllers.

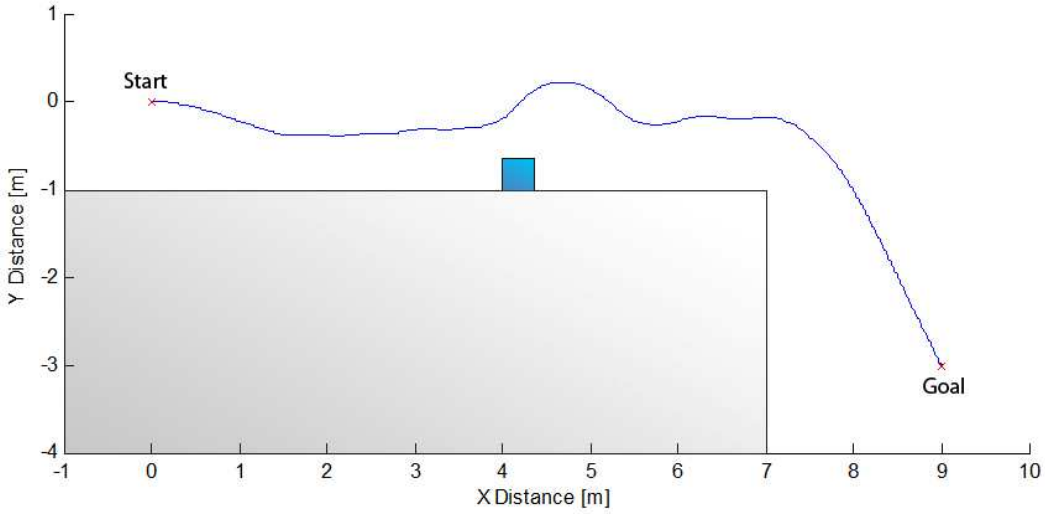

Figure 6.8: Hybrid controller response to obstacle course

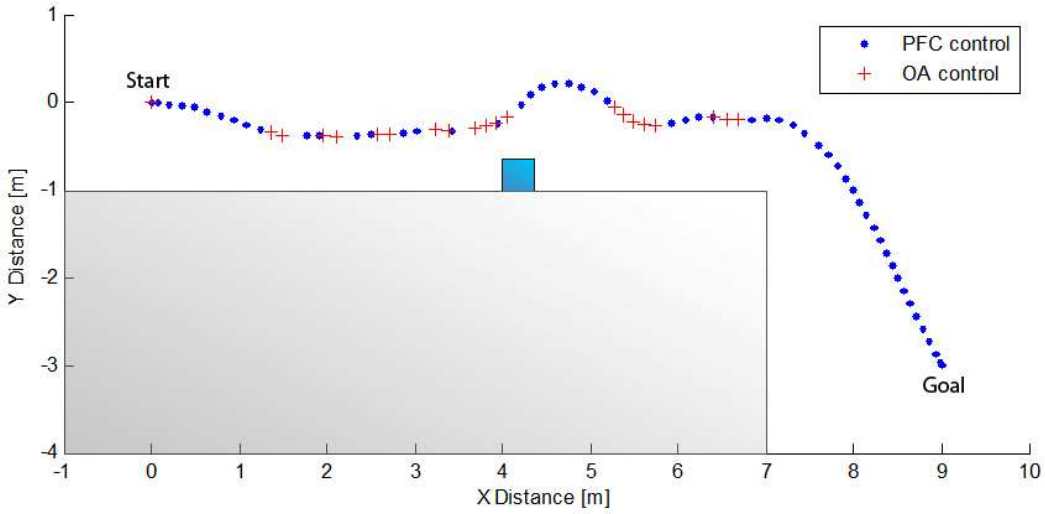

Figure 6.9: Distribution of hybrid control

CHAPTER 7 - CONCLUSIONS

7.1 CONCLUSIONS

In this thesis, a Tamiya TXT-1 RC vehicle was successfully modified into a mobile robotics research platform. This is capable of either front-wheel or four-wheel steering. Through the course of experimentation, several conclusions can be drawn. First, four-wheel steering greatly improves indoor navigation. Also, the original suspension components must be stiffened in order to facilitate a larger payload and eliminate sway in the body. Additionally, a higher capacity battery is needed to provide ample run time.

The onboard Mini-ITX computer developed for the rover provided greater than adequate processing capabilities for real-time control scheme execution. The Pololu servo controller, Phidget encoder readers, and Phidget interface kit all performed as required. However, the encoders introduced a high level of noise in the velocity estimation which required filtering. Additionally, the IR sensors provided a simple means for indoor obstacle avoidance. However, due to interference from sunlight, these are unsuitable for outdoor operation.

Linux RTAI, coupled with Matlab®/Simulink® Real-Time Workshop, proved to be an extremely valuable tool for real-time code generation. However, both software packages have a steep learning curve for someone unfamiliar with the Linux environment. A library of Simulink® blocks was developed to facilitate basic control of the rover.

The kinematic model was proven accurate for control of the rover. Odometric velocity measurements were used to construct a PID feedback controller for translational and rotational velocities. A PFC was utilized to provide waypoint tracking. An obstacle avoidance controller was successfully implemented using distance measurements from the IR sensor array. The PFC and OA

controllers were combined to form a hybrid waypoint tracking controller. This hybrid control scheme was used to successfully navigate cluttered environments to a goal.

7.2 RECOMMENDATIONS FOR FUTURE WORK

The electronic speed controller (ESC) used on the rover was sufficient for the experiments conducted in this research. However, the smart-braking feature, which cannot be turned off, proved to be an unwarranted nuisance. As a consequence of this feature, the translational velocity controller could not display any overshoot, or the behavior of the rover would become too jerky. Because of this inability to exhibit overshoot, the rise-time of the translational velocity response was limited. Thus it is recommended that any future platforms based on this research find a more suitable ESC.

It is believed that the Mini-ITX computer was the best design choice at the time of construction. However, since the completion of the project, the speed and price of computers have improved. Therefore, it is recommended that this be reevaluated before the construction of another rover. In particular, the price of solid-state hard drives has decreased rapidly since the onset of this project. Their tolerance to vibration could improve the overall resilience of the platform. Also, it is recommended that circuitry be added to the current computer power supply/battery to allow for simultaneous usage and charging.

The optical encoders on the wheels served a simple solution for localization. However, small errors in initial placement, distance truncation between time samples, and slippage in the drive-train can propagate into significant error over a long trial. Consequently, a second form of localization such as GPS, machine vision, etc. should be added for increased accuracy.

QRtaiLab, and the *varfromfile* Simulink® block were used in conjunction with remote desktop in order to tune control scheme parameters and to select missions. However, a more

polished GUI could be designed to allow for simpler interaction. Additionally, the promising networking capabilities of QRtaiLab were never realized. However, subsequent versions of RTAI and QRtaiLab may remedy this problem. As such, it is recommended to update the RTAI suite to see if this problem has been solved.

Finally, it is recommended that the IR sensors be used to construct primitive maps of the rover's environment. This could possibly aid the localization process by comparing previously encountered obstacles with the estimated position. Additionally, *a priori* knowledge of the environment could be used to plan more optimal paths to the goal.

BIBLIOGRAPHY

1. *A Real-Time Linux System for Autonomous Navigation and Flight Attitude Control of UAV.* **Hall Jr, Charles E.** Daytona Beach, FL : IEEE, 2001. 20th Conference on Digital Avionics Systems . pp. 1A1/1-1A1/9.

2. **Grocholsky, B, et al.** Cooperative Air and Ground Surveillance. *Robotics & Automation Magazine.* September 21, 2006, Vol. 13, 3, pp. 16-25.

3. *Unmanned Air Vehicle Testbed for Cooperative Control Experiments.* **McLain, Tim and Beard, Randy.** s.l. : IEEE, 2004. American Control Conference. pp. 5327-5331.

4. *Coordination and Control Experiments on a Multi-vehicle Testbed.* **King, Ellis, et al.** s.l. : IEEE, 2004. American Control Conference. pp. 5315-5320.

5. *Autonomous Vision-based Landing and Terrain Mapping Using an MPC-controlled Unmanned Rotorcraft.* **Templeton, Todd, et al.** Roma : IEEE, 2007. International Conference on Robotics and Automation. pp. 1349-1356.

6. *3 Aircraft Formation Flight Experiment.* **Seanor, B, et al.** Ancona, Italy : IEEE, 2006. 14th Mediterranean Conference on Control and Automation. pp. 1-6.

7. *Design and Flight Testing Evaluation of Formation Control Laws.* **Gu, Yu, et al.** s.l. : IEEE, 2006. IEEE Transactions on Control Systems Technology. Vol. 14, pp. 105-1112.

8. *MVWT-II: The Second Generation Caltech Multi-Vehicle Wireless Testbed.* **Jin, Z, et al.** s.l. : IEEE, 2004. American Control Conference. pp. 5321-5326.

9. *The Caltech Multi-Vehicle Wireless Testbed.* **Cremean, Lars, et al.** s.l. : IEEE, 2002. 41st Conference on Decision and Control. pp. 86-88.

10. **Stubbs, A, et al.** Multivehicle Systems Control Over Networks: A Hovercraft Testbed for Networked and Decentralized Control. *Control Systems Magazine.* June 2006, Vol. 26, 3, pp. 56-69.

11. *An Autonomous Servellance and Security Robot Team.* **Li, Tzuu-Hseng S, et al.** Hsinchu : IEEE, 2007. Workshop on Advanced Robotics and Its Social Impacts.

12. *Design and Construction of an Autonomous Fire Fighting Robot.* **Altaf, K, Akbar, A and Ijaz, B.** Karachi : IEEE, 2007. International Conference on Information and Emerging Technologies. pp. 1-5.

13. *Treaded Control System for Rescue Robots in Indoor Environment.* **Mano, H, et al.** Bangkok : IEEE, 2008. International Conference on Robotics and Biominetrics. pp. 1836-1843.

14. *Decentralized Cooperative Control: A Multivehicle Platform for Research in Networked Embedded Systems.* **Cruz, Daniel, et al.** June 2007, IEEE Control Systems Magazine, pp. pp58-78.

15. *A Mobile Robot Platform with DSP-based Controller and Omnidirectional Vision System.* **Xiao, Jizhong, et al.** 2004. Proceedings of the 2004 IEEE International Conference on Robotics and Biometrics. pp. pp844-848.

16. *A Real-Time Autonomous Rover Navigation System.* **Howard, Ayanna and Seraji, Homayoun.** Maui, HI : s.n., 2000. World Automation Congress.

17. *Real-Time Vision-Based Control of a Nonholonomic Mobile Robot.* **Das, A K, et al.** s.l. : IEEE, 2001. International Conference on Robotics and Automation. pp. 1714-1719.

18. **Spletzer, John R.** *Sensor Fusion Techniques for Cooperative Localization in Robot Teams.* 2003. Disseration.

19. *Experiments in Multirobot Coordination.* **Marshall, Joshua A, et al.** 2006, Robotics and Autonomous Systems, pp. pp265-275.

20. **Huang, Edward.** *A Semi-Autonomous Vision-Based Navigation System for a Mobile Robotic Vehicle.* Cambridge, Massachusetts : s.n., 2003. Master's Thesis.

21. *An Autonomous Robotic System for Mapping Abandoned Mines.* **Ferguson, D, et al.** s.l. : MIT Press, 2003. Conference on Neural Information Processing Systems.

22. **Abou-Samah, Michel.** *A Kinematically Compatible Framework for Collaboration of Multiple Non-holonomic Wheeled Mobile Robots.* Montreal, Canada : s.n., 2001. Master's Thesis.

23. **Marshall, Joshua A.** *Coordinated Autonomy: Pursuit Formations of Multivehicle System.* Toronto, Canada : s.n., 2005. PhD Thesis.

24. *AxeBot Robot the Mechanical Design for an Autonomous Omnidirectional Mobile Robot.* **do Nascimento, T P, da Costa, A L and Paim, C C.** Cuernavaca, Morelos : IEEE, 2009. Electronics, Robotics and Automotive Mechanics Conference. pp. 187-192.

25. *A Model-predictive Approach to Formation Control of Omnidirectional Mobile Robots.* **Kanjanawanishkul, K and Zell, A.** Nice : IEEE, 2008. International Conference on Intelligent Robots and Systems. pp. 2771-2776.

26. *Platform Development of an Omnidirectional Mobile Robot for the Elderly's Walking Support and the Caregiver's Power Assistance.* **Zhu, Chi, et al.** Guilin : IEEE, 2009. Internation Conference on Robotics and Biometrics. pp. 1900-1905.

27. **Moore, K L and Flann, N S.** A Six-wheeled Omnidirectional Autonomous Mobile Robot. *Control Systems Magazine.* December 2000, Vol. 20, 6, pp. 53-66.

28. *Real Time Controller for a Nonholonomic Mobile Robot.* **Majchrzak, J and Michalski, M.** s.l. : IEEE, 2004. Fourth International Workshop on Robot Motion and Control. pp. 357-362.

29. *Control Architecture and Obstacle Avoidance for Acquiring the Realtimeness.* **Sohn, Won-Jong, et al.** Seoul : IEEE, 2007. International Conference on Control, Automation, and Systems. pp. 212-217.

30. **Siegwart, Roland and Nourbakhsh, Illah R.** *Introduction to Autonomous Mobile Robots.* Cambridge, MA : MIT Press, 2004.

31. *A Framework and Architecture for Multi-Robot Coordination.* **Fierro, Rafael, et al.** November 2002, The International Jounal of Robotics Research, Vol. Vol. 21, pp. 977-995.

32. *Development of an Automatic Parking System for Vehicle.* **Hsu, Tsung-hua, et al.** Harbin, China : IEEE, 2008. Vehicle Power and Propulsion Converence. pp. 1-6.

33. *A Low Cost Modular Autonomous Robot Vehicle.* **Edwan, Ezzaldeen and Fierro, Rafael.** Cookeville, TN : IEEE, 2006. 38th Southeastern Symposium on System Theory. pp. 245-249.

34. *Avoiding Obstacles in Mobile Robot Navigation - Implementing the Tangential Escape Approach.* **Ferreira, A, et al.** Montreal, Quebec : IEEE, 2006. International Symposium on Industrial Electronics. pp. 2732-2737.

35. **Moret, Eric N.** *Dynamic Modeling and Control of a Car-like Robot.* Blacksburg, VA : Virginia Polytechnic Institute and State University, 2003. Thesis.

36. **De Luca, A, Oriolo, G and Samson, C.** Feedback Control of a Nonholonomic Car-Like Robot. *Robot Motion Planning and Control.* s.l. : Springer-Verlag, 1998, pp. pp171-253.

37. **Morin, Pascal and Samson, Claude.** Motion Control of Wheeled Mobile Robots. [book auth.] Bruno Sicilian and Oussama Khatib. *Springer Handbook of Robotics.* Berlin, Heidelberg, Germany : Springer-Verlag, 2008, pp. 799-826.

38. *Navigable Voronoi Diagram: A Local Path Planner for Mobile Robots Using Sonar Sensors.* **Lee, Kyoungmin and Chung, Wan Kyun.** San Diego, CA : IEEE, 2007. International Conference on Intelligent Robots and Systems. pp. 2813-2818.

39. *Voronoi Based Coverage Control for Nonholonomic Mobile Robots with Collision Avoidance.* **Dirafzoon, A., Menhaj, M. B. and Afshar, A.** Yokohama, Japan : IEEE, 2010. IEEE International Conference on Control Applications. pp. 1755-1760.

40. *Local Incremental Planning for a Car-like Robot Navigating Among Obstacles.* **Bemporad, Alberto, De Luca, Alessandro and Oriolo, Giuseppe.** s.l. : IEEE, 1996. International Conference on Robotics and Automation. Vol. 2, pp. 1205-1211.

41. **Minguez, Javier, Lamiraux, Florent and Laumond, Jean-Paul.** Motion Planning and Obstacle Avoidance. [book auth.] Bruno Sicilian and Oussama Khatib. *Springer Handbook of Robotics.* Berlin, Heidelberg, Germany : Springer-Verlag, 2008, pp. 827-852.

42. *Practical Obstacle Avoidance Using Potential Field for a Nonholonmic Mobile Robot with Rectangular Body.* **Seki, H., et al.** Hamburg : IEEE, 2008. IEEE International Conference on Emerging Technologies and Factory Automation. pp. 326-332.

43. *Dynamic Path Planning for Mobile Robots Using Fractional Potential Field.* **Poty, A., Melchior, P. and Oustaloup, A.** s.l. : IEEE, 2004. First International Symposium on Control, Communications, and Signal Processing. pp. 557-561.

44. *Industrial Motion Control Applications Using Linux RTAI.* **Chiandone, M, et al.** Ischia : IEEE, 2008. International Symposium on Power Electronics, Electrical Drives, Automation and Motion. pp. 528-533.

45. *Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in Hard Real-Time Application.* **Barbalace, A., et al.** s.l. : IEEE, 2008. IEEE Transactaions on Nuclear Science. pp. 435-439.

46. **Nahrstaedt, Holge.** Installation RTAI. *QRtaiLab: A User Interface for RTAI.* [Online] 2009. [Cited: January 23, 2010.] http://qrtailab.sourceforge.net/rtai_installation.html.

47. **RTAI.** An Overview of RTAI Schedulers. *RTAI API Documentation.* [Online] 2005. [Cited: May 10, 2010.] https://www.rtai.org/documentation/magma/html/api/sched_overview.html.

48. **Monforte, Jorge Cortes.** *Geometric, Control, and Numerical Aspects of Nonholonomic Systems.* Berlin, Germany : Springer-Verlag, 2002.

49. **Mataric, Maja J.** *The Robotic Primer.* Cambridge, MA : MIT Press, 2007.

50. **Dudek, Gregory and Jenkin, Michael.** *Computational Principles of Mobile Robotics.* Cambridge, MA : Cambridge University Press, 2000.

51. **Choset, Howie, et al.** *Principles of Robot Motion: Theory, Algorithms, and Implementations.* Cambridge, MA : MIT Press, 2005.

52. **Bloch, A M, et al.** *Nonholonomic Mechanics and Control.* New York, NY : Springer-Verlag, 2003.

53. **Bekey, George A.** *Autonomous Robots.* Cambridge, MA : MIT Press, 2005.

54. **Abbott, Doug.** *Linux for Embedded and Real-time Applications.* Burlington, MA : Elsevier Science, 2003.

55. *Study and Comparison of the RTHAL-based and ADEOS-based.* **Zhang, Guoyin, Chen, Luyuan and Yao, Aihong.** Hanzhou, Zhejiang : IEEE, 2006. First International Multi-Symposiums on Computer and Computational Sciences. pp. 771-775.

56. *Control of a Prototype Transmission-based Robot Servoactuator using Real Time Application Interface.* **Zhou, R and Hamel, W R.** Orlando, FL : IEEE, 2006. International Conference on Robotics and Automation. pp. 375-280.

57. *RePLiCS: An Environment for Open Real-Time Control of a Dual-Arm Industrial Robotic Cell Based on RTAI-Linux.* **Caccavale, F, et al.** s.l. : IEEE, 2005. International Conference on Intelligent Robots and Systems. pp. 2493-2498.

58. **De Luca, A., Oriolo, G. and Samson, C.** Feadback Control of a Nonholonomic Car-like Robot. [book auth.] J.-P. Laumond. *Robot Motion Planning and Control.* London, U.K. : Springer-Verlag, 1998.

59. **Campion, Guy and Chung, Woojin.** Wheeled Robots. [book auth.] Bruno Sicilian and Oussama Khatib. *Springer Handbook of Robotics.* Berlin, Heidelberg, Germany : Springer-Verlag, 2008, pp. 391-410.

APPENDIX A - HARDWARE PRICE AND SOURCE LIST

A.1 VEHICLE AND ACCESSORIES

Item	Source	Price
DuraTrax 6-Cell 7.2V DTX4200 NiMH Battery	www.towerhobbies.com	$41.49
DuraTrax Bearing 6x12mm (x10)	www.towerhobbies.com	$14.99
DuraTrax Old Kyosho Battery Connector Wire	www.towerhobbies.com	$1.49
ElectricFly Triton Jr DC Computer Charger	www.towerhobbies.com	$69.99
Hobbico Banana Plugs	www.towerhobbies.com	$3.49
Novak Super Duty XR ESC	www.teamnovak.com	$89.99
Tamiya Steering Knuckles 58065/89	www.towerhobbies.com	$8.59
Tamiya Rear Stub Axles 58280	www.towerhobbies.com	$14.99
Tamiya Silicone Dampener Oil Hard	www.towerhobbies.com	$11.99
Tamiya TXT-1	www.towerhobbies.com	$429.99
TS-70MG Super-Torque 2BB Servo	www.towerhobbies.com	$34.99

A.2 ON-BOARD COMPUTER

Item	Source	Price
Edimax EW-7128G PCI Wireless Card	www.newegg.com	$22.99
Evercool EC4010 40mm Case Fan (x4)	www.newegg.com	$11.56
MSI Fuzzy GM965 (MSI-9803)	www.itxdepot.com	$227.00
Patriot Performance 4GB (2x2GB) DDR2 RAM	www.newegg.com	$87.99
Pico PSU-120-WI-25	www.itxdepot.com	$55.00
Powerizer Polymer Li-ion 14.8V 5700mAh	www.batteryspace.com	$89.95

RS-232 for Intel Motherboards	www.itxdepot.com	$10.00
Western Digital 80GB 5400RPM SATA Hard Drive	www.newegg.com	$54.99

A.3 SENSORS AND CONTROLLER BOARDS

Item	Source	Price
Cable Assembly (encoders), Shielded - CA-FC5-SH-MIC4-2 (x2)	www.usdigital.com	$32.36
Encoder Disk - HUBDISK-E4P-256-236 (x2)	www.usdigital.com	$18.20
GP2Y0A02YK IR Sensor Kit (8-60 inches) (x6)	www.trossenrobotics.com	$154.50
Miniature Shaft-Mount Optical Encoder, 256 CPR 4mm Shaft - E4-256-157-D-H-T-B (x2)	www.usdgitial.com	$70.04
PhidgetEncoder High Speed Readers (x2)	www.phidgets.com	$120.00
Phidgets Interface Kit 8/8/8	www.trossenrobotics.com	$77.95
Pololu Serial 8-Servo Controller	www.pololu.com	$26.95

APPENDIX B - CODE

B.1 POLOLU SERIAL CONTROLLER

```
/*
 * sfuntmpl_basic.c: Basic 'C' template for a level 2 S-function.
 *
 *  -------------------------------------------------------------------------
 *  | See matlabroot/simulink/src/sfuntmpl_doc.c for a more detailed template
 *  -------------------------------------------------------------------------
 *
 * Copyright 1990-2002 The MathWorks, Inc.
 * $Revision: 1.27.4.1 $
 */



/*
 * You must specify the S_FUNCTION_NAME as the name of your S-function
 * (i.e. replace sfuntmpl_basic with the name of your S-function).
 */

#define S_FUNCTION_NAME roverservos
#define S_FUNCTION_LEVEL 2

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */
#include "simstruc.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>

/* RTAI headers for */
#include <rtai_serial.h>
#include <rtai_lxrt.h>

/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes
=============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl_doc.c for more details on the macros below */

    ssSetNumSFcnParams(S, 3);  /* Number of expected parameters */
```

```
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    /* Create 1 input port */
    if (!ssSetNumInputPorts(S, 1)) return;
    /* Size port based on size of input vector */
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal
access*/
    /*
     * Set direct feedthrough flag (1=yes, 0=no).
     * A port has direct feedthrough if the input is used in either
     * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
     * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
     */
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    /* No output ports */
    if (!ssSetNumOutputPorts(S, 0)) return;

    ssSetNumSampleTimes(S, 1);

    /* Create 2 work variables for handling smart-braking */
    ssSetNumRWork(S, 2);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}


/* Function: mdlInitializeSampleTimes
=========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /* Set sample time based on given parameter */
    double *dSampleTime = mxGetPr(ssGetSFcnParam(S,2));
    ssSetSampleTime(S, 0, *dSampleTime);
    ssSetOffsetTime(S, 0, 0.0);
}


#define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
```

```
#if defined(MDL_INITIALIZE_CONDITIONS)
  /* Function: mdlInitializeConditions
========================================
   * Abstract:
   *    In this function, you should initialize the continuous and discrete
   *    states for your S-function block.  The initial states are placed
   *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
   *    You can also perform any other initialization activities that your
   *    S-function may require. Note, this routine will be called at the
   *    start of simulation and if it is present in an enabled subsystem
   *    configured to reset states, it will be call when the enabled
subsystem
   *    restarts execution to reset the states.
   */
  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */


#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart
=======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
  static void mdlStart(SimStruct *S)
  {
      int res;
      unsigned char buf[3];
      double *dPort = mxGetPr(ssGetSFcnParam(S,0));
      unsigned int nPort;
      real_T *RWork = ssGetRWork(S);

      RWork[0] = 0.0;
      RWork[1] = 0.0;

      /* Set COM port */
      switch ((int)*dPort) {
          case 1:
              nPort = COM1;
              break;
          case 2:
              nPort = COM2;
              break;
      }

      /* Open COM port */
      res = rt_spopen(nPort, 9600, 8, 1, RT_SP_PARITY_NONE,
RT_SP_NO_HAND_SHAKE, RT_SP_FIFO_DISABLE);

      if(res) {
          printf("Error: rt_spopen\n");
```

```
            switch(res) {
                case -ENODEV:
                    printf("No device %d\n", res);
                    break;
                case -EINVAL:
                    printf("Invalid val %d\n", res);
                    break;
                case -EADDRINUSE:
                    printf("Address in use %d\n", res);
                    break;
                    default:
                        printf("Unknown %d\n", res);
                        break;
            }
            exit(0);
        }

        rt_spclear_rx(nPort);
        rt_spclear_tx(nPort);


        /* Initialize servo controller */
        buf[0] = 0xFF;
        buf[1] = 0x00;
        buf[2] = 127;
        rt_spwrite(nPort, buf, sizeof(buf));
    }
#endif /*  MDL_START */


/* Function: mdlOutputs
=======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    int           i;
    const real_T  *u = (const real_T*) ssGetInputPortSignal(S,0);
    real_T        nu = ssGetInputPortWidth(S,0);
    int           buflength = (int)nu;
    real_T        *RWork = ssGetRWork(S);

    unsigned char buf[3*(int)nu];
    double *dPort = mxGetPr(ssGetSFcnParam(S,0));
    unsigned int nPort;
    double *switchDelay = mxGetPr(ssGetSFcnParam(S,1));

    /* Set COM port */
    switch ((int)*dPort) {
        case 1:
            nPort = COM1;
            break;
        case 2:
```

```
            nPort = COM2;
            break;
    }

    /* Create buffer for servo commands */
    for (i = 0; i < nu; i++)
    {
        buf[(i*3)]      = 0xFF;
        buf[(i*3)+1]    = (unsigned char)i;
        buf[(i*3)+2]    = (unsigned char)u[i];
    }

    /* Check and correct for smart-braking */
    if (((double)RWork[0] >= 127) && ((double)u[0] < 127))
    {
        RWork[1] = 0.0;
    }
    if ((double)u[0] < 127)
    {
        RWork[1] += 1.0;
    }
    if (((double)RWork[1] > ((double)*switchDelay)/2.0) && ((double)RWork[1]
<= (double)*switchDelay) && ((double)u[0] < 127))
    {
        buf[2] = 127;
    }

    /* Write servo command buffer to COM port */
    rt_spwrite(nPort, buf, sizeof(buf));

    /* Store previous input */
    RWork[0] = (double)u[0];
}


#define MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate
======================================================
   * Abstract:
   *    This function is called once for every major integration time step.
   *    Discrete states are typically updated here, but this function is
useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */


#define MDL_DERIVATIVES  /* Change to #undef to remove function */
#if defined(MDL_DERIVATIVES)
```

```
  /* Function: mdlDerivatives
=============================================
   * Abstract:
   *    In this function, you compute the S-function block's derivatives.
   *    The derivatives are placed in the derivative vector, ssGetdX(S).
   */
  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */


/* Function: mdlTerminate
=====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    int                 i;
    double              *dPort = mxGetPr(ssGetSFcnParam(S,0));
    real_T              nu = ssGetInputPortWidth(S,0);
    unsigned char       buf[3];
    unsigned int        nPort;

    /* Set COM port */
    switch ((int)*dPort) {
        case 1:
            nPort = COM1;
            break;
        case 2:
            nPort = COM2;
            break;
    }

    /* Return servos to neutral position */
    for (i = 0; i < nu; i++)
    {
        buf[(i*3)]      = 0xFF;
        buf[(i*3)+1]    = (unsigned char)i;
        buf[(i*3)+2]    = 127;
    }

    rt_spwrite(nPort, buf, sizeof(buf));

    sleep(1);
    /* Close COM port */
    printf("Closing COM port\n");
    rt_spclose(nPort);
}


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

```
 * See sfuntmpl_doc.c for the optional S-function methods *
 *=====================================================*/

/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

B.2 PHIDGET ENCODER READER

```
/*
 * sfuntmpl_basic.c: Basic 'C' template for a level 2 S-function.
 *
 *  -------------------------------------------------------------------------
 *  | See matlabroot/simulink/src/sfuntmpl_doc.c for a more detailed template
 *  -------------------------------------------------------------------------
 *
 * Copyright 1990-2002 The MathWorks, Inc.
 * $Revision: 1.27.4.1 $
 */


/*
 * You must specify the S_FUNCTION_NAME as the name of your S-function
 * (i.e. replace sfuntmpl_basic with the name of your S-function).
 */

#define S_FUNCTION_NAME  phidget_encoder
#define S_FUNCTION_LEVEL 2

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */
#include "simstruc.h"
#include <stdio.h>
#include <phidget21.h>


/*====================*
 * S-function methods *
 *====================*/


/* Function: mdlInitializeSizes
=============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl_doc.c for more details on the macros below */

    ssSetNumSFcnParams(S, 2);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

    ssSetNumContStates(S, 0);
```

```
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 0)) return;
    /* ssSetInputPortWidth(S, 0, 1); */
    /* ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal
access*/
    /*
     * Set direct feedthrough flag (1=yes, 0=no).
     * A port has direct feedthrough if the input is used in either
     * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
     * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
     */
    /* ssSetInputPortDirectFeedThrough(S, 0, 1); */

    /* Define 2 output ports */
    if (!ssSetNumOutputPorts(S, 2)) return;
    /* One dimension per port */
    ssSetOutputPortWidth(S, 0, 1);
    ssSetOutputPortWidth(S, 1, 1);

    ssSetNumSampleTimes(S, 1);
    /* One work vector to hold tick count */
    ssSetNumRWork(S, 1);
    ssSetNumIWork(S, 0);
    /* Pointer to hold Phidgets handle */
    ssSetNumPWork(S, 1);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}


/* Function: mdlInitializeSampleTimes
=======================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /* Set sample time based on input parameters */
    double *dSampleTime;
    dSampleTime = mxGetPr(ssGetSFcnParam(S,1));

    ssSetSampleTime(S, 0, *dSampleTime);
    ssSetOffsetTime(S, 0, 0.0);
}



#undef MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
#if defined(MDL_INITIALIZE_CONDITIONS)
```

```
  /* Function: mdlInitializeConditions
========================================
   * Abstract:
   *    In this function, you should initialize the continuous and discrete
   *    states for your S-function block.  The initial states are placed
   *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
   *    You can also perform any other initialization activities that your
   *    S-function may require. Note, this routine will be called at the
   *    start of simulation and if it is present in an enabled subsystem
   *    configured to reset states, it will be call when the enabled
subsystem
   *    restarts execution to reset the states.
   */
  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */


#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart
=====================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
    int result;
    int serialNumber;
    const char *err;
    double *dSerialNumber;
    void **PWork = ssGetPWork(S);
    real_T *RWork = ssGetRWork(S);

    RWork[0] = 0.0;

    /* Get serial number from parameters (if defined) */
    dSerialNumber = mxGetPr(ssGetSFcnParam(S,0));

    /* Declare an encoder handle */
    CPhidgetEncoderHandle encoder = 0;

    /* Create the encoder object */
    CPhidgetEncoder_create(&encoder);

    /* Open the encoder */
    CPhidget_open((CPhidgetHandle)encoder, (int)(*dSerialNumber));

    printf("Waiting for encoder to be attached....\n");
    if((result = CPhidget_waitForAttachment((CPhidgetHandle)encoder, 2000)))
    {
        CPhidget_getErrorDescription(result, &err);
```

```
        printf("Problem waiting for attachment: %s\n", err);
    }

    /* Display serial number */
    CPhidget_getSerialNumber((CPhidgetHandle)encoder, &serialNumber);
        printf("Serial Number: %d\n", serialNumber);

    /* Pass along Phidgets handle */
    PWork[0] = encoder;


}
#endif /*  MDL_START */



/* Function: mdlOutputs
=====================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{

    int value;
    real_T       *y = ssGetOutputPortSignal(S,0);
    real_T       *RWork = ssGetRWork(S);


    if (ssGetPWork(S) != NULL) {
        CPhidgetEncoderHandle *encoder;
        /* Get Phidgets handle */
        encoder = (CPhidgetEncoderHandle *) ssGetPWorkValue(S,0);

        /* Get current position from */
        CPhidgetEncoder_getPosition((CPhidgetEncoderHandle)encoder, 0,
&value);
    }

    /* Update outputs */
    y[0] = (double)value - RWork[0];;
    y[1] = (double)value;
    RWork[0] = (double)value;
}



#undef MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate
======================================================
   * Abstract:
   *    This function is called once for every major integration time step.
```

```
   *    Discrete states are typically updated here, but this function is
useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */



#undef MDL_DERIVATIVES  /* Change to #undef to remove function */
#if defined(MDL_DERIVATIVES)
  /* Function: mdlDerivatives
=============================================
   * Abstract:
   *    In this function, you compute the S-function block's derivatives.
   *    The derivatives are placed in the derivative vector, ssGetdX(S).
   */
  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */



/* Function: mdlTerminate
=====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    if (ssGetPWork(S) != NULL) {
        CPhidgetEncoderHandle *encoder;
        /* Get Phidgets handle */
        encoder = (CPhidgetEncoderHandle *) ssGetPWorkValue(S,0);

        /* Close controller */
        printf("Closing...\n");
        CPhidget_close((CPhidgetHandle)encoder);
        CPhidget_delete((CPhidgetHandle)encoder);

        ssSetPWorkValue(S,0,NULL);
    }

}


/*====================================================*
 * See sfuntmpl_doc.c for the optional S-function methods *
 *====================================================*/
```

```
/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

B.3 PHIDGET INTERFACE KIT

```
/*
 * sfuntmpl_basic.c: Basic 'C' template for a level 2 S-function.
 *
 *  -------------------------------------------------------------------------
 *  | See matlabroot/simulink/src/sfuntmpl_doc.c for a more detailed template
|
 *  -------------------------------------------------------------------------
 *
 * Copyright 1990-2002 The MathWorks, Inc.
 * $Revision: 1.27.4.1 $
 */



/*
 * You must specify the S_FUNCTION_NAME as the name of your S-function
 * (i.e. replace sfuntmpl_basic with the name of your S-function).
 */

#define S_FUNCTION_NAME  phidget_interface_kit
#define S_FUNCTION_LEVEL 2

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */
#include "simstruc.h"
#include <stdio.h>
#include <phidget21.h>


/*====================*
 * S-function methods *
 *====================*/


/* Function: mdlInitializeSizes
=============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl_doc.c for more details on the macros below */

    ssSetNumSFcnParams(S, 2);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }
```

```
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    /* Define 1 input port */
    if (!ssSetNumInputPorts(S, 1)) return;
    /* Input 8-D vector (8 channel) */
    ssSetInputPortWidth(S, 0, 8);
    ssSetInputPortRequiredContiguous(S, 0, true);  /*direct input signal
access*/
    /*
     * Set direct feedthrough flag (1=yes, 0=no).
     * A port has direct feedthrough if the input is used in either
     * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
     * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
     */
    /* ssSetInputPortDirectFeedThrough(S, 0, 1); */

    /* Define 2 output ports */
    if (!ssSetNumOutputPorts(S, 2)) return;
    /* 8-D output ports */
    ssSetOutputPortWidth(S, 0, 8);
    ssSetOutputPortWidth(S, 1, 8);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    /* Pointer to hold Phidgets handle */
    ssSetNumPWork(S, 1);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}


/* Function: mdlInitializeSampleTimes
=========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /* Set sample time based on input parameter */
    double *dSampleTime;
    dSampleTime = mxGetPr(ssGetSFcnParam(S,1));

    ssSetSampleTime(S, 0, *dSampleTime);
    ssSetOffsetTime(S, 0, 0.0);
}



#undef MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
```

```
#if defined(MDL_INITIALIZE_CONDITIONS)
  /* Function: mdlInitializeConditions
========================================
   * Abstract:
   *    In this function, you should initialize the continuous and discrete
   *    states for your S-function block.  The initial states are placed
   *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
   *    You can also perform any other initialization activities that your
   *    S-function may require. Note, this routine will be called at the
   *    start of simulation and if it is present in an enabled subsystem
   *    configured to reset states, it will be call when the enabled
subsystem
   *    restarts execution to reset the states.
   */
  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */


#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart
=======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
    int result;
    int serialNumber;
    const char *err;
    double *dSerialNumber;
    void **PWork = ssGetPWork(S);

    /* Get serial number from parameters (if defined) */
    dSerialNumber = mxGetPr(ssGetSFcnParam(S,0));

    /* Declare an interface kit handle */
    CPhidgetInterfaceKitHandle interfaceKit = 0;

    /* Create the interface kit object */
    CPhidgetInterfaceKit_create(&interfaceKit);

    /* Open the interface kit */
    CPhidget_open((CPhidgetHandle)interfaceKit, (int)(*dSerialNumber));

    printf("Waiting for interface kit to be attached....\n");
    if((result = CPhidget_waitForAttachment((CPhidgetHandle)interfaceKit,
2000)))
    {
        CPhidget_getErrorDescription(result, &err);
        printf("Problem waiting for attachment: %s\n", err);
```

```
    }

    /* Display serial number */
    CPhidget_getSerialNumber((CPhidgetHandle)interfaceKit, &serialNumber);
        printf("Serial Number: %d\n", serialNumber);

    /* Pass along Phidgets handle */
    PWork[0] = interfaceKit;
}
#endif /*  MDL_START */


/* Function: mdlOutputs
=======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    int value, i;
    double analogInputs[8];
    double digitalInputs[8];
    real_T *y = ssGetOutputPortSignal(S,0);
    const real_T *u = (const real_T*) ssGetInputPortSignal(S,0);

    if (ssGetPWork(S) != NULL) {
        CPhidgetInterfaceKitHandle *interfaceKit;
        /* Get Phidgets handle */
        interfaceKit = (CPhidgetInterfaceKitHandle *) ssGetPWorkValue(S,0);
        for (i = 0; i < 8; i++)
        {
            /* Get analog input value for current channel */

CPhidgetInterfaceKit_getSensorValue((CPhidgetInterfaceKitHandle)interfaceKit,
i, &value);
            analogInputs[i] = (double)value;
            /* Get digital input value for current channel */

CPhidgetInterfaceKit_getInputState((CPhidgetInterfaceKitHandle)interfaceKit,
i, &value);
            digitalInputs[i] = (double)value;
            /* Set digital output state for current channel */

CPhidgetInterfaceKit_setOutputState((CPhidgetInterfaceKitHandle)interfaceKit,
i, (int)u[i]);
        }
    }

    /* Update block outputs */
    for (i = 0; i < 8; i++)
    {
        y[i] = analogInputs[i];
        y[i+8] = digitalInputs[i];
    }
}
```

```c
#undef MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate
=====================================================
   * Abstract:
   *    This function is called once for every major integration time step.
   *    Discrete states are typically updated here, but this function is
useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */


#undef MDL_DERIVATIVES  /* Change to #undef to remove function */
#if defined(MDL_DERIVATIVES)
  /* Function: mdlDerivatives
=================================================
   * Abstract:
   *    In this function, you compute the S-function block's derivatives.
   *    The derivatives are placed in the derivative vector, ssGetdX(S).
   */
  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */


/* Function: mdlTerminate
=====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    if (ssGetPWork(S) != NULL) {
        CPhidgetInterfaceKitHandle *interfaceKit;
        /* Get Phidgets handle */
        interfaceKit = (CPhidgetInterfaceKitHandle *) ssGetPWorkValue(S,0);

        /* Close device */
        printf("Closing...\n");
        CPhidget_close((CPhidgetHandle)interfaceKit);
        CPhidget_delete((CPhidgetHandle)interfaceKit);

        ssSetPWorkValue(S,0,NULL);
    }
```

```
}


/*=============================================================*
 * See sfuntmpl_doc.c for the optional S-function methods *
 *=============================================================*/

/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

B.4 SAVEDATA

```
/*  Originally created by WVU Flight Control Systems Group */
/*  Store Data at the end of simulation */

#define S_FUNCTION_NAME savedata
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include "stdio.h"     /* for file handling */

/* mdlCheckParameters, check parameters, this routine is called later from
mdlInitializeSizes */
#define MDL_CHECK_PARAMETERS
static void mdlCheckParameters(SimStruct *S)
{
    /* Basic check : All parameters must be real positive vectors
*/
    real_T *pr;
    int_T  i, el, nEls;

    for (i = 0; i < 3; i++) {
        if (mxIsEmpty(    ssGetSFcnParam(S,i)) || mxIsSparse(
ssGetSFcnParam(S,i)) ||
            mxIsComplex(  ssGetSFcnParam(S,i)) || !mxIsNumeric(
ssGetSFcnParam(S,i))  )
                  { ssSetErrorStatus(S,"Parameters must be real finite
vectors"); return; }
        pr   = mxGetPr(ssGetSFcnParam(S,i));
        nEls = mxGetNumberOfElements(ssGetSFcnParam(S,i));
        for (el = 0; el < nEls; el++) {
            if (!mxIsFinite(pr[el]))
                  { ssSetErrorStatus(S,"Parameters must be real finite
vectors"); return; }
        }
    }

    /* Check number of elements in number of channels parameter */
    if ( mxGetNumberOfElements(ssGetSFcnParam(S,0)) != 1 )
    { ssSetErrorStatus(S,"The parameter must be a scalar"); return; }

    /* get the basic parameters and check them */
    pr = mxGetPr(ssGetSFcnParam(S,0));
    if ( (pr[0] < 1) )
    { ssSetErrorStatus(S,"The number of channels must be greater than zero");
return; }


    /* Check number of elements in [T_start T_end] parameter */
    if ( mxGetNumberOfElements(ssGetSFcnParam(S,1)) != 2 )
    { ssSetErrorStatus(S,"[T_start T_end] must be a two element vector");
return; }
```

```
    /* get the basic parameters and check them */
    pr = mxGetPr(ssGetSFcnParam(S,1));
    if ( (pr[0] < 0) | (pr[1] < 0))
    { ssSetErrorStatus(S,"[T_start T_end] must be non negative"); return; }
    if (pr[0] >= pr[1])
    { ssSetErrorStatus(S,"T_end must be greater than T_start"); return; }

    /* Check number of elements in sampling time parameter */
    if ( mxGetNumberOfElements(ssGetSFcnParam(S,2)) != 1 )
    { ssSetErrorStatus(S,"The parameter must be a scalar"); return; }

    /* get the sampling time and check it */
    pr = mxGetPr(ssGetSFcnParam(S,2));
    if ( (pr[0] <= 0) )
    { ssSetErrorStatus(S,"The sampling time must be greater than zero");
return; }

    /* get the input type and check it */
    pr = mxGetPr(ssGetSFcnParam(S,5));
    if ( ((int)(pr[0]-0.7) < 0) || ((int)(pr[0]-0.7) > 8) )
    { ssSetErrorStatus(S,"The output type must be an integer between 0 and
8"); return; }

}


/* mdlInitializeSizes - initialize the sizes array
*********************************************/
static void mdlInitializeSizes(SimStruct *S) {

    real_T *NumChannels, *SampTime, *SimTime;

    int Ti =(int) (*mxGetPr(ssGetSFcnParam(S,5))-0.7);    /* get the input
type                    */

    ssSetNumSFcnParams(S,6);                          /* number of expected
parameters        */

    /* Check the number of parameters and then calls mdlCheckParameters to
see if they are ok */
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S))
    {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) return;
    }
    else return;

    ssSetNumContStates(S,0);                          /* number of continuous
states          */
    ssSetNumDiscStates(S,0);                          /* number of discrete
states            */

    NumChannels = mxGetPr(ssGetSFcnParam(S,0));       /* Access the number of
channels parameter */
```

```
    SimTime     = mxGetPr(ssGetSFcnParam(S,1));       /* Access the start and
end time of the simulation */
    SampTime    = mxGetPr(ssGetSFcnParam(S,2));       /* Access the sampling
time parameter   */


    if (!ssSetNumInputPorts(S,1)) return;             /* number of input
ports                */
    ssSetInputPortWidth(S,0,*NumChannels);            /* first input port
width               */
    ssSetInputPortDirectFeedThrough(S,0,1);           /* first port direct
feedthrough flag   */

    /* set the input data type, basically equivalent to
ssSetInputPortDataType(S,0,Ti);       */
    switch (Ti) {
    case 0:
        ssSetInputPortDataType(S,0,SS_DOUBLE);
        break;
    case 1:
        ssSetInputPortDataType(S,0,SS_SINGLE);
        break;
    case 2:
        ssSetInputPortDataType(S,0,SS_INT8);
        break;
    case 3:
        ssSetInputPortDataType(S,0,SS_UINT8);
        break;
    case 4:
        ssSetInputPortDataType(S,0,SS_INT16);
        break;
    case 5:
        ssSetInputPortDataType(S,0,SS_UINT16);
        break;
    case 6:
        ssSetInputPortDataType(S,0,SS_INT32);
        break;
    case 7:
        ssSetInputPortDataType(S,0,SS_UINT32);
        break;
    case 8:
        ssSetInputPortDataType(S,0,SS_BOOLEAN);
        break;
    default:
        ssSetErrorStatus(S,"Error in mdlInitializeSizes : input port type
unrecognized");
        return;
    }

    if (!ssSetNumOutputPorts(S,0)) return;            /* number of output
ports               */

    ssSetNumSampleTimes(S,0);                         /* number of sample
times               */
```

```
    ssSetNumRWork(S,0);                                 /* number work vector
elements          */
    ssSetNumIWork(S,0);                                 /* number int_T work
vector elements    */
    ssSetNumPWork(S,1);                                 /* number ptr work
vector elements      */
    ssSetNumModes(S,0);                                 /* number mode work
vector elements     */
    ssSetNumNonsampledZCs(S,0);                         /* number of nonsampled
zero crossing   */
}

/* mdlInitializeSampleTimes - initialize the sample times array
*****************************/
static void mdlInitializeSampleTimes(SimStruct *S) {

    real_T *pr = mxGetPr(ssGetSFcnParam(S,2));

    ssSetSampleTime(S, 0, *pr);
    ssSetOffsetTime(S, 0, 0);
}

/* mdlStart - initialize hardware
************************************************************/
#define MDL_START
static void mdlStart(SimStruct *S) {

    real_T *NumChannels, *SampTime, *SimTime;
    int_T i, Si, NumberOfElements;
    unsigned char *Buffer;

    /* input port data type */
    DTypeId Ti=ssGetInputPortDataType(S,0);

    /* retrieve pointer to pointers work vector */
    void **PWork = ssGetPWork(S);

    /* calculate input datatype size in bytes */
    switch (Ti) {
    case SS_DOUBLE:
        Si=sizeof(real_T);
        break;
    case SS_SINGLE:
        Si=sizeof(float);
        break;
    case SS_INT8:
        Si=sizeof(int8_T);
        break;
    case SS_UINT8:
        Si=sizeof(uint8_T);
        break;
    case SS_INT16:
        Si=sizeof(int16_T);
        break;
    case SS_UINT16:
```

```
        Si=sizeof(uint16_T);
        break;
    case SS_INT32:
        Si=sizeof(int32_T);
        break;
    case SS_UINT32:
        Si=sizeof(uint32_T);
        break;
    case SS_BOOLEAN:
        Si=sizeof(boolean_T);
        break;
    default:
        ssSetErrorStatus(S,"Error in mdlStart : input port type
unrecognized");
        return;
    }

    NumChannels = mxGetPr(ssGetSFcnParam(S,0));       /* Get the number of
channels */
    SimTime     = mxGetPr(ssGetSFcnParam(S,1));       /* Get the start and
end time of the simulation */
    SampTime    = mxGetPr(ssGetSFcnParam(S,2));       /* Get the sampling
time */

    NumberOfElements = ((int_T) ((SimTime[1]-
SimTime[0])/(*SampTime)+1.5))*((int_T) *NumChannels);

    /* allocate memory for the buffer, in bytes */
    Buffer = malloc(Si*NumberOfElements);

    /* store pointers in PWork so they can be accessed later */
    PWork[0] = (void*) Buffer;

    /* check if memory allocation was ok */
    if (PWork[0]==NULL)
        { ssSetErrorStatus(S,"Error in mdlStart : could not allocate
memory"); return; }

    /* zero out memory before starting */
    for(i=0;i<Si*NumberOfElements;i++)
        Buffer[i]=0;

}

/* mdlOutputs - compute the outputs
*********************************************************/
static void mdlOutputs(SimStruct *S, int_T tid) {

    int_T i=0,j=0,n;
    real_T *NumChannels, *SampTime, *SimTime;
    real_T ctime = ssGetT(S);
    real_T *Buffer;

    /* input ports */
    InputPtrsType u = ssGetInputPortSignalPtrs(S,0);
```

```
    real_T **ud;
    float **us;
    int8_T **u8;
    uint8_T **uu8;
    int16_T **u16;
    uint16_T **uu16;
    int32_T **u32;
    uint32_T **uu32;
    boolean_T **ub;

    /* pointers to the same buffer */
    real_T *Bd;
    float *Bs;
    int8_T *B8;
    uint8_T *Bu8;
    int16_T *B16;
    uint16_T *Bu16;
    int32_T *B32;
    uint32_T *Bu32;
    boolean_T *Bb;

    /* input port data type */
    DTypeId Ti=ssGetInputPortDataType(S,0);

    /* retrieve pointer to pointers work vector */
    void **PWork = ssGetPWork(S);

    NumChannels = mxGetPr(ssGetSFcnParam(S,0));       /* Get the number of
channels */
    SimTime = mxGetPr(ssGetSFcnParam(S,1));           /* Get the simulation
start & stop times  */
    SampTime    = mxGetPr(ssGetSFcnParam(S,2));       /* Get the sampling
time */

    n = (int_T) *NumChannels;

    if ( (ctime >= SimTime[0]) && (ctime <= SimTime[1]) ) {
        i = (int_T) ((ctime-SimTime[0])/(*SampTime)+0.5);

        switch (Ti) {
            case SS_DOUBLE:
                Bd = (real_T*) PWork[0];
                ud = (real_T**) u;
                for(j=0;j<n;j++)
                    Bd[n*i+j]=(*ud[j]);
                break;
            case SS_SINGLE:
                Bs = (float*) PWork[0];
                us = (float**) u;
                for(j=0;j<n;j++)
                    Bs[n*i+j]=(*us[j]);
                break;
            case SS_INT8:
                B8 = (int8_T*) PWork[0];
                u8 = (int8_T**) u;
```

```
                for(j=0;j<n;j++)
                    B8[n*i+j]=(*u8[j]);
                break;
            case SS_UINT8:
                Bu8 = (uint8_T*) PWork[0];
                uu8 = (uint8_T**) u;
                for(j=0;j<n;j++)
                    Bu8[n*i+j]=(*uu8[j]);
                break;
            case SS_INT16:
                B16 = (int16_T*) PWork[0];
                u16 = (int16_T**) u;
                for(j=0;j<n;j++)
                    B16[n*i+j]=(*u16[j]);
                break;
            case SS_UINT16:
                Bu16 = (uint16_T*) PWork[0];
                uu16 = (uint16_T**) u;
                for(j=0;j<n;j++)
                    Bu16[n*i+j]=(*uu16[j]);
                break;
            case SS_INT32:
                B32 = (int32_T*) PWork[0];
                u32 = (int32_T**) u;
                for(j=0;j<n;j++)
                    B32[n*i+j]=(*u32[j]);
                break;
            case SS_UINT32:
                Bu32 = (uint32_T*) PWork[0];
                uu32 = (uint32_T**) u;
                for(j=0;j<n;j++)
                    Bu32[n*i+j]=(*uu32[j]);
                break;
            case SS_BOOLEAN:
                Bb = (boolean_T*) PWork[0];
                ub = (boolean_T**) u;
                for(j=0;j<n;j++)
                    Bb[n*i+j]=(*ub[j]);
                break;
            default:
                ssSetErrorStatus(S,"Error in mdlOutput : input port type
unrecognized");
                return;
        }

    }

}

/* mdlTerminate - called when the simulation is terminated */
static void mdlTerminate(SimStruct *S) {

    real_T *NumChannels, *SampTime, *SimTime;
    int_T i, j, NumberOfDataPoints, n;
    FILE *data_file; /* pointer to file */
    unsigned char FileName[16], FpOutStr[16];
```

```
    /* retrieve pointer to pointers work vector */
    void **PWork = ssGetPWork(S);

    /* input port data type */
    DTypeId Ti=ssGetInputPortDataType(S,0);

    /* pointers to the same buffer */
    real_T *Bd;
    float *Bs;
    int8_T *B8;
    uint8_T *Bu8;
    int16_T *B16;
    uint16_T *Bu16;
    int32_T *B32;
    uint32_T *Bu32;
    boolean_T *Bb;

    /* get the file name */
    mxGetString(ssGetSFcnParam(S,3),FileName,16);
    mxGetString(ssGetSFcnParam(S,4),FpOutStr,16);

    NumChannels = mxGetPr(ssGetSFcnParam(S,0));       /* Access the number of
channels parameter */
    SimTime     = mxGetPr(ssGetSFcnParam(S,1));       /* Access the start and
end time of the simulation */
    SampTime    = mxGetPr(ssGetSFcnParam(S,2));       /* Access the sampling
time parameter   */
    NumberOfDataPoints = (int_T) ((SimTime[1]-SimTime[0])/(*SampTime)+ 1.5);

    n = (int_T) *NumChannels;

    /* open data file */
    data_file = fopen(FileName,"w");

    /* check out if it has actually been opened */
    if (data_file==NULL) { ssSetErrorStatus(S,"Could not open the data
file"); return; }

    switch (Ti) {
    case SS_DOUBLE:

        /* retrieve Buffer Pointer */
        Bd = (real_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bd[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
```

```
        fclose(data_file);

        /* free memory */
        free(Bd);

        break;

    case SS_SINGLE:

        /* retrieve Buffer Pointer */
        Bs = (float*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bs[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(Bs);

        break;

    case SS_INT8:

        /* retrieve Buffer Pointer */
        B8 = (int8_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,B8[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(B8);

        break;

    case SS_UINT8:

        /* retrieve Buffer Pointer */
        Bu8 = (uint8_T*) PWork[0];
```

```
        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bu8[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(Bu8);

        break;

    case SS_INT16:

        /* retrieve Buffer Pointer */
        B16 = (int16_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,B16[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(B16);

        break;

    case SS_UINT16:

        /* retrieve Buffer Pointer */
        Bu16 = (uint16_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bu16[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);
```

```
        /* free memory */
        free(Bu16);

        break;

    case SS_INT32:

        /* retrieve Buffer Pointer */
        B32 = (int32_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,B32[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(B32);

        break;

    case SS_UINT32:

        /* retrieve Buffer Pointer */
        Bu32 = (uint32_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bu32[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(Bu32);

        break;

    case SS_BOOLEAN:

        /* retrieve Buffer Pointer */
        Bb = (boolean_T*) PWork[0];

        /* Write the data into the file */
        for (i=0; i < NumberOfDataPoints; i++) {
```

```
            for (j=0; j < n; j++) {
                fprintf(data_file,FpOutStr,Bb[n*i+j]);
            }
            fprintf(data_file,"\n");
        }

        /* close file */
        fclose(data_file);

        /* free memory */
        free(Bb);

        break;

    default:
        ssSetErrorStatus(S,"Error in mdlTerminate : input port type
unrecognized");
    return;

    }

}

/* Trailer information to set everything up for simulink usage
*****************************/
#ifdef  MATLAB_MEX_FILE                    /* Is this file being compiled
as a MEX-file?   */
#include "simulink.c"                      /* MEX-file interface mechanism
*/
#else
#include "cg_sfun.h"                       /* Code generation registration
function        */
#endif
```