DESIGN AND DEVELOPMENT OF A NOVEL OMNI-DIRECTIONAL PLATFORM

by

Steven Bemis

Bachelor of Engineering, University of Ontario Institute of Technology, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Engineering and Applied Science

Mechanical Engineering

Supervisor(s): Scott Nokleby, Faculty of Engineering and Applied Science
Examiner Board: Remon Pop-Iliev, Faculty of Engineering and Applied Science
External Examiner: Ed Waller, Faculty of Energy Systems and Nuclear Science

THE UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

August 2009

©Steven Bemis 2009
Abstract

This thesis presents the design and development of a unique omni-directional platform known as the Omnibot which was built in the Mechatronic and Robotic Systems Laboratory at UOIT. The Omnibot’s layout is novel because its drive axes do not intersect with the geometric center of the body, which is typical for omni-directional platforms using segmented omni-directional wheels. This design enables the center of mass to be lower in the design and increases the stability. A suspension system was designed for each of the four wheels to limit vibrations and to ensure contact between the wheels and operating surface. The Omnibot was built to modularly support many systems, including a robot arm, without altering the mechanical design of the frame. Two control modes were developed: local and global. Commands to drive the Omnibot can be received from either a joystick that can be directly interfaced with the controller or with commands that are sent from other systems that are either on or off of the Omnibot. Both control modes require encoder feedback to ensure commanded velocities are being executed as specified. Global control requires feedback from an indoor localization system to determine the Omnibot’s pose. Early implementation of the localization system is discussed. An open source robotics software, known as Robot Operating System (ROS) was selected for implementation of the Omnibot systems. ROS serves as a middleware which allows components, such as the localization system and remote desktop, to communicate with each other through a decoupled messaging system. ROS is modular and flexible, allowing for easy adaptation of future components. Test results of the Omnibot in operation are presented.
Dedication

To my beloved, Victoria, and to my family.
Acknowledgements

My heart-felt thanks goes to my advisor Dr. Scott Nokleby. He provided technical guidance, major support, and grants towards the completion of this thesis.

I would like to acknowledge the contributions of Mr. Sasha Ginzberg and Mr. Florentin von Frankenberg in the joint development of several key sections of the prototype. I thank Mr. Brian Riess for his assistance on various parts of my prototype as well.

I’d also like to acknowledge Mr. Curtis Agar for his work on the preliminary development of the Omnibot.

I would also like to acknowledge the financial support of BRIC Engineered Systems, the Ontario Centers of Excellence (OCE), and the Natural Sciences and Engineering Research Council (NSERC) of Canada.
# Table of Contents

Abstract ii  
Dedication iii  
Acknowledgements iv  
Table of Contents x  
List of Tables xi  
List of Figures xiv  
List of Abbreviations xv  
Nomenclature xvii  

1 Introduction 1  
1.1 Introduction ................................................. 1  
1.1.1 Serial Robots ............................................. 2  
1.1.2 Parallel Robots .......................................... 2  
1.1.3 Mobile Robots ............................................ 3  
1.1.4 Mobile-Manipulators .................................... 4  
1.2 Thesis Objectives ............................................ 5  
1.3 Summary of Contents ....................................... 5
2 Background and Literature Survey

2.1 Designing Mobile-Manipulators ........................................ 7
2.2 Need for a Test-Bed ....................................................... 8
2.3 Omni-Directional Vehicles .............................................. 9
   2.3.1 Omni-Wheels .......................................................... 10
   2.3.2 Platform Configurations .......................................... 12
   2.3.3 Control of Omni-Directional Vehicles .......................... 20
2.4 Autonomous Systems for Mobile Robots .............................. 22
2.5 Introduction to Open Source Robotic Software ...................... 25
2.6 Existing Open Source Robotic Software .............................. 27
   2.6.1 OROCOS ............................................................... 27
   2.6.2 ORCA ................................................................. 28
   2.6.3 Player ................................................................. 28
   2.6.4 CLARAty ............................................................. 29
   2.6.5 Microsoft Robotics Studio ....................................... 29
   2.6.6 URBI ................................................................. 30
   2.6.7 iRobot AWARE ..................................................... 30
   2.6.8 Skilligent .......................................................... 31
   2.6.9 Willow Garage’s ROS ............................................. 31
   2.6.10 Others ............................................................... 32

3 Mechanical Design .......................................................... 33

3.1 Overview ........................................................................... 33
3.2 Need Statement .............................................................. 33
3.3 Problem Statement .......................................................... 34
3.4 Functional Requirements .................................................. 34
   3.4.1 Constraints ............................................................ 35
   3.4.2 Assumptions .......................................................... 36
4 Kinematics, Control, and Electrical Systems

4.1 Overview ................................................. 60
4.2 Omnibot Kinematics ..................................... 60
4.3 Control Modes ........................................... 63
  4.3.1 Local Control ........................................ 64
  4.3.2 Global Control ....................................... 65
  4.3.3 Local Control with Directional Feedback .......... 66
4.4 Electrical Systems ...................................... 67
  4.4.1 Base Drive System ................................. 67
  4.4.2 Power Source ....................................... 67
  4.4.3 Circuitry ............................................ 69
  4.4.4 Amplifiers .......................................... 70
4.4.5 Encoders .................................................. 73
4.4.6 Encoder Counters ..................................... 74
4.4.7 Direct User Interface ................................ 75
4.4.8 Safety Components .................................. 76
4.4.9 Microcontroller .................................... 77
4.5 Velocity Control ........................................ 79
4.6 Indoor Global Localization System .................... 82
4.7 Summary ................................................. 85

5 Software and Programming 86
5.1 Overview .................................................. 86
5.2 Microcontroller Programming .......................... 86
5.3 Omnibot Software .................................... 87
5.4 Remote Desktop and On-board Data Processor .......... 88
5.5 Review and Comparison of Open Source Platforms .... 89
5.5.1 Requirements for Open Source Robotics Software ... 89
5.5.2 Software Evaluation ................................. 91
5.6 OROCOS Implementation ............................... 92
5.6.1 Problems Encountered ............................... 93
5.7 ROS Implementation .................................... 94
5.8 Summary ................................................. 98

6 Test Results and Discussion 100
6.1 Overview .................................................. 100
6.2 Mechanical Test Results ................................ 100
6.3 Velocity Controller Test Results ........................ 104
6.3.1 Encoder Tests ....................................... 104
6.3.2 PID Tuning .......................................... 105
List of Tables

3.1 Table describing robot arm fastening requirements
3.2 Estimated total mass of the Omnibot
6.1 Spring compression test
6.2 Encoder counter test
List of Figures

1.1 Epson Pro Six PS3 robot arm [1] ................................. 2
1.3 PowerBot [4] ......................................................... 3

2.1 The Jasper Mobile-Manipulator [9] ............................... 9
2.2 Three main types of omni-wheels [12] ............................ 11
2.3 Mecanum wheel and profile of motion [14] ......................... 11
2.4 Three-wheeled ODV [15] .......................................... 12
2.5 Four-wheeled ODV with segmented omni-directional wheels [17] . 13
2.6 Tank-like ODV [18] .................................................... 13
2.7 Wheelchair using two omni-wheels [20] ............................ 14
2.8 Sketch of ODV using caster wheels [21] ........................... 15
2.9 Four-wheeled ODV with differential gearing [23] .................. 16
2.10 Vuton II [24] .......................................................... 16
2.11 Azimut: Modular mobile robotic platform with legs-tracks-wheels [25] 17
2.12 Mecanum wheel with rotatable rollers [27] ......................... 18
2.13 Airtrax’s Sidewinder industrial lift truck [28] ...................... 18
2.14 Segway RMP 400 Omni [29] ..................................... 19
2.15 Hammonds ODV tractor [30] ...................................... 20
2.16 AGV transporting material [37] ................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.17</td>
<td>Sketch of AGV using a wire-guided system [3]</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Functional decomposition of the ODV</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>ODV presented by Agar [59]</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Constructed prototype Omnibot</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>CAD rendered views of the Omnibot</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>CAD rendered image of the frame of the Omnibot</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Various views of the Omnibot’s frame composed of 80/20 profiles</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Placement of the inner frame member on the Omnibot</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Moment diagram for inner structure frame member</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>MicroMo motor mounted on the Omnibot</td>
<td>49</td>
</tr>
<tr>
<td>3.10</td>
<td>Custom designed motor interface plate</td>
<td>51</td>
</tr>
<tr>
<td>3.11</td>
<td>Motor interface plate and L-shaped brackets</td>
<td>52</td>
</tr>
<tr>
<td>3.12</td>
<td>Guide plate on the Omnibot</td>
<td>53</td>
</tr>
<tr>
<td>3.13</td>
<td>Omnibot suspension</td>
<td>54</td>
</tr>
<tr>
<td>3.14</td>
<td>Exploded view of the Omnibot suspension with labels</td>
<td>54</td>
</tr>
<tr>
<td>3.15</td>
<td>Double universal joint on the Omnibot</td>
<td>56</td>
</tr>
<tr>
<td>3.16</td>
<td>Exploded view of transmission with labels</td>
<td>57</td>
</tr>
<tr>
<td>3.17</td>
<td>Electronics section on the Omnibot</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic schematic of the Omnibot</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Omnibot showing different motions</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Simplified flowchart of systems used in global control of the Omnibot</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Flowchart showing drive system components on the Omnibot</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>PCB mounted on the Omnibot</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Image used in making PCB during photoengraving process</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Duty cycle versus velocity for the Victor 884 amplifiers</td>
<td>72</td>
</tr>
<tr>
<td>4.8</td>
<td>Graph showing quadrature encoder transactions [65] (see Appendix A)</td>
<td>73</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGV</td>
<td>Automated Guided Vehicle</td>
</tr>
<tr>
<td>BFL</td>
<td>Bayesian Filtering Library</td>
</tr>
<tr>
<td>DOF</td>
<td>Degrees-of-Freedom</td>
</tr>
<tr>
<td>E-Stop</td>
<td>Emergency Stop</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ICE</td>
<td>Internet Communications Engine</td>
</tr>
<tr>
<td>ICs</td>
<td>Integrated Circuits</td>
</tr>
<tr>
<td>KDL</td>
<td>Kinematics and Dynamics Library</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LED</td>
<td>Light-Emitting Diode</td>
</tr>
<tr>
<td>MARS</td>
<td>Mechatronic and Robotic Systems</td>
</tr>
<tr>
<td>OCL</td>
<td>Orocos Components Library</td>
</tr>
<tr>
<td>ODV</td>
<td>Omni-Directional Vehicle</td>
</tr>
<tr>
<td>ORCA</td>
<td>Open Robot Control Architecture</td>
</tr>
<tr>
<td>ORocos</td>
<td>Open Robot Control Software</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional-Integral-Derivative</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>ROVs</td>
<td>Remotely Operated Vehicles</td>
</tr>
<tr>
<td>RTT</td>
<td>Real-Time Toolkit</td>
</tr>
<tr>
<td>URBI</td>
<td>Universal Real-Time Behavior Interface</td>
</tr>
<tr>
<td>US</td>
<td>Ultrasonic Signals</td>
</tr>
</tbody>
</table>
Nomenclature

\( \theta \) Angle of rotation of the platform
\( a \) Distance between load and support
\( C \) Circumference of wheel
\( d \) Distance from the neutral axis to the edge of the beam
\( d_{ij} \) Specified distance between nodes \( i \) and \( j \)
\( d_n \) Directional unit vector for wheel \( n \)
\( E \) Efficiency of the gearhead
\( e \) Error
\( f \) Friction coefficient
\( GR \) Gear ratio
\( I \) Moment of inertia
\( k \) Spring coefficient
\( K_D \) Derivative gain constant
\( K_I \) Integral gain constant
\( K_P \) Proportionality gain constant
\( K_{Pc} \) Critical value of the proportional gain constant
\( l \) Fixed distance between geometric center and center of the omni-wheel
\( l_f \) Free length of spring
\( l_i \) Installation length of spring
\( l_o \) Operating length of spring
\[ n_{rot} \quad \text{Number of pulses per rotation of the motor} \]

\[ p_{on} \quad \text{Position of wheel } n \text{ relative to the geometric center} \]

\[ r \quad \text{Radius of wheel} \]

\[ Rot \quad \text{Number of counted rotations} \]

\[ t \quad \text{Total time of experiment} \]

\[ T_c \quad \text{Periodic time of oscillations} \]

\[ T_D \quad \text{Derivative time constant} \]

\[ T_I \quad \text{Integral time constant} \]

\[ v_d \quad \text{Desired velocity} \]

\[ v_n \quad \text{Velocity of wheel } n \]

\[ v_{out} \quad \text{Measured velocity} \]

\[ W \quad \text{Load} \]

\[ W_{total} \quad \text{Total Weight of system} \]

\[ W_{un} \quad \text{Total weight of the Omnibot when not loaded} \]

\[ x_{Ai} \quad \text{X coordinate of the actual node position for node } i \]

\[ x_{Ei} \quad \text{X coordinate of the estimated node position for node } i \]

\[ y_{Ai} \quad \text{Y coordinate of the actual node position for node } i \]

\[ y_{Ei} \quad \text{Y coordinate of the estimated node position for node } i \]
Chapter 1

Introduction

1.1 Introduction

Robots have become commonplace in our world during the last few decades and have served as a reliable solution to many of our problems. How they interact with people day-to-day is constantly changing as further advancements in technology are made. From industry to household work, military operations to precision surgery, it is obvious that robots are an integral part of today’s society. While robots have been used to replace workers in industry, they have also been used as a way to move people away from dangerous environments while still allowing those people to do their jobs. This thesis will explore the development and control of an omni-directional platform, which is a mobile platform that has the ability to drive in all directions without restriction. This platform will be used for autonomous applications as well as for testing of mobile-manipulator applications. In addition, the software that operates this platform will also be examined so that the robot can be easily adapted to many different applications.

To understand what problems are associated with creation and control of robots, it is first necessary to discuss what different types of robots exist. Robots can be either
serial, parallel, mobile, or any combination of the three.

1.1.1 Serial Robots

Serial robots consist of a chain of single degree-of-freedom actuators that move in such a way that the end effector or tool can be moved to a location and orientation in space. Serial robots are commonly used in manufacturing environments where complex, yet repetitive motions are required. An example application would be the spot welding of a car frame. A serial robot can be as seen in Figure 1.1.

1.1.2 Parallel Robots

Parallel robots can be most easily envisioned as several serial robots acting on a common platform. They have an advantage over serial robots because they are typically stronger, more accurate, and more resistant to vibration. However, the motion planning of these devices is much more complex. A common application of a parallel robot are flight simulators such as the one presented in Figure 1.2.
1.1.3 Mobile Robots

Unlike a serial or parallel robot that is bolted to the floor, a mobile robot has an unlimited workspace. However, mobile robots are not manipulators such as robotic arms and lack the ability to manipulate the environment, so they are usually used in transportation tasks. An example of a mobile robot is an automated guided vehicle (AGV) [3]. One such vehicle can be seen in Figure 1.3. Since mobile robots have a much larger workspace than a typical parallel or serial robot, precision tasks are much more difficult to do.
1.1.4 Mobile-Manipulators

Mobile-manipulators are a combination of a mobile robot with either a parallel or serial robot. This combination gives the mobile robot an ability to manipulate objects in the space it can occupy. Mobile-manipulators have been used in space, underwater, and hazardous environments such as nuclear reactors. It is beneficial to use a mobile-manipulator as an extension of a worker’s body, allowing the worker to do a job in a dangerous environment while preventing exposure to hazards. Since mobile-manipulators have a redundant number of degrees-of-freedom (DOF), it becomes more difficult to control them. Examples of mobile-manipulators are the NASA Spirit and Opportunity Mars rovers (see Figure 1.4). There have been several rovers sent to the planet Mars on fact-finding missions. NASA’s robot geologists were sent to Mars to find evidence of water by navigating the terrain and searching with various sensors on-board [5].

Another type of mobile-manipulator is an underwater Remotely Operated Vehicle (ROV). ROVs are used for exploration, extraction, maintenance, and construction. They are also seeing increasing use in military applications.
1.2 Thesis Objectives

The objective of this thesis is to document the design, development, and testing of the experimental omni-directional platform known as the Omnibot. The Omnibot must be able to carry a robot arm and other systems. The Omnibot is planned to be used for testing mobile manipulator and autonomous applications. The implementation of the robotic arm in conjunction with the Omnibot is beyond the scope of this thesis. The focus of this work is the development of the mobile platform. As part of this, a velocity controller must be designed. Preliminary implementation of a global positioning system is also done. It is desired to have all of the sub-systems that will be incorporated on the Omnibot communicate using an open source robotics architecture and show that the Omnibot can be tele-operated remotely.

The problem of developing an omni-directional vehicle and its support systems is solved by designing the core mechatronic elements, namely: the mechanical, the electrical, the control, and the software systems, in a concurrent manner. The requirements for the vehicle are listed clearly, and a design is selected and improved upon. Individual components of the vehicle are selected given the requirements specified. A fully functional working prototype is built. Extensive mechanical, electrical, and software testing is undertaken in order to verify the design and satisfy the objectives of this thesis.

1.3 Summary of Contents

Chapter 2 reviews available designs of omni-directional vehicles as well as control strategies. It also has an extensive literature survey of available open source robotic software.

Chapter 3 discusses the mechanical design of an omni-directional platform known as the Omnibot. The Omnibot is broken up into a number of smaller sub-systems.
including the suspension, motor coupling, motor, frame, and drive system. Each section is discussed in terms of design decisions and development.

Chapter 4 discusses the controller design for the Omnibot. This section first looks at the specific kinematic nature of the Omnibot, then discusses control methods and implementation on the Omnibot. Other systems are also discussed such as methods for potential global control of the Omnibot.

Chapter 5 considers the use of open source software to interface the various subsystems on the Omnibot. The selection of open source software is also discussed.

Chapter 6 presents the experimental results from testing the Omnibot. This includes test results for the mechanical design, velocity controller, indoor localization system, and open source robotics software.

Chapter 7 concludes this thesis with a summary of contributions as well as a discussion on where this research can be continued in the future.
Chapter 2

Background and Literature Survey

2.1 Designing Mobile-Manipulators

Since the environments that mobile-manipulators are built for are generally hazardous for humans, design of mobile-manipulators can be quite difficult. For example, an ROV must be water tight and have the ability to withstand large pressure while still being able to communicate with its operators at a distance. While other mobile-manipulators, such as the NASA Spirit and Opportunity Mars rovers, must be able to operate from a considerable distance while working in an environment devoid of any power supply besides that obtained from solar panels.

The successful control of a mobile-manipulator is also challenging. If tele-operation is used, mobile-manipulators typically must be controlled by several operators at once, one to control the mobile base, and others to control the various manipulators. This makes operation of such equipment quite complex, expensive, and slow, because operators must communicate with each other to determine the best way to tackle a situation.

Another way to tackle this problem is to develop autonomous systems by adding sensors to read in information about the environment. The information can then be
used by the mobile-manipulator to determine how it should move.

Research is being conducted into how to deal with the problem of co-ordinating control between the various manipulators and their perspective bases (see for example [6–8]). In the Mechatronic and Robotic Systems (MARS) Laboratory at the University of Ontario Institute of Technology (UOIT), research is being conducted to simplify the control of redundant mobile-manipulators using knowledge of the kinematic singularities of the manipulator to control the system [9,10]. When this research is complete, it will only be necessary to have one operator for the entire system, making it simpler for the operator to control the whole body and react faster and efficiently to changes in the environment the robot is working in. This would also make mobile-manipulators more practical as a tool for workplace use, making jobs feasible or even safe where it would otherwise be too dangerous.

2.2 Need for a Test-Bed

Research is being conducted to simplify the control of a mobile-manipulator [9,10]. As part of this research, a test-bed was designed. The system, dubbed Jasper, consists of a 6-DOF robotic arm attached to a 2-DOF mobile base as can be seen in Figure 2.1. It is currently being used to develop and test an algorithm that accepts a 6-DOF joystick input to control the robot arm while at the same time moving the base to counter-act singularities that the arm may reach while in motion. A singularity is a configuration where the robot arm instantaneously loses a DOF of motion capability. This means that the robot is not capable of moving in one direction due to its current configuration.

Jasper is only meant for factory-type environments, and is ideal for testing these algorithms before they are applied to move complex systems such as an underwater ROV or tele-operated mining equipment. The test-bed setup has one obvious disadvantage.
Since it is attached to a wheeled base the system is non-holonomic [11]. A vehicle, such as a car, is considered non-holonomic because it has restrictions on where it can travel. A car can go forwards or backwards, and change its angle of direction over a distance. However, a car cannot travel sideways. A holonomic vehicle is one that does not have these restrictions. It is able to travel in any direction at any point.

To extend the work on co-ordinated control of mobile-manipulators, it is desirable to have a holonomic base such as an omni-directional platform.

### 2.3 Omni-Directional Vehicles

The basis of an omni-directional vehicle (ODV) is that it has the ability to travel in any direction while maintaining a certain orientation. In order to do this, a series of contacts must be made with the travel surface that allow more then one travel direction at a time. The most common solution to this is to use omni-directional wheels or omni-wheels. Omni-wheels have the special ability to travel in more then one direction at a time. In contrast, a car tire is only capable of rolling in the direction
2.3.1 Omni-Wheels

Although there are many different types of omni-wheels, their operating premise is generally the same. The point of contact for the wheel has the ability to roll in two different directions simultaneously. If the omni-wheel is being driven, i.e., connected to a motor, the driven axis of rotation would be the primary axis, while the other axis would be allowed to roll freely. The free-rolling direction of the wheel is not parallel to that of the driven direction. To achieve 3-DOF motion in a plane, a minimum of three wheels must be mounted in a configuration that allows the 3-DOF to be controlled by the driven wheels. This means that the vehicle is not only capable of traveling forward and backward, but also horizontally, diagonally, rotating on the spot, or a combination of translating and rotating.

There are three types of omni-directional wheels that are primarily used on omni-directional platforms. They are the segmented omni-directional wheel, the double omni-directional wheel, and the mecanum wheel (or Swedish wheel). Each of these wheels have distinct advantages and disadvantages.

The segmented omni-directional wheel has a series of rollers mounted around the circumference of a larger wheel as shown in Figure 2.2(a). The rollers are mounted so that they will roll perpendicular to that of the primary axis. Since it is impossible to have enough barrels covering the entire circumference, there is spacing between each barrel. This causes vibrations and clicking as the wheel rolls across a surface.

The double omni-directional wheel comprises two interlocked segmented omni-directional wheels as shown in Figure 2.2(b). This wheel solves the problem of clicking as the wheel travels a surface by alternating between each wheel. Therefore, there is always one barrel in contact with the floor. However, since the wheel is always switching between one wheel and the other, the center of force alternates position between the
two wheels as they rotate. This can cause noise and vibration as the wheel moves over uneven surfaces.

The mecanum wheel [13] is similar to that of the segmented omni-directional wheel with the exception that the barrels are mounted 45° instead of perpendicular to that of the circumference of the larger wheel (see Figures 2.2(c) and 2.3). The wheels must always work against each other to achieve the desired motion because of how the barrels are laid out. This causes forces on the frame that connects the wheels together.

There are other unique omni-directional wheels that have been developed; all work on
the principle that each wheel can travel in more than one direction at a time. There are also other platforms that can produce 3-DOF movement using caster wheels that have their direction of travel controlled as well. This is not considered true omni-directional travel because the wheels must be rotated to change the direction of travel. Both true and non-true 3-DOF ODVs will be discussed in the next section.

2.3.2 Platform Configurations

One of the first patented omni-directional vehicles was developed by Smith [15]. His vehicle uses three segmented omni-directional wheels mounted in three different directions from the geometric center of the vehicle as can be seen in Figure 2.4. Many variations of Smith’s design exist that use either three or four segmented omni-directional wheels mounted with their axes intersecting the geometric center of the ODV. Leow et al. [16] show a similar three-wheeled design in [15]. Rojas and Förster [17] show a four-wheeled design (see Figure 2.5). All of these designs have each wheel controlled independently. The direction and velocity of the ODV is governed by the sum of the vectors produced by each motor. Since the axis of each wheel is coincident with the geometric center of the cart, the direction of the body is relatively easy to calculate if the velocity of each wheel is known.
Mark and West [18] patented another type of omni-directional vehicle. The vehicle uses two tracks, similar to that of the treads of a tank. However, unlike a tank, each track is filled with spheres that can be propelled around the track in series. The spheres make contact with the ground and are pushed through the track to move the vehicle forward and backward. The spheres in one or both tracks can also be rotated perpendicular to the direction of the track by means of friction contact with one or more of the spheres in contact with the ground. This friction contact is powered by a separate motor. A sketch of this design can be found in Figure 2.6. The design enables the desired 3-DOF motion of the vehicle [19].
As can be seen in the previous example, not all ODVs require omni-directional wheels. Wada [20] developed another type of ODV that uses two regular wheels and two omni-directional wheels. A CAD model of his design and a sketch of the top-down view can be seen in Figure 2.7. The vehicle uses two motors in the form of 4WD, with the back tire on one side driven by the same motor as the front omni-directional wheel of the same side. It should be noted that this vehicle cannot achieve true omni-directional travel on its own because two of the wheels are not omni-directional.

Omni-directional wheels have been used to replace caster wheels, as caster wheels tend to produce undesired, rough travel when making sharp turns or suddenly reversing directions. Omni-directional wheels have overcome this issue by avoiding the caster design altogether. However, several ODVs have been developed including one by Park [21] which uses three caster wheels partnered with six motors. The researchers state that since omni-directional wheels are sensitive to the conditions of the surface they are traveling on, caster wheels are more ideal for good omni-directional travel. A sketch of the platform can be seen in Figure 2.8. Each caster wheel on the vehicle is controlled by two motors, one to control the wheel’s orientation and the other to drive the wheel itself. Combined with three different caster wheel branches, the vehicle can be controlled much the same as an ODV. However, this design requires the
wheels to be able to slide in the direction perpendicular to the wheel axis of rotation. Another castor ODV by Yu et al. [22] uses active split offset castor wheels. Unlike regular castor wheels, the active split offset castor wheel has two wheels that are powered individually while sharing the same orientation with respect to the passive joint the wheels share. By attaching to a body and actuating several of these wheels, omni-directional motion is achieved.

Three omni-directional wheels are enough to drive the platform in true 3-DOF. However, stability is much better when there are four wheels to rely on. The ODV introduced by Asama et al. [23] is an interesting design that utilizes a four omni-wheel base, but only uses three motors to drive it. This is accomplished with the use of differential gearboxes. Figure 2.9 reveals that each wheel is controlled by two motors, linked through a differential and gear system. The system has been described as a decoupled drive. Each motor is responsible for only one direction of motion and therefore is easier to control. However, as can be seen in Figure 2.9, the mechanical design becomes much more complex.

Another interesting type of ODV is a holonomic ODV with a controlled caster wheel mechanism. This vehicle is quite complex, but its design is essentially the same as any other ODV. The Vuton II, designed by Damoto [24] uses omni-discs to traverse...
a surface. Four omni-discs have been made out of a collection of casters wheels and locking plates. The omni-discs are built to keep each caster wheel facing the same direction while still allowing the entire disk to spin. Each caster wheel is allowed to spin freely. The disc is mounted on the ground at an angle of about 4 degrees so that only one or two of the caster wheels is actually contacting the ground (see Figure 2.10). The disk is then driven so that the caster wheels alternate being on the ground causing a force in the direction perpendicular to that of the caster wheels rolling direction. These four disks driven independently achieve 3-DOF motion. Azimut [25] is a unique ODV that makes use of multiple mechanisms on one platform.
Earlier versions of the platform used a concept that is kinematically similar to that of the caster design mentioned earlier. That is, one wheel and then another driver to control the wheels direction (see Figure 2.11(a)). Later designs changed the shape of the wheel to a track around a unique shape that could also be rotated. These unique shapes serve as legs which can be seen in Figure 2.11(b). This means each drive axis has three motors associated with it, making the robot very versatile and well suited for working in tight areas. A similar version to the early version of Azimut was presented by Mori et al. [26] as well as a discussion for control of such a vehicle. Diegel et al. [27] present another type of ODV that makes use of a unique mecanum wheel design. Each mecanum wheel has the ability to actuate the angle of its passive axis from $45^\circ$ to $-45^\circ$. The passive barrels can also be locked, allowing the mecanum wheel to drive like a car when the second axis is at $0^\circ$. With this functionality, the wheels can avoid traditional problems that occur with mecanum wheels such as the stress always occurring in the frame of the vehicle. A picture of this type of mecanum wheel can be seen in Figure 2.12. The disadvantage with this type of mecanum wheel is that the radius is no longer continuous, and travel becomes bumpy like that of a single segmented omni-directional wheel.
Figure 2.12: Mecanum wheel with rotatable rollers [27]

Figure 2.13: Airtrax’s Sidewinder industrial lift truck [28]
Many of the practical ODV applications make use of the mecanum wheel. One of the most notable vehicles is an omni-directional forklift manufactured by Airtrax [28]. The forklift uses four mecanum wheels with their primary axes mounted just like traditional forklifts. The Airtrax omni-directional drive system boasts its ability to drive in any direction, meeting the tight needs of warehouse space. Conventional forklifts must move back or forward in order to be able to turn, while the Airtrax forklift does not have this issue. A picture of the forklift can be seen in Figure 2.13.

Another design that uses mecanum wheels is the Segway RMP Omni [29]. If a user is on the platform, driven motion can be produced simply by leaning in the direction the user wishes to go. Since mecanum wheels are used, the platform is capable of moving in any direction (see Figure 2.14). This is in contrast to other Segway platforms that use regular wheels that must translate to change the direction of travel.

Hammonds Technical Services [30] manufactures non-holonomic AGV tractors that serve in any number of applications including towing tasks such as plane, people, and luggage towing for airports, as well as snow removal. The ODV works by rotating two independent drive wheels that are coincident with the center of geometry. The tow bar that is around the circumference of the ODV is allowed to stay stationary as the ODV rotates inside. Since the rotation is based on two regular tires, some motions are not possible such as side-to-side motion. These motions can be produced only...
with translation (like a car) or rotating in its own footprint. This means the tractor
can move in any direction with some re-orientating, but is a non-holonomic vehicle.
A picture of one of these tractors is shown in Figure 2.15.
The drive mechanism for the Hammonds AGV is similar to that of TRC LabMate
trucks that make up the OmniMate which is presented by Borenstein and Evans [31].
The Omnimate uses two 2-DOF trucks linked together by a compliant platform which
can handle a fairly large load. Like the Hammonds AGV, this vehicle is not truly
holonomic, but is capable of 3-DOF motion. Having the trucks work in conjunction
with each other allows for any range of motion.

2.3.3 Control of Omni-Directional Vehicles

The control of omni-directional platforms has also been extensively researched. Sev-
eral strategies have been devised such as the one presented in Leow et al. [16]. The
model they devised gives a set of equations that describe the motion of three omni-
directional wheels relative to the motion of the body. The authors also claim that the
same model could be used to describe the control of a four-wheeled platform.
Rojas and Förster [17] discuss a similar algorithm to [16] that also examines a three-
wheeled design, but is written to handle \( n \) wheels when \( n > 3 \). As soon as more
than three wheels are used, the platform becomes redundant, meaning there are more wheels than necessary to achieve the desired motion. Rojas and Förster [17] also discuss slip detection and energy-saving drive techniques.

Leow et al. [16] studied specifically the kinematic control of omni-directional platforms. A relation between the position of the wheels and their unit vectors relative to the center of the body is established to produce equations that show the exact relation between the wheels’ velocities and the overall velocity of the body.

Another control method is seen in Kalmar-Nagy et al. [32]. Similar to [16], control algorithms are developed and a Jacobian matrix is derived for a three-wheeled omni-directional platform. These control algorithms generate near-optimal trajectories for an ODV by taking the second-order dynamics of the vehicle into account.

The path of an omni-directional robot can also be optimized for shortest travel time. Balkcom et al. [33] analyzed a three-wheeled ODV for different types of trajectories and their affect on time of travel. They concluded that there are only four types of optimal trajectories, and a maximum of 18 control switches are needed to move in those trajectories.

Similarly, Purwin and D’Andrea [34] developed a trajectory generation algorithm for a four-wheeled ODV. The algorithm uses the vehicle dynamics, limited friction, and weight transfer to compute optimal trajectories.

Motion planning for ODVs has been addressed by Smid et al. [35]. Their 6-wheeled ODV is controlled by an Intelligent Motion Planning scheme used in obstacle avoidance. A virtual simulation environment was developed to test the algorithms developed.

Borenstein [36] used internal position error correction to assist the OmniMate in dead-reckoning. Due to the OmniMate’s lack of omni-directional wheels and unique configuration, there is much less slippage associated with driving it. He used the variances between the two trucks to compensate for tracking error that might occur
as the OmniMate traveled over uneven terrain. Borenstein’s test results show significant improvement in accuracy with the internal position error calculation that he developed.

2.4 Autonomous Systems for Mobile Robots

Since the ODV to be designed will be autonomous, it is advantageous to review systems that allow mobile robots to be autonomous. Autonomous robots work without human interaction, they are able to do a job automatically. In the case of a mobile robot, it is important to have a system or set of systems capable of giving the mobile robot information about the environment and where it is safe and necessary to drive. The most important system of any autonomous mobile robot is its navigation system. An example of an autonomous robot are mobile robots used in factory environments primarily for material transport. These vehicles are known as Automated Guided Vehicles (AGV). An example of an AGV is one that carries the chassis of a vehicle from station to station in an auto manufacturing plant as various assembly operations are done to it such as the motor being attached (see Figure 2.16). AGVs work like a conveyor, in that they transport the part through sections, both in series and in parallel, and can be easily told to change course during operation. If the material or AGV is damaged or needs special work done to it, the AGV can be simply driven away from the line manually.

An AGV works autonomously through a guidance system. There are several types of systems available to navigate the vehicle around the factory floor. They are a wire-guided, paint strip-guided, self-guided systems.

In wire-guided systems, a wire is embedded in the factory floor where the AGV will travel. The wire serves as the AGVs guide and controller as signals are sent to the AGV as it moves along the wire. The vehicle stays centered on the wire by monitoring
the magnetic field generated by the wire. The distance to the wire is proportional to that of the strength of the magnetic field detected by sensors mounted on-board the AGV. Two sensors are used on the AGV as shown in Figure 2.17. The AGV keeps itself centered over the wire by constantly adjusting its angle of travel so as to keep the reading of the strength of the magnetic field at both sensors the same. The wire is also capable of sending instructions to a specific AGV as it moves down the line to change course, speed, etc. Also, if several wires broke off in parallel from one path, the AGV would need further instruction on where to go. A disadvantage to this system is the expense of initial installation because the floor must be cut into to embed the wire. Also, if re-configuration of the path must be done, the wire must be removed, and new slots must be cut for the wire path.

Paint strip-guided AGVs work on the same principle of the wire-guided system. A line is either painted or taped to the floor for the AGV to follow. A vision system is mounted in the AGV to detect the position of the line with respect to the center of the AGV. As the line moves to one side, the vision system reports to the drive system, and course correction is made to keep the AGV centered over the line. Paint strip-guided systems are easier to install than wire-guided systems, but are harder to maintain. If the factory floor is a high traffic area, the line will frequently need
to be cleaned or repaired so that the camera can continue to see the line without problem. Paint systems also do not have to be powered throughout the factory like the wire-guided system does, which makes the system passive.

Self-guided AGVs use forms of wireless communication and sensing to map their position and path throughout the factory. The self-guided systems described in [3] use a combination of dead-reckoning and beacon placement for navigation. Dead-reckoning is the ability to travel from a known position to another position with only wheel monitoring. Each wheel is told to move at a certain velocity and is monitored for accuracy through the use of encoders. Adjustments are made if a wheel is going too fast or too slow so as to keep the vehicle as close to the original commanded direction as possible. Once the AGV has dead-reckoned for awhile, it would be in proximity of another beacon and be able to adjust itself from any error in direction it accumulated during dead-reckoning. Beacons are placed in strategic positions around the plant, and broadcast their known position to the AGV. As the AGV moves between beacons using dead-reckoning, it updates its map on-board to help navigate to other beacons and back to previously found beacons. Beacons can consist of a variety of systems, including bar-codes placed where an AGV can scan them via rotating laser scanner or they can be magnetic beacons mounted in the floor. Using two beacons, the AGV can triangulate its own position.
The self-guided system presented in [38] uses reflective targets that are scanned by the rotating laser scanner mounted on an AGV. The AGV measures the laser light that is reflected off these reflective targets and from this data the AGV interprets its distance and angle from the targets. The reflective tags store position information as well, therefore, the position of the AGV can be determined from the tag. Using its position and previously stored knowledge of the path, the AGV moves throughout the facility. Such a system is presented in [39].

Another type of self-guided AGV uses gyros to detect very slight changes in direction. Using corrective directional control, the error produced from dead-reckoning is decreased significantly. Again, this system uses beacons to correct the cumulative error that comes from dead-reckoning as it travels through the factory.

An ODV could be used as an AGV as well. Many of the proposed systems above would work for an ODV. In terms of dead-reckoning, there was a system developed in [40] that attaches a passive two-wheeled castor to a ODV to be used as an odometer. Either wheel is allowed to move independently of the other, and the rotation of each wheel and the caster joint is tracked. Using this information, a rather accurate interpretation of position can be tracked.

2.5 Introduction to Open Source Robotic Software

As a general rule, it is best to use readily available hardware and software instead of starting from scratch for new designs. This is important when designing a robot. A mobile-manipulator must have communication not only between the two or more robotic systems involved (e.g., robot arm and mobile base), but also the various sensors and subsystems that support the operation of the entire system. When multiple systems need to communicate, it is best to have a common language or at least, a master system that can speak to all components at any time during operation.
In industry, robots are generally programmed using proprietary algorithms and languages. However, a movement towards producing universal software has begun among the software development community. Among the benefits of producing a “one glove-fits-all” solution is modularity, flexibility, powerful abstraction, and simplicity [41]. Other benefits of open source software can be seen in [42]. No universally accepted robotics software exists as of today because of the vast diversity in robotic applications and the countless proprietary algorithms to make them work. There is, however, a large number of groups that are trying to produce such software. Even companies that previously developed proprietary hardware and software for robots are starting to re-consider this approach to satisfy the growing need in the market for conformance. Therefore, it is important to sort through and compare existing software architectures to determine which is best suited for the application of mobile manipulation and specifically for the proposed omni-directional platform.

In order to discuss what open source robotic software packages are available, one must first understand what is open source software. An open source program does not necessarily mean a universal program. A universal program would be capable of operating on several different types of robotic arms. This simply does not happen. An open source program is a program that is based on known specifications, meaning that the software source code is available to a user and the user may modify the code as needed.

Open source software is usually free software, as in it does not need to be purchased. It is important to note that open source software may not always cost nothing. However, it can be considered ‘free’ in the sense that the user is allowed to do anything with the software once they possess it. The official definition of free software according to [43] is that the user of the software will not have to pay or ask for permission to redistribute, alter, study, or sell the software in any way.

Some open source robotic software is referred to as middleware. Middleware is soft-
ware designed to integrate separate software and/or hardware systems. Middleware provides the communication between the separate systems.

2.6 Existing Open Source Robotic Software

One review of available robotic software can be found in [44]. The discussion here will be more towards open source software while still taking a general look at all software platforms.

2.6.1 OROCOS

Open Robot Control Software (OROCOS) is one of the major contenders for the open source robotics market [45]. It serves as an advanced middleware that allows communication between various components in the form of “Taskcontexts.” Each Taskcontext is decoupled from other Taskcontexts, and serves as the interlocking template for communication between the various components. The advantage of this framework is that each component can be tested individually and other components do not need to be connected with it in order for it work. OROCOS also provides real-time communication between components. OROCOS also offers several useful tools and libraries including the Kinematics and Dynamics Library (KDL), Bayesian Filtering Library (BFL), a library that allows it to interface with MATLAB Simulink, and a variety of sample components. OROCOS only operates on Linux systems and cannot work on Microsoft Windows because Windows’ mandatory background tasks will not allow for real-time operation of OROCOS components. OROCOS also does not have a universal Graphical User Interface (GUI).
2.6.2 ORCA

Open Robot Controller Architecture (ORCA) is another middleware used in partnership with another branch of middleware known as Internet Communications Engine (ICE) to create software for robotic projects [46]. It was originally partnered with the developers of OROCOS but they broke off to another stream of middleware. ORCA is comprised of many software components that can be easily linked together through the limitation of design constraints. This is considered open source and free software. The components are software pieces that can be uploaded as freeware on to online databases such as the ORCA website. Once a user has developed code, the user can decide whether they would like to share it and it can be further evaluated by other users allowing for the code to be re-created for multiple purposes instead of just one. Since most component software can be used more than once and tested across many different applications, it is feasible to develop software that is as close to universally compatible as possible. ORCA2 has overcome many issues ORCA initially had with scaling to larger distributed systems. The resolution of these issues can be read about in-depth in [46]. ORCA is still a new middleware and has not appeared strongly in any mainstream research or business applications. ORCA, unlike OROCOS, is considered a non-real time programming architecture which makes it questionable for applications needing real-time responsiveness.

2.6.3 Player

Player is a middleware open source and free software provider that has been around for a number of years [47]. It has proven to be an excellent method of both simulating and implementing applications applying to swarm or team robot applications. It has an excellent GUI and is based on the C++ language. Player was designed as a hardware abstraction layer for robots, meaning that the user need only worry about the larger parts of the program such as where should a robot drive versus programming drivers
for hardware like encoders. It is commonly used with most Pioneer platforms [48]. Player generally serves the mobile robot applications and not robot arm applications. Since the connections are TCP/IP based, it is difficult to run any true real-time applications.

### 2.6.4 CLARAty

NASA’s contribution to open source software comes in the form of CLARAty [49]. Many of the applications presented by NASA prove interesting and well thought through. However, CLARAty cannot be used in an open source context because the software does not comply with the rules of free software which were stated in [43]. It is possible to download parts of CLARAty including some algorithms for free use.

### 2.6.5 Microsoft Robotics Studio

Microsoft introduced Microsoft Robotics Studio which is their version of robotics software [50]. It has an excellent GUI (similar to that of MATLAB Simulink) and interfaces along with other Windows related software. Since it is a Microsoft project, there is a lot of financial support for this product, which makes it free (in price) for hobbyists and researchers. It is already well supported by many robotic related industries. It is not an open source product. The system must be linked together through means of extensive software writing which means that the programming environment is still fairly complex. Also, since it is a Microsoft product, it will not run on anything but Windows. At a minimum, the robot or the remote computer must be running Windows in order for the middleware to function. Also, Robotics Studio does not support abstract design, meaning that one cannot build the program in digestible layers. This will be a problem for integrating multiple systems or when new models of robots are introduced.
2.6.6 URBI

Universal Real-Time Behavior Interface (URBI) [41] is a new robotics language that was introduced to take advantage of new developments in not only robotic but computer hardware. Since it is becoming quite common to have more than one processor running on a machine, it has become the task of the programmer to re-write previous code to optimize the use of all processors involved. Gostai, the producers of URBI, have developed an entirely new language to accommodate this. Gostai’s approach to robotic software is flexible, modular, powerful, and simple [41]. Since the language promotes good processing speed, it uses mostly event-driven commands to allow for very responsive programming. URBI is able to communicate with many languages including C++ and CORBA which are highly recognized industry standards. URBI also makes use of an excellent GUI for simple program composition. The main disadvantage to using URBI is that it is not open source software nor is it free. Gostai creates URBI conversion software to integrate any language, even proprietary software. This means that the customer is stuck with whatever Gostai develops and has little freedom to optimize the linkage. Part of the reason for using open source software is to take advantage of the many programmers working on various projects to produce common ideas and eliminate flaws along the way. URBI has limited real-time capabilities, but their primary connection type is TCP/IP which will limit connection speeds making real-time unlikely for components that communicate this way. Inter-operating components will work in real-time.

2.6.7 iRobot AWARE

iRobot is quickly becoming a household name. Made famous by their military and residential applications (e.g. the Roomba, Scooba, and Looj), the developers at iRobot have developed many applications [51]. The software package AWARE is used on their robots. It is quite advanced and is made from 98% open source software. The
software is very well tested and implemented due to their heavy involvement in the growing robotics industry. The disadvantages with AWARE are that there is a large fee for buying the development software, the software is proprietary, and the software has primarily been designed for use with iRobot hardware only.

### 2.6.8 Skilligent

Although not a middleware for robotic applications, Skilligent is an excellent new technique involved in robotic application [52]. Skilligent is a program that can be run with any of the previously mentioned software packages. It is a program that allows the end-user to physically teach the robot how to do the application. This means that the programs are self-teaching and learn about their environment through interactive learning, thus traditional programming can almost be limited to abstract scripting. This software is not open source or free but it is certainly an interesting concept.

### 2.6.9 Willow Garage’s ROS

Willow Garage’s Robot Operating System (ROS) is another open source robotics project that has recently been unveiled. It is compared to Player, Microsoft Robotics Studio, and ORCA. The founder of the Player project is one of the heads for developing software for ROS [53]. Unlike other companies that make robotic software, Willow Garage develops open source software for free, and none of their components are proprietary. Since Willow Garage is a company, they offer full-support in the development of ROS components for any robotic project. Development is rapid, and well controlled. It is also relatively easy to make use of anybody else’s published code. ROS is used as an operating system for hardware abstraction, low-level device implementation of well-used functionality, message-passing between processes, and package management. ROS can currently be used with Linux and MAC operating systems and there are plans to have support for Microsoft Windows in the future. The only
disadvantage to using ROS is that it is still quite early in development and no official release of ROS has been delivered yet. However, several prestigious universities in the United States are already working with the software.

2.6.10 Others

There are many other middleware open source packages available to the consumer that were not mentioned above. Due to the fact that they either do not serve the entire robotics industry or are too young to be considered a full middleware. Some of these products include MARIE [54], CARMEN [55], OpenRTM-aist [56], JAUS [57], and ERSP [58].
Chapter 3

Mechanical Design

3.1 Overview

This chapter discusses the mechanical design for the omni-directional vehicle (ODV). The needs, problems, and requirements of the design are discussed. Then a detailed description of each mechanical subsystem of the ODV is described.

3.2 Need Statement

The ODV to be designed is required for research purposes in the Mechatronic and Robotic System (MARS) Laboratory. The ODV will be used for two purposes. The first is to serve as an omni-directional base for the development and testing of algorithms for control of mobile-manipulator systems. The second application will serve as a platform for testing various control algorithms and hardware involving autonomous operations of ODVs. This includes testing indoor localization systems, safety sensors, vision systems, etc. The need for this development is perceived because the use of omni-directional systems in industry is currently limited. However, it is believed that this research will lead to increased demand for such a product.
3.3 Problem Statement

The problem is to build an omni-directional platform that can serve both as the base of a mobile-manipulator system and as an autonomous vehicle. There are several problems that must be overcome in the design to meet the requirements of both aspects of this platform. For the mobile-manipulator system, the platform must be able to support the mass and movement of the robot that is mounted on the platform. The base must have as low a profile as possible so that the robot arm mounted on-board will not have to navigate around the ODV’s components. A low-profile would also keep the center of mass lower which will improve balance and stability.

In order for the ODV to move in the desired direction, all wheels must remain in contact with the floor. If any wheels lose contact with the ground, control will be impacted and the desired direction of the platform may not be achievable. A mechanical solution must be devised to ensure the wheels stay in contact at all times.

The platform must be able to accommodate various systems in the future. Therefore, the platform should accommodate various components while not interrupting operation.

Depending on the mode of operation, the user should be able to interact with the ODV through a relatively simple interface for driving the device. It should also be obvious to the user how to stop the robot quickly in the unlikely event of an emergency. Buttons and systems should be installed for the safety of both the ODV and the user.

A preliminary solution to this problem was presented by Agar [59]. Some of his ideas are re-iterated and expanded upon here.

3.4 Functional Requirements

During the design phase, it is important to understand the requirements of the design according to what was requested by the customer and imposed requirements
by the environment that the device that is being built will operate. The functional requirements that define the platform to be built are as follows:

- Must have the ability to travel omni-directionally (3-DOF).
- Vibrations and changes in height should be limited under full loading so as to maintain stability.
- Platform should accommodate various sizes of robot arms.
- Platform should be modular both in construction and in software.
- Operate autonomously if required.
- Operate wirelessly with no tethers for power or control.

Constraints, assumptions, and opportunities have been analyzed to aid with the concept development of the final product. These have been aimed at the functionality of the final design. Constraints state what has to happen in order for the product to function properly. Assumptions outline what the customer would assume the product should do. Opportunities outline the possibilities the product could have to exceed the customer’s expectations of the end product.

### 3.4.1 Constraints

1. Environmentally friendly.
   - The ODV should minimize the effect on the environment caused by functional parts that emit toxins or otherwise.

2. Minimize power consumption.
• Since the ODV must not be tethered, the systems on-board will need to consume as little power as possible so that the ODV will be able to run for a long time before needing a recharge. The time between recharges should be maximized.

3. Controllable speed.

• The ODV should not in any way exceed the limitations on speed imposed by the user for safety reasons.

4. Must remain fully operational during battery life-cycle.

• The ODV and its components should perform at peak capacity during the quoted operational cycle of the battery.

3.4.2 Assumptions

1. User interface will control the ODV accurately.

• However the user chooses to control the ODV, the control should be accurate.

2. Wheels will have contact with the ground at all times.

• The ODV will not operate accurately without having all wheels in contact with the surface during operation, therefore, all wheels must be in contact with the ground as much as possible.

3. Safe for users, as well as people and equipment in the work-area.

• Safety systems must be installed to allow both the user operating the ODV and the people and equipment within the work area to be safe.

4. Robust.
• User interfaces, maintenance ports, etc. should be easy for the user to interact with. The control should be intuitive.

5. Will operate well in proposed environment.

• The ODV is expected to perform well in lab and factory environments where temperature is regulated and moisture is kept to a minimum.

6. Maximize time of operation.

3.4.3 Opportunities

1. New type of omni-directional motion and/or layout.

• New methods of driving the ODV could be explored such as type of wheels. The layout of the wheels could also be explored.

2. New user interface.

• New types of control for the ODV could also be explored, or perhaps options between local and global control modes.

3. Quiet operation.

3.5 Physical Requirements

The physical requirements of the ODV also need to be explored. They are listed like the functional requirements using constraints, assumptions, and opportunities to help with conceptualization.

3.5.1 Constraints

1. Must be able to travel on smooth to minor rough surfaces and small inclines.
• The ODV is not expected to travel more than on slightly rough surfaces such as that of switching from different materials on the floor. Small inclines are also expected.

2. Minimize overall mass of the ODV.

3. Minimize overall size, including height.

3.5.2 Assumptions

1. Will withstand weight of robot arm and payload.
   • The ODV should be able to hold the robot and its payload in all the robot’s positions, accounting for the varying location of the center of mass as the robot moves around.

2. ODV will use motors.
   • The ODV is expected to travel autonomously without assistance from outside forces. Therefore, motors of some description will be used.

3.5.3 Opportunities

1. Modularity.
   • The vehicle could accommodate various components of varying size by being as modular as possible.

2. Standard parts could be used.
   • The parts that are used to build the ODV should be standard as much as possible, making maintenance and replacement of parts simple and keeping building costs of the ODV low.
3. Recyclable parts should be used.

- ODV should be made from parts that are recyclable, allowing re-use of materials that are leftover from production as well as the ODV itself at the end of its life cycle.

4. Esthetically pleasing.

### 3.6 Functional Decomposition

The basic functionality of the ODV was considered by sorting between the flow of materials, energy, and information. This is known as functional decomposition. Figure 3.1 shows a graphical depiction of the functional decomposition.

![Functional decomposition of the ODV](image)

**Figure 3.1: Functional decomposition of the ODV**

From the flowchart, it can be seen that there are seven subsystems that collectively take in the inputs of energy and information and produce the material motion of the ODV. The input information enters the system in the form of commanded motor
velocities. Energy comes in the form of electrical power. It can also be seen that
stability must be accounted for by the suspension subsystem as vibrations enter the
system during operation. A feedback loop is present amongst the subsystems to allow
for error checking between the desired velocity and measured velocity.

3.7 Design Conception

Through the patent and literature survey and the discussion of the various require-
ments of the mechanical design of the ODV, it is easier to conceptualize new ideas
for an ODV. It is necessary to first decide what type of ODV out of all the ones
that were discussed would best suit the specified requirements. The ODV must be
truly holonomic, and it must be able to carry a robot arm. Therefore, conventional
omni-directional wheels are the best option which means using either single omni-
directional wheels, double omni-directional wheels, or the mecanum wheels. Other
experimental omni-directional wheels are not readily available and may not be able
to withstand the weight requirements of this application.

Examining the three type of omni-directional wheels, single omni-directional wheels
are not desirable because they create too much noise and oscillations while in use.
Mecanum wheels are also not desirable as they cause unnecessary force on the frame to
generate the desired motion and are a patented technology. Double omni-directional
wheels do not have these drawbacks, therefore, they were selected for the ODV.

There is not much variation in the ODVs that use double omni-directional wheels.
Both three-wheel and four-wheel designs exist. Only three wheels are required to
achieve true holonomic motion, however, more wheels can be added to increase sta-

bility and support. However, adding more wheels, has the undesired effect of adding
complexity to the control process as more motors must be powered and controlled
synchronously to achieve the desired motion.
Almost all examples of double omni-directional wheeled platforms had one thing in common, their drive axes intersect the center of geometry of the platform. This means that the motors and transmission are oriented towards the center of the platform, causing other necessary components to be mounted on top which raises the center of gravity of the platform. However, one design by Agar [59] used a configuration where the drive axes were not coincident with the geometric center. This design allows for other components to be positioned lower in the center of the platform which creates a lower center of gravity that increases stability and prevents tipping. This design is also symmetrical and orthogonal.

It was decided at this stage to re-investigate the design discussed in [59] and see what improvements could be made, since the prototype that was built featured a novel wheel layout and solved a number of issues but had some shortcomings. The prototype from [59] can be seen in Figure 3.2. Agar’s design was built to carry a robot arm of varying size while allowing for stability through shocks that were installed around the wheels. The frame was made out of lightweight aluminum bars which were welded together. The plate that directly held the robot was designed to handle the bases of various robot arms.

In this work, Agar’s design was improved upon to create a new mechanical design for the ODV. The prototype, the Omnibot, can be seen in Figure 3.3. Improvements were made in the frame, motor mount, wheel mount, suspension, sizing, and electronics.
mount. The following sections will discuss each of these improvements independently while highlighting why they were improvements with respect to the requirements. The mechanical design of the Omnibot is also discussed in [60–62].

### 3.7.1 Basic Concept

The basic concept of the Omnibot is similar to that of Agar’s design. The same double-segmented omni-wheels and motors that he used were also used in this design. A suspension system was still used to keep the frame as stable as possible, and to ensure that each wheel is making contact with the floor at all times. Also, the idea of moving the motors away from the geometric center of the design was also used. This allowed for an overall lower center of gravity and improved stability in the same size footprint because the load did not have to be built on top of the drive components. Double omni-directional wheels were selected for this design because they provide the largest amount of support while avoiding any internal loading on the frame that other wheels, such as mecanum wheels, would cause. Also, a four wheel design was chosen instead of three to offer improved stability that three wheels could not deliver. Since the platform is 3-DOF but there are four actuators, the platform is redundant. Figure
3.4 shows various CAD views of the Omnibot.

### 3.7.2 Frame

Many of the requirements mentioned refer to the design of the frame of the Omnibot. One of the main requirements, being able to accommodate varying components, is of major consideration. On top of this, the various components must still remain as low as possible in the design, allowing the robot arm that will be mounted on board to have as much workspace as possible. The frame must also support the weight of the robot and its payload while being as rigid as possible to avoid unnecessary vibrations.
Table 3.1: Table describing robot arm fastening requirements

<table>
<thead>
<tr>
<th>Robot Arm</th>
<th>Bolt Diameter (mm)</th>
<th>Distance Between Holes (mm)</th>
<th>Mass (kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epson Pro Six PS3</td>
<td>10</td>
<td>200</td>
<td>45</td>
</tr>
<tr>
<td>Mitsubishi PA-10</td>
<td>17</td>
<td>175</td>
<td>40</td>
</tr>
<tr>
<td>Thermo F3</td>
<td>12</td>
<td>165.3</td>
<td>53</td>
</tr>
</tbody>
</table>

** Note: each robot base uses 4 bolts in a square pattern with the spacing described

For testing purposes, there are a number of robot arms that may be used in the final design, so it should be able to accommodate all of these sizes. Table 3.1 has information regarding the mounting and mass of the various robot arms the system must be able to hold. The frame must also allow for ample room for the wheels, motors, transmission, and suspension. The frame should also be easily assembled and disassembled for maintenance and construction purposes. The whole frame should be as light as possible as well.

Agar’s frame was a set of sturdy pieces of square aluminum welded together at intersections. While this frame was able to hold together the various components, maintenance would be difficult if damage was ever done to the frame. The frame itself was one assembled piece made entirely from non-standard parts, it was not modular in the sense that new components could be easily adapted to it, and it was heavier then it needed to be. In addition, construction of the frame was overly complex.

The new frame took these requirements into consideration through the use of extruded aluminum from 80/20 [63]. While typically used in the building of static equipment, 80/20 boasts good non-vibrational properties even though the components are held together by nuts, bolts, and threaded holes. Therefore, 80/20 products will not come apart during operation involving constant or non-constant vibrations. In addition, because 80/20 is held together by nuts and bolts, it is very easy to assemble and disassemble for assembly and maintenance purposes. A CAD view can be seen in Figure 3.5 and pictures of the actual frame can be seen in Figure 3.6.
New pieces can be made to order from existing stock with simple cutting, drilling, and tapping operations. Also, aluminum is a soft metal, which means no special tooling is required and there is very little wear on tools used for machining. The aluminum profiles from 80/20 were designed to give maximum strength while reducing the amount of material used. This means the frame can be kept lightweight while still maintaining the rigidness required for this design. All components are made from standard parts, which means parts are easily replaced and relatively inexpensive. Adjustments to component placement on the Omnibot’s frame requires only loosening of bolts, sliding the components along the bars, and then tightening the bolts once the parts are in their desired positions. New nuts can be introduced anywhere a bar is, which makes it easy to attach new parts securely.

Two bars have been placed in the middle section to support the robot as seen in Figure 3.5. Each bar is allowed to slide within the section, allowing for varying sizes of robot bases to be attached. Attaching the base to these bars will involve drilling through the support bars and bolting the robot to the frame.

Several plates were custom designed to hold the motors and axles of the wheels. They will be discussed in the following sections.
To prove the frame is strong enough to withstand the largest loads of the Omnibot, it is necessary to do a stress calculation. According to the manufacturer [63], the aluminum bars have a yield strength of 241.1 MPa, and an ultimate tensile strength of 262.0 MPa. Permanent deformation of the frame will happen if the yield strength is ever exceeded. In the event that the ultimate tensile strength is exceeded, the frame would fracture. Therefore, it is imperative to keep the stress in all members of the frame to a minimum. Through deductive reasoning, it was determined that the inner frame member may experience the most stress, specifically the inner frame member that is directly connected to the robot support beams (see Figure 3.7). Since the robot support beams are wider, they are stronger and less likely to bend under the weight of the load.

The inner frame member has three fixed points where it connects to two other inner frame members and one other outer frame member as well as two points for connecting to the suspension. It is assumed that each of these points are fixed. Therefore, only the longer part of the inner frame member will be analyzed. It will be assumed that each robot support member causes equal loading on the beam in question. The distance $a$ (100 mm) is between points of loading and their respective fixed ends as can be seen in the moment diagram in Figure 3.8. A worst case scenario is assumed,
Figure 3.7: Placement of the inner frame member on the Omnibot

Figure 3.8: Moment diagram for inner structure frame member
so the entire 53 kg load of the robot arm is assumed to be shared equally across both robot support members. Therefore, the load $W$ is the force of the load in Newtons caused by 26.5 kg on each robot support member. The inertia of the inner frame member $I$ is given to be $1.7726 \, cm^4$. The distance from the neutral axis to the edge $d$ is 12.5 mm. The maximum stress in the member can then be calculated through a beam deflection equation as [64]:

$$\frac{W \cdot a}{I \cdot d} = \frac{(9.81)26.5 \cdot 0.1}{1.7726e^{-8}/0.0125} = 18.3 \, MPa$$

(3.1)

The maximum stress in this beam is 18.3 MPa which, when compared to the yield strength of 241.1 MPa, yields a safety factor of 13. This proves that the member which was thought to be the weakest will be able to withstand the stress caused by heavy loads on the Omnibot.

### 3.7.3 Motor Discussion

There are many factors that go into selecting the correct motor for an application. Required torque, gearhead ratio, speed, type, size, control method, and encoder need to be carefully considered to determine the motor well suited for the application. To meet the demands of the design requirements, the motors must be both small and lightweight while still delivering the necessary torque and speed to move the loaded system. Since the platform is redundant, each direction has at least two motors pushing towards the direction of travel. That means the required torque only needs to be half of the total torque required to move the system. It is also necessary to have some sort of encoder to determine the position and velocity relative to the other motors on the Omnibot to enable feedback control of the system.

The Omnibot is battery powered, so as to remain untethered. Since this is a con-
trolled application, electric DC motors were the logical selection. DC motors offer a long lifespan, high efficiency and low maintenance when compared with other electric motors such as AC motors. To keep the motor size and weight down, it was necessary to look at companies that specialize in motors such as MicroMoTo keep the motor size and weight down, it was necessary to look at companies that specialize in high torque, but compact motors such as those produced by MicroMo [65]. MicroMo specializes in making motors that are small yet powerful. Using a rough estimate of the mass of the system, a selection was made. The motor that was chosen is a 12 V, 70 mNm motor with a 1:45 gearhead as seen in Figure 3.9. According to Appendix A, the recommended maximum speed is 5,000 rpm. Taking into account the gear ratio, \( GR \), and the circumference of the wheel, \( C \), the maximum velocity of a wheel can be determined by:

\[
\frac{5000 \cdot C}{60 \cdot GR} = \frac{5000 \cdot 2\pi r}{60 \cdot 45} = \frac{5000 \cdot 2\pi \cdot 0.06}{2700} = 0.6984 \text{ m/s} \quad (3.2)
\]

where \( r \) is the radius of the wheel. This proves that each motor is capable of traveling
up to 0.7 m/s.

It is also necessary to verify that the motors are capable of moving the Omnibot. Referring to Table 3.2, the unloaded mass of the Omnibot is 25 kg. Since at least two motors will produce motion for any one direction of travel, it is necessary to determine the force that two motors can produce. According to Appendix A, the recommended torque for this motor is 70 mNm. To discover how much force this motor produces at the wheel, it is necessary to know the gear ratio, \( GR \), the radius of the wheel, \( r \), and the efficiency of the gearhead, \( E \). The horizontal force produced by one motor can than be calculated as:

\[
60 \cdot T_i \cdot E \cdot GR \cdot 2\pi r = 60 \cdot 0.07 \cdot 0.9 \cdot 45 \cdot 2\pi 0.06 = 47.25 \text{ N} \tag{3.3}
\]

Since each motor is capable of producing a force of 47.25 N, two motors combined will produce a force of 94.5 N continuously. The force that works in the opposing direction is caused by rolling friction. To determine the amplitude of this force, a simple equation for rolling friction is used. A friction coefficient of 0.01 is assumed based on estimates of rolling friction from [66]. The force caused by the rolling friction is determined as:

\[
f \cdot \frac{W_{un}}{r} = 0.01 \cdot \frac{25(9.81)}{0.06} = 40.9 \text{ N} \tag{3.4}
\]

where \( W_{un} \) is the unloaded weight of the Omnibot and \( f \) is the friction coefficient. Since the force caused by friction is less than half of the force that could be exerted by the Omnibot, the Omnibot will be able to move without over exerting the motors. The complete specifications for the motor and gearhead can be found in Appendix A.
3.7.4 Motor Mount

The motor mount in Agar’s design consisted of two U-shaped bolts holding the motor to the frame. These bolts allow easy access to the motor for repair, but relied on friction between the motor and the bolt to take the torque applied to the frame by the motor and wheel. The U-bolt was not shaped to the exact inner-diameter as the outer-diameter of the motor, meaning there were only two points of frictional contact. This design allowed for the possibility that the motor could slip in place. To maintain accuracy and control, it was imperative that the motors be held rigidly and not allowed to rotate.

The motor mounts on the Omnibot are custom designed motor mount plates (see Figure 3.10). The plates were designed to fit tightly to the connection points on the chosen motors. Either end of the plate connects to L-shaped brackets that hold it to the frame as seen in Figure 3.11. Since the frame is slotted, the motor position can be altered by simply untightening the bolts that hold the motor to the frame and moving the motor. The plate was designed to resist the torque delivered by the motor, ensuring no slippage.
3.7.5 Omni-Wheel Mounting and Suspension

Since the Omnibot’s motion is based on the sum of the vectors produced by the wheels, it is imperative that all wheels stay in contact with the floor at all times. Otherwise, accurate control for autonomous motion cannot be achieved. To ensure contact, a simple suspension system pushing the wheel towards the floor at all times can be used. Suspension systems also have the benefit of limiting shocks and vibrations. This is important due to the oscillating nature of the loading of the double segmented omni-wheels used on the Omnibot. Limiting shocks and vibrations and keeping the wheels in contact with the floor are both critical requirements in the final design of the Omnibot. Other requirements that apply to the suspension system include use of standard parts, easily maintained, and must maintain a low profile.

Agar’s suspension system achieved these requirements to some extent. Each wheel had four independent springs that worked to push down the axle. A guiding plate was also used to keep the wheel stationary while allowing the axle to traverse in the direction that the springs were pushing. The suspension system was improved upon by making the overall profile of the suspension lower, evaluating the spring coefficient...
The omni-wheels are mounted on a steel hexagonal axle. This axle is held on either side by a pillow-block bearing that allows the axle to rotate freely. The axle is notched on either side of the wheel to allow for the axle to fit into two slotted plates that hold the omni-wheel on either side. The guide plate is featured in Figure 3.12. These thin slotted aluminum plates are secured to the frame, allowing the axle to rotate and travel up and down in the direction of the suspension while preventing unwanted translation. Two bolts are fastened to each pillow block after each bolt is passed through holes in the top part of the frame. The springs that support the wheel vertically are wrapped around the bolts and compressed between the frame and pillow block. The bolts are allowed to slide through the frame so that compression can occur on the springs. The bolt head is accompanied by a rubber washer to reduce noise. A picture of the suspension can be seen in Figure 3.13 and an exploded view of the suspension system can be seen in Figure 3.14.

Spring sizing can occur once the total mass of the load is estimated. The selected springs have a free length of 89 mm and a spring coefficient of 3.33 N/mm. Due to the low profile of the other components, the spring only has a total of 30 mm of displacement available for travel. It is also necessary to have the spring already
Figure 3.13: Omnibot suspension

Figure 3.14: Exploded view of the Omnibot suspension with labels
Table 3.2: Estimated total mass of the Omnibot

<table>
<thead>
<tr>
<th>Part Name</th>
<th>Mass ($kg$)</th>
<th>Quantity</th>
<th>Total Mass ($kg$)</th>
<th>Total Weight ($N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor</td>
<td>0.63</td>
<td>4</td>
<td>2.52</td>
<td>24.71</td>
</tr>
<tr>
<td>Universal Joint and Connectors</td>
<td>0.34</td>
<td>4</td>
<td>1.37</td>
<td>13.43</td>
</tr>
<tr>
<td>Square Connector</td>
<td>0.02</td>
<td>4</td>
<td>0.08</td>
<td>0.75</td>
</tr>
<tr>
<td>Interface Plate</td>
<td>0.12</td>
<td>4</td>
<td>0.50</td>
<td>4.87</td>
</tr>
<tr>
<td>Robotics Arm</td>
<td>53.00</td>
<td>1</td>
<td>53.00</td>
<td>519.93</td>
</tr>
<tr>
<td>Fasteners</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>0.78</td>
</tr>
<tr>
<td>Frame</td>
<td>13.86</td>
<td>1</td>
<td>13.86</td>
<td>136.00</td>
</tr>
<tr>
<td>Battery</td>
<td>2.41</td>
<td>3</td>
<td>7.23</td>
<td>70.93</td>
</tr>
<tr>
<td>Microcontroller and Amplifiers</td>
<td>1.00</td>
<td>1</td>
<td>1.00</td>
<td>9.81</td>
</tr>
</tbody>
</table>

compressed by about 10 mm to allow for vertical travel downward. This is to allow for minor depressions in the floor because each wheel must stay in contact. This also means that the spring can only compress 20 mm from its standard position. The maximum force the springs will undergo is double that of the estimated weight of the system. Table 3.2 was produced to roughly estimate the total weight of the system. Only the components on the Omnibot that apply a downward force on the springs were considered in the calculation for the total weight because the other components will have no effect. Assuming a symmetrical platform and, therefore, equal weighting on each spring, the various spring lengths can be calculated. A spring has three lengths to consider: the free length $l_f$, the installation length $l_i$, and the operating length $l_o$. The free length was already given as 89 mm. The installation length can be calculated as:

$$l_i = l_f - \frac{W_{total}/16}{k} = 89 - \frac{781.21/16}{3.33} = 74.3 \ mm$$

(3.5)

where $W_{total}$ is the total weight of the system in Newtons and $k$ is the spring coefficient.
measured in millimeters. The operating length can be determined by doubling the weight on the spring. This results in an operating length of 60 mm. Since the operating range of the spring is 30 mm and the free length is 89 mm, it can be determined that the largest amount of compression the spring can experience is 59 mm. Since the maximum compression is 60 mm, the spring will work within the operating range of the system.

Since the platform is modular, and the weight of the system may change with the addition of new components, the springs will not always stay within the allowable operating range. To counter this, the installation length of the spring can be decreased by adding bushings or washers, allowing for the same operating range with more load.

### 3.7.6 Transmission

With the frame, motor mount, and wheel mount decided, the problem remained of how to transmit the power from the motor to the drive axle of each wheel. The axle is allowed to displace 30 mm vertically while the motor is fixed to the frame. A link was needed that could transmit the torque and be rigid enough to maintain accuracy, while still allowing for easy access for maintenance and be made of standard parts as much as possible. The motor shafts are keyed circular shafts, while the axles of the omni-wheels are hexagonal.
It was decided that a double universal joint would work well in this system. However, a simple double universal joint would not allow for the translation in the direction collinear with the axle. Since there is vertical movement, there must also be horizontal movement caused by the vertical movement. Therefore, it was planned to use a double universal joint with the ability to translate linearly. The exact amount of the horizontal translation can be determined by the vertical displacement and the length of the shaft between the two universal joints. The longer the shaft, the less horizontal displacement there would be.

The final design of the transmission used parts you would find in a socket wrench set that are readily available in any hardware store as can be seen in Figure 3.15. Since socket wrenches are normally hexagon shaped, a socket wrench head worked perfectly as a coupler to the hexagon axle of the omni-wheel. The rest of the axle is made from two universal joints, a socket head wrench extender, and another hexagon socket head. The other socket head was fit as tightly as possible to the motor shaft, and then a hole was drilled through both the motor shaft and socket head so that a pin could be used to hold the assembly together. The extender was used to limit the horizontal movement of the axle. With this extender, the horizontal displacement...
was calculated to be only 4 mm which is acceptable for the coupling. There is some
translational slip allowed in the connection between the axle and socket head, and
between the universal joint and the large socket head with the use of a slip block.
All of these components can be seen in the exploded view in Figure 3.16. This slip
provides the necessary 4 mm translation required. Since the motor is already held
firmly by the the frame, and the axle is not allowed to move horizontally because of
the wheel mount, the assembly of the transmission cannot come apart on its own.
Access to the transmission is still easy though because only the motor plate must be
loosened to open the linkage.

3.7.7 Electronics Mount

For control, the motors must be powered and receive control signals. The components
that deliver this are the microcontroller, batteries, and motor amplifiers. These sys-
tems must be mounted on-board the Omnibot. Agar’s previous design had a plate
that was elevated above the rest of the platform to hold these components. It was
noted during the re-design that this plate must be lowered as much as possible to allow for an over-all low profile of the Omnibot. The new plate has been custom fit to hold the power source. The board was attached to the frame with nuts, bolts, and bushings, so that it can be easily removed for maintenance purposes. The plate also houses the microcontroller, a breadboard, and four amplifiers to boost the signals given from the microcontroller to the motors. A top-down view of the electronics section on the Omnibot can be seen in Figure 3.17.

Modularity was maintained when designing the electronics mount. The three batteries are held firmly on four sides by blocks that protrude from the baseboard of the mount. It is easy to replace a battery by unplugging the battery at its terminals, lifting out the battery, and replacing it with a new one by dropping it in. Tight tolerances keep the batteries from sliding. The base is connected to the frame by six bolts and nuts, and the entire board can be removed and replaced with ease. All other components are screwed on to the baseboard ensuring none of the components move during operation.

3.7.8 Summary

This chapter reviewed the mechanical design of the Omnibot. The functional and physical requirements of the design were outlined as well as the needs and wants of the customer. The Omnibot was shown in detail highlighting the basic concept, frame, motor discussion, motor mount, omni-wheel mount, transmission, and electronics mount.
Chapter 4

Kinematics, Control, and Electrical Systems

4.1 Overview

In this chapter, the controller development and implementation are discussed. The geometry of the Omnibot is shown and equations are developed for describing control of the Omnibot. Both local and global control are discussed, as well as the electrical systems that allow the Omnibot to operate. The developed control methods and electrical systems are also discussed in [60–62].

4.2 Omnibot Kinematics

The direction and velocity of an omni-directional vehicle (ODV) is simply the sum of the velocity vectors produced by the wheels. However, the vectors will have a different effect on the motion depending on where the omni-wheels are placed relative to the other omni-wheels. For a holonomic vehicle, there are only three variables that define its motion: the translational velocities in the $x$ and $y$ directions, and the angular velocity about the $z$ axis.
Kalmar-Nagy et al. [32] did a derivation for a three wheeled ODV as was discussed in Section 2.3.3. A similar derivation can be done for the four-wheeled Omnibot. A basic layout of the Omnibot with the unit vectors of each wheel, the geometric center, the global position, and their relation to one another is shown in Figure 4.1. From this figure, the position of each of the wheels relative to the geometric center are given by:

\[
\begin{align*}
p_{01} &= l \begin{bmatrix} \cos 45^\circ \\ \sin 45^\circ \end{bmatrix}, \\
p_{02} &= l \begin{bmatrix} \cos 135^\circ \\ \sin 135^\circ \end{bmatrix}, \\
p_{03} &= l \begin{bmatrix} \cos 225^\circ \\ \sin 225^\circ \end{bmatrix}, \\
p_{04} &= l \begin{bmatrix} \cos 315^\circ \\ \sin 315^\circ \end{bmatrix}
\end{align*}
\]

(4.1)

where \(p_{0n}\) represents the position of wheel \(n\) relative to the geometric center and \(l\) is defined as the fixed distance between the geometric center and the center of the omni-wheel.
The directional unit vectors $d_n$ for each wheel can also be developed using the general equation:

$$d_n = \frac{1}{l} R(45^\circ) p_{0n}$$  \hspace{1cm} (4.2)

where $R$ is the rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$  \hspace{1cm} (4.3)

and $\theta$ is the angle of rotation of the platform. With these equations derived, a Jacobian matrix can be determined as [32]:

$$p_{0n}^T R^T (\theta) R (\theta) d_n = l \theta$$  \hspace{1cm} (4.4)

where $\dot{\theta}$ is the angular velocity of the platform and $l \dot{\theta}$ represents the tangential velocity of the platform. Applying Equation (4.4) to the omni-directional platform being studied yields the Jacobian matrix ($J$) that can be used to determine the velocities of all four wheels to drive the system as:

$$V = J \dot{x}$$

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} -\sin \theta & \cos \theta & l \cos (45^\circ) \\ -\cos \theta & -\sin \theta & l \cos (45^\circ) \\ \sin \theta & -\cos \theta & l \cos (45^\circ) \\ \cos \theta & \sin \theta & l \cos (45^\circ) \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$  \hspace{1cm} (4.5)

where $v_n$, $n = 1$ to 4, are the wheel velocities and $\dot{x}$, $\dot{y}$, and $\dot{\theta}$ define the velocity of the platform. Equation (4.5) maps the desired velocity of the platform to the required wheel velocities. Similarly, the wheel velocities can be mapped to the platform velocity...
using:

\[ \dot{x} = J^{-1}V \] (4.6)

An interesting observation involving the geometry of the Omnibot is that the Omnibot can achieve a different top speed depending on which direction it is driving. If two motors are driving in the same direction, while the other two are locked, the maximum speed is 0.7 m/s which is the max speed with two motors acting in the same direction. However, if all four motors are driven at their max speed, which would cause the Omnibot to drive at an angle, the max speed is closer to 1 m/s. This is a result of summing the vectors. The difference between horizontal and diagonal motion can be seen clearly in Figure 4.2.

The Omnibot is also capable of rotating in its own footprint. By rotating all of the motors either clockwise or counter-clockwise with respect to the driven axis, a rotational motion is achieved. This rotational motion can be seen in Figure 4.2. By combining horizontal, diagonal, and rotational motions, the Omnibot is capable of driving in any direction, simultaneously translating and rotating.

4.3 Control Modes

Using Equation (4.5) to govern its motion, the Omnibot can now be controlled using velocity control. There are two modes in which to control the Omnibot: local and global control. Local control is when the position of the Omnibot is not factored into its operation, meaning that the Omnibot drives with respect to its present position and orientation. The system cannot distinguish its position relative to any fixed point in its workspace. Local control is easiest to envision by imagining the user is sitting on the Omnibot and driving. The forward direction is always ahead of the user, regardless of how much the platform rotates in the workspace. On the contrary,
global control uses position tracking to determine where the Omnibot is with respect to a global coordinate system. The pose can be tracked by a localization system. Global control can be visualized by driving the Omnibot from above.

### 4.3.1 Local Control

Local control is a much simpler method of control from an implementation point of view because local control does not require a localization system. A user interface such as a joystick is set to communicate movement commands to the Omnibot. When the user indicates through the joystick to move in a certain direction, the platform moves in that specified direction at a velocity proportional to how far the joystick is pushed in that direction. The velocities that the wheels move at must be synchronized because...
at least two wheels are moving in one direction for any one commanded movement from the user. Digital encoders are attached to each motor to help achieve this goal of synchronization. The encoders are used to determine the velocity of each motor, and thus each wheel, through finite differencing.

Referring to Figure 4.1, to achieve a linear motion along the $x$ axis, wheels 1 and 3 are driven. For only $y$ axis motion, wheels 2 and 4 are driven. To drive on an angle, a combination of both $x$ axis and $y$ axis motion is needed, so all four wheels are used. Rotation can be achieved by rotating all four wheels in the same direction (clockwise or counter-clockwise) such as the unit vector directions presented in Figure 4.1.

Since local control only works on a point-to-point basis, $\theta$ can be set to zero. Consequently, the Jacobian matrix derived in Equation (4.5) can be simplified to:

\[
\begin{bmatrix}
  v_1 \\
v_2 \\
v_3 \\
v_4
\end{bmatrix} =
\begin{bmatrix}
  0 & 1 & l \cos (45^\circ) \\
-1 & 0 & l \cos (45^\circ) \\
0 & -1 & l \cos (45^\circ) \\
1 & 0 & l \cos (45^\circ)
\end{bmatrix}
\begin{bmatrix}
  \dot{x} \\
  \dot{y} \\
  \dot{\theta}
\end{bmatrix}
\]

(4.7)

### 4.3.2 Global Control

Since global control relies on pose feedback, it is the more complicated type of control to implement. Not only is a localization system involved, but other subsystems are required as well. A schematic is shown in Figure 4.3 which presents the subsystems that control the Omnibot.

From a kinematic perspective, global control is defined by the Jacobian presented in Equation (4.5). Since $\theta$ is no longer evaluated to zero, the drive direction can differ depending on the current orientation of the Omnibot. For visualization purposes, global control makes it easier for a user to drive the Omnibot, because no matter
which way its facing, the platform will always travel in one global direction when commanded, despite the current pose. The challenge with global control is how to get accurate pose information in a timely manner to the Omnibot.

### 4.3.3 Local Control with Directional Feedback

Another type of control is given by using a combination of local and global control. If the only feedback the ODV receives is the orientation or $\theta$, the platform can be driven in terms of a global orientation instead of a local orientation. This type of control is more intuitive for a user who is manually driving the ODV from a plugged in joystick. Forward would be according to a direction indicated by the orientation feedback such as an electronic compass. The user no longer has to keep track of which way the ODV is facing, which can be tricky when the design is symmetrical.
4.4 Electrical Systems

The Omnibot contains many electrical systems that help either directly or indirectly drive the Omnibot. The systems are summarized in Figure 4.3. The onboard motion controller and optional manual control interface are completely implemented, and will be discussed in detail. A preliminary implementation on the global localization system has been completed and will be discussed in detail as well. The vision system and collision detection systems have yet to be implemented and are not part of this thesis. The onboard data processor and remote host are components that serve as a communication highway between these various subsystems as well as future subsystems that may be added. The onboard data processor and remote host will be discussed in greater detail in Chapter 5.

4.4.1 Base Drive System

The base drive system comprises various electrical components, most of which can be seen represented in Figure 4.4. The components are the power source, circuitry, amplifiers, motors, encoders, encoder counters, direct user interface, safety components, and the microcontroller.

4.4.2 Power Source

It was decided that 12V gel-cell batteries would be used to power the Omnibot as these are readily available. It was determined that the power source would consist of three 12V gel cell batteries, two for the motors and one for the microcontroller. It was decided that the microcontroller would have an independent power source so as not to be affected by the fluctuating current draw from the motors. It is difficult to determine how long the batteries would last in general operation. However, if assumptions are made, an estimate can be made to the total amount of
time the system will operate before needing a re-charge.

It is assumed that the pair of batteries that power the motors will not last as long as the microcontroller, based on the current draw. The motors will draw up to 4 A continuously, however, the motors will not usually experience continuous draw above 2 A while in operation. Since two 7 Ah batteries have double the capacity, 14 Ah, when connected in parallel, and four motors are in use, it is reasonable to assume that the Omnibot will remain in continuous operation for 1.75 hours. This is ideal for laboratory testing, since the Omnibot will not be used continuously in the laboratory. However, should the Omnibot be used in a factory setting, it would be ideal to have a power supply with more capacity. It should also be noted that the motors draw more power if the surface the Omnibot is traveling on is rough. Therefore, for maximum time between charging, the Omnibot should only be driven on relatively flat, clean,
and low friction environments.

For the microcontroller and accompanying circuitry, a 5 V source was required. A voltage regulator was used to convert the 12 V battery supply to a 5 V source.

4.4.3 Circuitry

The circuitry on the Omnibot has been designed to be modular for many components. Referring back to Figure 3.17, it can be seen that a breadboard has been mounted directly beside the microprocessor and in close proximity to most of the other electronic components. The breadboard has two power strips, one at either side. One side provides the 12 V source, and the other the 5 V necessary to power all the components. The breadboard allows for easy adaptation of new electronic circuits, as well as easy replacement of burnt out components should the need arise during testing.

The motors and microcontroller are safe-guarded by current limiters. The microcontroller is guarded by a 0.25 A fast blow fuse. The motors are individually guarded by a 4 A breaker specifically designed for 4 A motors. These breakers allow for current over 4 A for a brief period of time, as this is acceptable for normal motor operation.
The microcontroller itself is connected to the Omnibot by a 30 x 2 male pin connector and a barrel connector for power. The barrel connector is used to easily connect and disconnect the power as necessary. A custom designed printed circuit board (PCB) was used to easily mount and unmount the large 60-pin port to the Omnibot. This allows for the microcontroller to be easily removed and replaced while not disconnecting any of the wiring on-board the Omnibot which can be seen in Figure 4.5. The PCB features two 30 x 2 female pin connectors mounted side by side. One holds the microcontroller while the other serves as ports for individual wires for each pin of the microcontroller. It should be noted that the pins are reversed from one side to the other, meaning the two columns of 30 pins swap for the other port. This one layer PCB was made using a photoengraving process. A cellophane sheet with the desired image of the copper to be left behind was used. The image can be seen in Figure 4.6. The photoengraving process involves a light exposure process that allows the exposed copper to be removed through several chemical baths. Once the circuit board was developed, the holes were drilled manually and the female pin connectors were inserted and soldered in place. The PCB was mounted using washers and screws to ensure the exposed side of the circuit board was not resting on the wooden baseboard.

4.4.4 Amplifiers

An amplifier is used to convert the pulse width modulation (PWM) signal that is sent from the microcontroller into a voltage the motor can use. The 5 V PWM signal
that comes from the microcontroller is an on-off signal that has a fixed period but the percentage of time that the signal is on in the period can be adjusted. This percentage is known as the duty cycle. By varying the duty cycle, the average amount of power to the motor can be increased or decreased. The current and voltage of these 5 V signals is insufficient to run a 12 V motor, so an amplifier is necessary to step up the voltage and current.

The amplifiers that were initially selected were the model Victor 884 from IFI Robotics [67]. Each model features a 12 V power connection, motor connector, individual cooling fan, and a port for connecting to a PWM signal. Features include a jumper that can allow the user to have the option to run the motor with electronic braking on or off, a calibration button, and an indicator light-emitting diode (LED). The LED is used to indicate the state of the motor and problem states should issues arise. Full speed forward on a motor is a solid green light, full speed reverse is a solid red light, and a solid orange light indicates a full stop. Once properly connected to a motor, the calibration pin is depressed while moving the control interface from full speed forward to full speed reverse. Further information about the Victor amplifier can be found in Appendix A.

The Victor 884 amplifier uses only a PWM signal to determine the full range of speed from full forward to full reverse. It updates at a frequency of 120 Hz, and accepts a PWM signal varying in duty cycle range of 1 - 2 ms. This means that the motor will be traveling at full speed in reverse with a duty cycle of 1 ms, it will be stopped with a duty cycle of 1.5 ms, and it will be rotating full speed forward with a duty cycle of 2 ms. According to IFI Robotics, the Victor is supposed to have 94 different speeds between being stopped and rotating full speed in a given direction. In testing, however, the number of different speeds is much less for this configuration. Referring to Figure 4.7, it is clear to see that there are only nine steps between stop and full speed (either direction) of the Victor. This would require the control of the Omnibot.
to be done in a more discrete fashion due to the limitation of available motor velocities. The results in Figure 4.7 were found by using an encoder attached to the motor to calculate the velocity as the duty cycle of the PWM signal was changed slowly from 1 - 2 ms. The wheel was not in contact with the ground during this test, therefore, the motor was in a no-load condition.

The limitations caused by the Victor 884 lead to the selection of another amplifier. The Dual VNH3SP30 Motor Driver Carrier MD30A from Pololu Robotics & Electronics [68] was selected as the replacement. This motor driver is capable of delivering a full range of different velocities between zero and maximum speed. Instead of using only a PWM signal as input, it also uses two digital i/o (on-off) ports to control the direction and braking of the motor. This means the full range of the PWM signal can be used to control the speed of the motor, which allows for more steps. These motor drivers are also capable of controlling two motors with one IC, therefore, only two MD30A amplifiers were needed to run the four motors on the Omnibot. Further information about these motor drivers can be found in Appendix A.
4.4.5 Encoders

A rotary encoder is a device that enables a processor to track rotation in a motor. An optical encoder works through the use of optical sensors, a reflector, and a slotted disk that is attached to the motor shaft. As the shaft turns, an optical sensor will be able to see through the slots as they pass by to see the reflector on the other side. The optical sensor emits a high signal when it views the reflector and a low signal when it does not. For a quadrature encoder, a second optical sensor is used slightly offset from the first optical sensor. A combination of the two signals gives four unique types of pulses: on-on, on-off, off-off, and off-on. This is illustrated in Figure 4.8 where $C$ represents the entire period of the two signals and $S_1$ to $S_4$ represent each of the four states. Direction is easily tracked with this type of encoder because the two bit signal can only go through the sequence in a specific order and that order is reversed when the shaft rotation changes direction.

Each motor on the Omnibot has an optical quadrature encoder attached to the drive shaft. Each encoder has two channels and 1,024 lines (or slots) per revolution. The two channels allow the processor to see four pulses per line, and the ability to see changes in direction easily. Since each line has four pulses, there are 4,096 pulses in

![Figure 4.8: Graph showing quadrature encoder transactions](image)
one full rotation of the motor shaft. Details of the encoder can be found in Appendix A. Using these encoders, the velocity of the wheels can be tracked. The encoder is attached directly to the motor shaft, before going through the gearhead. Since the gearhead has a ratio of 45:1, the encoder gives 184,320 pulses for every one rotation of the wheel.

The omni-wheel radius is 60 mm. The pulse per distance can be calculated using $C$, the circumference of the wheel, through:

$$\frac{n_{\text{rot}} \cdot GR}{C} = \frac{(4 \cdot 1024) \cdot 45}{2\pi r} = \frac{184320}{2\pi (0.06)} = 488736 \text{ pulses/m}$$

where $n_{\text{rot}}$ is the number of pulses per rotation of the motor, $GR$ is the gear ratio of the gearhead, and $r$ is the radius of the omni-wheel.

Since each omni-wheel is capable of driving up to 0.7 m/s, the encoders are capable of delivering up to 342,115 pulses per second. Since there are four motors, the microcontroller has to be able to manage all of this counting while still meeting all of the other events that need to happen. To assist the microcontroller with this task, several integrated circuits (ICs) were used to keep track of the encoders as the Omnibot moves.

### 4.4.6 Encoder Counters

The encoder counter is a HCTL-2021 Quadrature Decoder/Counter Interface IC. A data sheet for this IC can be found in Appendix A. The encoder counter has three inputs, a 5 V power source, a quadrature encoder, and a crystal oscillator that produces a signal frequency of 10 MHz. The counter keeps track of how many pulses the motor has rotated one way or the other and, when requested, will deliver that number in two parts to an 8-bit port. The encoder counter will count up to 16-bits, or 65,535
before resetting back to 0. Since the number is 16 bits long, but only 8 bits can be displayed at one time, the number is split into 2 halves. Activation of a separate pair of bits will cause either the first or second half of the number to display across the 8-bit port, or no number at all. The microcontroller requests for the largest half of the number before the smaller half, and then combines them inside the program for evaluation of the velocity. Since four encoders are used, four encoder counters are used. The wiring diagram for the encoder counters can be found in Appendix B.

4.4.7 Direct User Interface

The manual interface is a 3-DOF Hall effect sensing joystick from CH Products [69]. The joystick is capable of moving in three different directions at once, that is $x$, $y$, and twist or $\theta$ (see Figure 4.9). The joystick also takes a 5 V source to output the three analog signals to be interpreted by the microcontroller. The joystick is connected to the Omnibot through a regular serial cable, allowing for a varying length in the connection cable. The tolerance of each analog signal is $\pm 2\%$ which is significant.
when discussing the summation of signals. The interpretation of the joystick signals will be discussed further in Section 4.4.9.

4.4.8 Safety Components

The Omnibot features an Emergency Stop (E-stop) button that is connected directly to the frame (see Figure 4.10). When activated, power is stopped to the motors, preventing all motion from happening. In future work, a collision detection system will be installed to sense the presence of objects within close proximity of the Omnibot. Such a system would allow the ODV to change its trajectory or simply stop if it encountered an object or person in its workspace. The speed of the platform has also been limited to prevent the Omnibot from traveling too quickly.

The 3-DOF joystick that can be plugged into the Omnibot for direct manual control (see Figure 4.9) also has several safety features. In order for the user to drive, a dead-man switch must be pressed and held. If the dead-man switch is released, the brakes on the Omnibot will lock up causing it to stop. Also, in the event that the cable between the Omnibot and the joystick becomes disconnected, the Omnibot will stop.
4.4.9 Microcontroller

As can be seen in Figure 4.4, the microcontroller interacts either directly or indirectly with all drive components on the Omnibot. The microcontroller’s primary job is to interpret the commands sent from either the remote station or from the manual joystick and translate them into motor commands. It is also the job of the microcontroller to ensure that the motors are moving at their commanded speeds through encoder feedback.

The HCS12 microcontroller from Freescale Semiconductor Inc. was selected for the processing requirements of the Omnibot [70]. As was mentioned, the microcontroller is mounted on a PCB, allowing easy connection and disconnection. The microcontroller is partnered with a development board that allows the microcontroller to be easily programmed and tested. The serial port on the microcontroller can also be configured to output relevant data so troubleshooting can be conducted while the microcontroller is mounted on the Omnibot. Commands can also be sent through the serial port to the microcontroller.

The microcontroller is directly connected to the joystick, amplifiers, encoder counters, and power source. The joystick communicates through three analog signals that are converted by the analog to digital converters on the microcontroller. The program converts each signal to a byte, which is between 0 and 255. Full speed reverse is 0, full speed forward is 255 and 127 is full stop. The analog to digital converts the signal a number of times and takes an average of the values to help minimize the error in the original analog signal. To compensate for the rest of the error, the program enforces a deadband by keeping the brake value at 127 if it varies anywhere within the range of tolerance of the full stop signal (approximately a value of 120-135). If connected to another interface such as a remote driving program, a deadband may not be required. The commanded velocity is then converted into a decimal number in rad/s. This conversion is necessary because it is used for direct comparison to the measured wheel
velocity for velocity control, which is discussed in Section 4.5.
The encoder counters submit data to the microcontroller upon request. One byte
is used to receive the packages of half 16-bit numbers and the other byte serves to
activate and deactivate each of the eight halves of numbers received from the four
encoder counters. The program requests for the data regularly and analyzes it to
determine the velocity of the specified motor. This is done by getting the difference
between the previous encoder counter value and the current value and then dividing
this number by the amount of time to receive the two pieces of data. This results
in an estimated velocity of the motor which is then converted to the wheel velocity.
This information is then used for feedback control.
The amplifiers require a PWM signal and two phase signals for each motor. The
program on the microcontroller, after interpreting the commanded velocity from the
user interface, delivers the commands to the motor to deliver the appropriate power.
The controller must use the encoders to determine the current velocity and change
appropriately based on the commanded velocity. The velocity controller must dictate
the PWM value and the direction of the motor as it continues to receive the commands
and feedback. The microcontroller also monitors two on-off ports: the dead-man
switch and the mode switch. The dead-man switch will stop the motors immediately
if released and the mode switch serves as an interface to switch between local and
remote operation (see Chapter 5 for more information).
The microcontroller program can be seen represented in Figure 4.11. The program
reads in the available commands through the input interface. The commands are
then sent to the Jacobian conversion to translate into four motor velocities from three
commanded velocities. The commanded velocities and the measured velocities are
then used to determine the output speed for the motors. The full program can be
seen in Appendix D.
4.5 Velocity Control

The controller mounted on-board the Omnibot was designed to be used for both manual and autonomous control. Manual control is done either through a hard-wired user interface straight to the Omnibot or through a wireless link. Both autonomous and manual control require velocity control.

Since each wheel is individually driven by its own motor and transmission, it is up to the controller to be sure that each motor is moving at the commanded velocity. A feedback loop in the program was created to adjust the velocity of each wheel independently. While in operation, each motor velocity is compared to the commanded velocity for that motor. This means that if a motor is going too slow, it will be commanded to go faster to compensate. Likewise, if the motor is moving too quickly, the motor is commanded to slow down. Proportional-Integral-Derivative (PID) control is used as the control algorithm for each motor.
PID control works by using the measured error between the commanded velocity and the actual velocity. By applying this error along with the proportional, integral, and derivative gains, velocity control can be produced.

The output velocity is calculated given the desired velocity, measured velocity, and the error using:

\[ e = v_d - v_{out} \]
\[ v_{out} = K_P e + K_I \int e dt + K_D \frac{de}{dt} + v_d \]  

where \( K_P \) is the proportionality gain constant, \( K_I \) is the integral gain constant, \( K_D \) is the derivative gain constant, \( e \) is the error, \( v_{out} \) is the measured velocity, and \( v_d \) is the desired velocity. Figure 4.12 shows a block diagram of the PID controller for one motor.

The PID constants are tuned to make the motors run in a desired fashion. Each constant has a different effect on the output of the PID system. The proportional gain, when increased, produces a faster response time. However, a large proportional gain can also cause instability and unwanted oscillations in the motor. Also, proportional gain by itself causes a steady state error, which means the error never reaches zero.

To combat steady state error, the integral gain is used. The more the integral gain is increased, the quicker the steady state error is eliminated and the faster the response time. However, larger integral gains cause overshoot and longer settling times. Large
overshoot and long settling time is undesirable, so the derivative gain is used as well. By increasing the derivative gain, the overshoot and settling time is decreased. However, increasing the derivative gain also slows the response time and also causes instability. To get good control for a motor, the PID constants must be tuned. This was done using a manual process that will be discussed in Section 6.3.

Each motor is controlled by an independent PID feedback loop. Since the motors are of the same make and model, they share the same gain values. However, each motor does not perform exactly like the other, therefore, during operation, one motor may have more error then the other due to physical causes. The physical causes could be more friction in the transmission for one motor or the load not being completely balanced causing unequal loading across the four wheels. PID is used to address these problems and ensure that each wheel is moving as close to the desired speed as possible.

For local control, the PID feedback control on each motor is enough to ensure the motors are moving at the same desired velocity proportional to the other motors. It is not practical to ensure that each wheel is moving proportional to one another because course correction can be made by the operator. If such a system were implemented, the operator would have a hard time noticing if the vehicle is not performing at peak condition. For instance, if one wheel could only perform at 70% of the commanded velocity, the other motors would also have to slow down to 70% to match the slow motor. Continuous operation in this state could go unnoticed, and may cause the operator to believe the battery is dying instead of seeing the actual problem.

The proposed global control system should be able to handle variances in the commanded pose caused by motors that are not performing at peak condition. By using the feedback from the global system, an autonomous vehicle can have the course corrected like it would if a user was driving it manually. Velocity control using global feedback data will also be possible, but is not a goal of this thesis.
4.6 Indoor Global Localization System

The global localization system will help define the pose of the ODV relative to a defined world coordinate system. A localization system is necessary because an autonomous mobile system cannot operate without one. It is also ideal to use a global localization system in factory type environments, which fits the requirements of the Omnibot. Localization systems can be either active or passive.

In both active and passive localization systems, there are signals between sensors and a greater network of known positions to determine location of a node. An active system relies on a beacon mounted on a mobile device to request position information from receivers. The receivers communicate with each other, and determine the beacon’s location relative to the known positions of the receivers. The receivers then broadcast the position back to the beacon in the mobile device. The information is public because the position information of the mobile device is known to all devices involved. This information can then be passed to other systems through the network. Examples of active systems include UbiSense and Active Bat [71–73].

In a passive localization system, the beacons are fixed, and the receiver is on the mobile device. Position information is broadcast from the beacons and the receiver determines where it is based on the distance it is away from the known beacon locations. Since the beacons do not know which sensors they are broadcasting to, the information remains private to the mobile device, meaning tracking from other systems is not possible without explicit permission from the mobile device.

There are clear advantages to using either active or passive localization systems for tracking mobile devices. Active systems have a large amount of communication happening between receivers, allowing for faster communication and calculation time. Since all receivers are listening to the same signal coming from a beacon, processing happens almost instantaneously. Passive systems lack this advantage because receiving can only happen one beacon at a time. By the time enough beacons have been
heard, a significant amount of time has passed and the error in the position calculation is greater. On the other hand, passive systems have the advantage of easily having a large number of beacons because the number of transmitters is not limited. This allows for a much larger workspace. Active systems must limit the number of receivers because a large network between components must be made in order for simultaneous communication to happen between nodes.

The Omnibot uses MIT’s Crickets [74–76] for global localization. Crickets are considered a passive localization system (see Figure 4.13). Crickets use ultrasonic signals (US) and radio frequency (RF) to determine their positions relative to one another based on the difference in speed between the US and RF signals. Through triangulation, the Crickets can interpret their location and, ultimately, the pose of the Omnibot and communicate to the controller of the Omnibot the information for control purposes. Once the positions of the Crickets on the Omnibot (see Figure 4.14) are known, the pose of the Omnibot can be determined. With this data, the Omnibot can be controlled both remotely and globally in near real-time. It is also important to note that each Cricket is programmable, and can be re-configured to an active or a hybrid system depending on the programmer’s desires. Since Crickets can be converted to active or passive systems, it was the ideal choice for the Omnibot because of the Omnibot’s experimental nature.

Crickets can either be listeners, which receive signals, or beacons, which send signals. The beacons are fixed on the roof of the workspace and the listeners are mounted on
the Omnibot. Through calibration, the beacons are given their position relative to one another and, after calibration, constantly broadcast their location for the listeners to receive. Once the listeners are activated, they receive signals from multiple beacons, they then triangulate their position based on the positional information from the beacons and by how far they are away from the beacons.

Distance is measured through the listeners by first receiving a RF signal from a beacon then receiving the US signal after some time has passed from the same beacon. Since both the RF and US signals are sent at the same time from the beacon and the US signal travels $10^6$ times slower than the RF signal, the time between receiving the RF and US signals is proportional to that of the distance between the beacon and the listener. Using this method, a listener can calculate the distance it is away from a beacon. A listener must calculate its distance away from at least three different beacons before determining its single solution position.

There are at least two listeners mounted on-board the Omnibot, and each listener must determine its location in order to determine the pose of the ODV. The reason two listeners are being used is to determine a single solution for position and orientation of the Omnibot. However, in practical application, three or more listeners are used so as to help decrease the potential error in the signals through redundancy. The controller that receives the position data from the listeners then determines the most likely pose of the Omnibot based on the location of the listeners relative to the geometric center of the Omnibot.

Once the pose is known relative to the global coordinate system, a comparison can be made between the commanded pose of the Omnibot and the actual pose. Adjustments can then be made to the trajectory as a result of the error between the two poses. Since this is a passive localization system, there is error associated with each measurement made. It is expected that the Cricket localization system, using three or more listeners, will produce an error within a tolerance of $\pm 5$ cm which is considered acceptable and
within the design constraints.

4.7 Summary

This chapter discussed the kinematics of the Omnibot as well as several types of control modes including local and global control. A detailed description of the Omnibot’s electrical systems was given. The implementation of each system was presented. A detailed description of the velocity controller was shown. A global localization system was also discussed.
Chapter 5

Software and Programming

5.1 Overview

This chapter will discuss the selection and implementation of the open source robotics software and hardware and the programming of the Omnibot. The selected software is reviewed in detail. The benefits and challenges of using open source software are also discussed.

5.2 Microcontroller Programming

The microcontroller can be programmed in either C or C++ and this code is developed in Codewarrior [77]. Auto-generated code can be made using another application called Processor Expert [78]. Processor Expert serves as a vital tool in the programming of the Omnibot because it greatly simplifies the monitoring and control of various functions and parameters pertaining to the type of control the user wishes to use. It makes use of Embedded Beans which are batches of pre-written code that can be incorporated into a working microcontroller program. Each bean corresponds to a function that can be used on the microcontroller through the available interfaces. For example, a bean is selected to do the analog-to-digital conversion for the
joystick signals. This bean has variables that control the timing of the interrupt, the pin to connect to, the number of samples to take for averaging, etc. The bean also contains functions specific to analog-to-digital conversion such as a function that converts the analog signal to a byte. The code associated with these functions and variables is auto-generated. The user may select which functions to use and which not to compile.

One of the key problems with programming a microcontroller is the size of the code, so limiting the number of pre-compiled functions is one way to combat this problem. Codewarrior and Processor Expert are processor specific, so it will only load code and beans that are pertinent to the microcontroller that is being used.

### 5.3 Omnibot Software

Limitations in robotics are often caused by the merging of hardware components that use proprietary software. This becomes especially true for researchers that are trying to combine products that are sold from different manufacturers. These limitations can be avoided if software and hardware designers comply to standards or if they open their software for changes by the end user (open source).

Standards are extremely useful when it comes to use and replacement of components with varying systems. An example of a standard that is used today is the USB port. USB ports allow many different types of devices to be connected to a computer such as mice, keyboards, external hard drives, scanners, printers, digital cameras, among others. Some of these devices also use standard software drivers to run the equipment, meaning no installation is required. This greatly simplifies the user’s job of using the device. Having a standard is another way of saying that the equipment or software is modular, meaning that it can be easily added on to, replaced, maintained, and managed. Standards are also highly regulated by governing bodies, meaning
that hardware and software are highly tested before being considered compliant to a standard. This is especially important when safety is of concern. Software must be thoroughly tested before applied to robots, otherwise, dangerous situations can be created.

Although standards are a great way of allowing components to conform to robot programming needs, standards do not force the software to be non-proprietary. Customers can still be barred from optimal solutions because the software they purchased to operate the robot cannot be seen or changed. Open source software offers a route around this problem.

As was mentioned in Section 2.5, companies in general are beginning to shift to open source software development and away from the usual proprietary software development. Robots that cannot communicate with other components easily are undesirable as that creates unnecessary lead time in the development of a system. Also, the time to market for the companies themselves is longer because they are not using the knowledge available through an open source community. The biggest benefit from open source software is the ability to share ideas between the various groups in industry and research.

5.4 Remote Desktop and On-board Data Processor

Referring back to Figure 4.3, there are two systems of importance: the on-board data processor and the remote desktop. The on-board data processor serves as the highway for all systems to communicate, while the remote desktop is where off-board systems can interact with the Omnibot.

The remote desktop is a computer that will serve mainly as the user interface for controlling the Omnibot remotely. It will also serve as the access point for other
potential systems in the future, meaning if the Omnibot were to work in conjunction with other robots or systems, those systems would communicate through the remote desktop to the Omnibot. Data is made available to the remote desktop from the Omnibot through a wireless link.

The on-board data processor serves as the main hub for all data transmission on the Omnibot while allowing communication with off-board systems. The data processor is a laptop. A laptop was chosen because they are built to handle vibrations, are lightweight, and compact in design. Laptops also have their own power supply. The laptop interacts through USB and a wireless local area network (LAN) to communicate between the various systems. Since most of the on-board systems communicate only using serial, a USB to serial hub is used. It is desired to use the laptop not only as a data router for the various components, but also for future algorithms and applications as well.

5.5 Review and Comparison of Open Source Platforms

5.5.1 Requirements for Open Source Robotics Software

A review was done in Section 2.5 of the various available open source robotic platforms. In order to pick the one most suited for the Omnibot, a set of requirements should be used. These requirements should be similar to that of the requirements outlined in Section 3.4. However, some of the requirements do not apply as they only refer to the physical design of the Omnibot. The requirements that apply to the selection of the open source robotics software are as follows:

1. Modular.

   - The software should easily adapt to new applications while not hindering
existing applications.

2. Minimize time to implementation.
   - Installing the framework, learning how it works, and programming applications should be done as quickly as possible.

3. Real-time capabilities.
   - Since the software will be used to control a moving system, real-time data transfer is necessary.

4. Reliable.
   - Programs, once implemented, should not falter. Data transmission should work to specification, and data loss should be limited.

5. Robust.
   - Programming interface and installation should be intuitive and simple for the user and programmer.

6. Flexible.
   - Architecture should not be limited by operating systems or development patterns. It should also work with many applications such as robotic arms, mobile bases, sensors, etc.

7. Economical.

8. Open source.
   - Software should not have any restrictions in terms of editing, re-distribution, etc.

9. Re-usable code.
• Source code should be readily available and ready to use.

Comparing robotic software packages outlined in Section 2.5 to the requirements outlined above will help determine the best software for the application. A full analysis of each software package is not possible given the sheer complexity and volume of software to learn and test. However, the results can be drawn from other published articles and from the software websites.

5.5.2 Software Evaluation

OROCOS, ORCA, Player, CLARAty, MSRS, URBI, iRobot AWARE, Skilligent, and ROS (see Section 2.5 for acronym references) were all discussed in terms of capabilities. From this discussion, it was deduced that iRobot AWARE, Skilligent, and CLARAty are not free or open source, as some of the programming architecture are copyright protected by the various agencies they serve. Therefore, they are not candidates for this application.

MSRS, although a good possibility for future applications, is not practical for use in large research applications such as the Omnibot. It is not open source, and is limited to Microsoft Windows only, meaning the programs will never work in real-time.

Player has potential as a candidate, except it lacks the libraries necessary to drive robotic arms. The Omnibot will eventually be working in conjunction with a robotic arm and possibly other mobile manipulators. Therefore, Player lacks the flexibility required.

URBI is only partially open source, and the programmer must wait until the developers at URBI program drivers for a robot before it can be used.

ORCA lacks the ability to work in real-time. This is an important requirement of the software, so ORCA cannot be used.

The remaining software packages are OROCOS and ROS. Both meet the requirements of this application, even though they are limited to certain operating systems. In or-
der to maintain real-time capabilities, Microsoft Windows cannot be used because Windows architecture does not permit real-time operation of its software through its design. Despite this, ROS developers claim that they are working on an implementation with Windows in the near future. These two packages were downloaded and implemented for further comparison.

5.6 OROCOS Implementation

OROCOS is a framework that encompasses many applications. From a programmer’s perspective, it is an architecture that can be controlled at every level. It supports four different C++ libraries: the Real-Time Toolkit (RTT), Kinematics and Dynamics Library (KDL), Bayesian Filtering Library (BFL), and the Orocos Components Library (OCL). The KDL, BFL, and RTT libraries all serve as functions for the OCL, which consists of many components for various applications. The RTT provides functions to control components in real-time. The KDL has functions that compute kinematic chains for robots. The BFL has various filtering algorithms commonly used in robotic applications.

Many programs can be designed using the modules, called TaskContexts, in the OCL as a guideline. The base function of a TaskContext could be to control an entire robot, or many TaskContexts could be used to make up an entire system, say a series of various hardware and software components to control a navigation system. TaskContexts communicate to each other through software ports, which allow for lock-free and thread-safe data exchange. This means each component is de-coupled, and if one suddenly de-activates, function will not completely stop across the other modules. The modules can be triggered by event or periodically. The TaskContexts and their properties can be monitored and changed at any time during operation from a terminal. This makes it easy to troubleshoot problems as they arise as well as plan
for system failures that cannot be simulated through the hardware.

OROCOS was installed with a computer running the Debian Etch [79] version of Linux as its operating system. Linux itself is an open source project, and fits nicely into the requirements of this project. Implementation was only done as far as running some of the sample TaskContexts from OCL.

5.6.1 Problems Encountered

In general, the installation of OROCOS was difficult. A lot of knowledge is required of the general Linux operating system. There are many variations to the installation instructions due to the various Linux projects that are available. Within those various versions of Linux are also various kernel versions. Amongst the kernel versions are various drivers that were only recently created or recently made obsolete, meaning that some software packages are non-backwards compatible. OROCOS also requires many other prerequisites in the form of programs. Not all prerequisites are mandatory for all parts of OROCOS, but some of the programs are needed for smaller applications inside the libraries that come with OROCOS. Also, some of the libraries are still awaiting update in OROCOS, making them non-functional with some other programs that have been upgraded.

This problem exists because the entire open source framework is based strictly on a volunteer basis, meaning that programs must be updated voluntarily and often without any form of compensation. It is a continuous line of upgrades that can only end with the definition of solid standards that prevent the necessary functions for backward compatibility from becoming obsolete until sufficient upgrades can be made for all parties involved. Another way around this problem is if the software was managed by people who were paid to keep it up-to-date and compatible with current and previous versions of operating systems and their prerequisites.
5.7 ROS Implementation

ROS is unique amongst all the other software packages presented in that it is managed by a host of paid employees who are designated to specific positions in bringing the software to fruition, but the software is still free and open source like the user-managed packages such as OROCOS. The employees at Willow Garage who design ROS are people who have a demonstrated drive in robotics and a third of the team are respected leaders in robotics research. There funding comes from generous private investors and acts of philanthropy. [80]

The primary focus of Willow Garage is to lay the groundwork for the industry that will invest in personal robotics. Their philosophy on this dictates active development in the community, so their software is readily available during all stages of development. Willow Garage is still a new company, hence it has not released a full version of ROS. However, the framework and many sample applications are readily available and have been implemented for use on the Omnibot.

It should also be mentioned that ROS already works on a number of operating systems including Linux and MAC OS X. Also, ROS has real-time capabilities just like OROCOS, and meets the control requirements of the Omnibot. An added benefit with ROS over OROCOS is of safety consideration. With user-only supported software, it is unlikely that the base implementation of components will enforce protocols like safety. However, if the software is regulated by a governing body, safety is considered along with other protocols.

ROS consists of components called nodes. These nodes communicate between each other by passing messages in the forms of data structures. Nodes can publish messages in which other nodes can subscribe. Nodes can be both publishers and subscribers at the same time. The subscribing nodes receive these messages without the publishing node knowing whether the messages were received, or even if it is being subscribed to. This is similar to OROCOS in that the nodes that communicate with each other are
de-coupled and will not stop working if one node shuts down prematurely because of an error or user interference. Each node communicates with a master node in order to communicate its position and characteristics, and to subscribe to and/or publish data. The master node’s job is to keep track of all subscribers and publishers and broadcast location information of publisher nodes as they become available to subscriber nodes requesting that information. The subscriber nodes then use the information from the master node to directly connect to the corresponding publisher node for direct message transmission.

Figure 5.1 shows a simple example with one subscribing node and one publishing node interacting with the master node. The publisher node connects to the master node and advertises the name of its published topic (in this case: ‘scan’) and the port where the XML-RPC server that corresponds to this topic is broadcasting (‘foo:1234’). The master then looks for the subscriber node or nodes that are requesting this topic. The subscriber node is requesting ‘scan’, so it is sent the corresponding address of the XML/RPC server. The subscriber node then attempts to connect to the published
topic by first connecting with the XML/RPC server and getting another address for the specific data messages. Upon receiving this address (‘foo:2345’), a connection is made and direct subscriber to publisher communication is established. It should be noted that the master node only serves as a negotiator and does not handle the message data itself. Also, all location information data is sent and received through XML/RPC servers.

ROS was installed on both the remote desktop and onboard data processor with ease. Although there were prerequisites, commands were given in the installation instruction to upgrade the software to the most recent editions of the software required by ROS. Some programs that work inside the ROS framework require outside software to be installed. ROS has built-in commands to quickly identify and install these programs making the entire installation experience simple.

A simple remote operation was implemented on the Omnibot. The goals of this setup are to send velocity commands from a user interface on the remote desktop to the onboard data processor through a wireless link. The velocity commands are then delivered to the onboard microcontroller through a serial connection where the commands are processed and the desired velocity is tracked. This hardware setup can be seen in Figure 5.2. In addition, the corresponding motor velocities are transmitted back to the remote desktop for viewing.

Three nodes were designed for this implementation which are the OmnibotUI, OmnibotPinger, and OmnibotPonger nodes (see Appendix D). The OmnibotUI and
OmnibotPinger nodes run on the remote desktop while the OmnibotPonger node runs on the onboard data processor. The OmnibotUI node interprets the data from the user interface (joystick) and publishes the commands. The OmnibotPinger node subscribes to the OmnibotUI data and publishes the data at a rate designated by the transfer rate between the onboard data processor and remote desktop. The OmnibotPonger node on the onboard data processor subscribes to these commands and then sends the commands through the serial link to the microcontroller. Since the laptop only has USB ports, a USB-to-serial link is used. The node also receives the velocity data from the microcontroller and publishes the data. This velocity data is then received by the OmnibotPinger node on the remote desktop. The command velocities and actual velocities are displayed to the user by the OmnibotPinger node. This implementation can be seen in Figure 5.3.

It was decided to have two separate nodes on the remote desktop to allow for implementation of other nodes that could publish the topic ‘JoyChatter’. The current node, OmnibotUI, is capable of receiving commands from a joystick through a serial port and publishing the data in a string. In the future, it may be decided to have a node capable of commanding the Omnibot from the keyboard or perhaps a more automated process such as an off-board navigation system. Node swapping allows for the programmer to easily swap code as required based on topics alone.
The OmnibotPinger and OmnibotPonger nodes wait on each other before sending data. Since they each publish and subscribe to one another, it is possible to make one of the nodes wait while the other processes and sends data to it through a wireless TCP connection. This is how these nodes communicate their messages with each other.

The master node was loaded on the remote desktop. In general, it does not matter where the master node is loaded as long as a master node is loaded and that all other ROS computers know where the master node is located. If the master node fails, all other nodes will also stop communicating. Therefore, it was beneficial to put the master node on the most reliable computer. The remote desktop is the most reliable in this system because it does not rely on battery power.

Serial communication occurs in two instances in this setup: between the joystick and OmnibotUI node and between the microcontroller and the OmnibotPonger node. In each instance, a coupled connection is made. If the node was activated but did not successfully link with the microcontroller or joystick, the node would fail to launch. Once a connection is established, messages are sent back and forth as fast as possible. The microcontroller is programmed to handle errors such as communication breaks or lost messages from the serial connection. If data is not received within a certain length of time, the controller will command the motors to stop moving until such a time when serial communication is re-established.

5.8 Summary

This chapter reviewed the selection and implementation of open source software on the Omnibot. The requirements for the software of the Omnibot were discussed. Ultimately, Willow Garage’s Robotic Operating System (ROS) was selected for its unique stance in industry and research as an open source robotic architecture. ROS
was implemented using a remote desktop and onboard data processor. OROCOS was also evaluated for use with the Omnibot, but installation issues prevented full implementation.
Chapter 6

Test Results and Discussion

6.1 Overview

This chapter discusses the results of testing the mechanical design, velocity controller, indoor localization system, and open source system. Each section reflects on the discussion of implementation in the previous chapters while describing implementation issues and specifics.

6.2 Mechanical Test Results

The Omnibot was manufactured and assembled as described in Chapter 3. Since then, the Omnibot has undergone extensive operation, mostly in lab environments, but also on surfaces that were carpeted as well as rough surfaces like asphalt and cement. In terms of hours of operation, it has been used in many demonstrations and used extensively in testing. The Omnibot has easily surpassed 100 hours of operation. Through operation, failures have only been witnessed in the transmission. The frame, suspension, electrical mount, and motors have all shown no sign of failure to date. The suspension failed in two places. The first was in a U-joint and the second was the sheering of a number of pins that hold the transmission to the motor. Both of
Table 6.1: Spring compression test

<table>
<thead>
<tr>
<th>Spring #</th>
<th>Compression length (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>16.0</td>
</tr>
<tr>
<td>1B</td>
<td>18.2</td>
</tr>
<tr>
<td>1C</td>
<td>16.6</td>
</tr>
<tr>
<td>1D</td>
<td>17.8</td>
</tr>
<tr>
<td>2A</td>
<td>19.3</td>
</tr>
<tr>
<td>2B</td>
<td>17.2</td>
</tr>
<tr>
<td>2C</td>
<td>16.4</td>
</tr>
<tr>
<td>2D</td>
<td>16.3</td>
</tr>
<tr>
<td>3A</td>
<td>16.1</td>
</tr>
<tr>
<td>3B</td>
<td>16.3</td>
</tr>
<tr>
<td>3C</td>
<td>13.3</td>
</tr>
<tr>
<td>3D</td>
<td>13.8</td>
</tr>
<tr>
<td>4A</td>
<td>16.2</td>
</tr>
<tr>
<td>4B</td>
<td>16.6</td>
</tr>
<tr>
<td>4C</td>
<td>17.7</td>
</tr>
<tr>
<td>4D</td>
<td>16.5</td>
</tr>
</tbody>
</table>

these failures were caused by over-loading through excess torque. This large torque loading was experienced when the wheel velocity was changed suddenly from full speed forward to full speed backward and vice-versa quickly. This problem was corrected by adding a small algorithm to the program to average out commanded velocities over a period of time and allowing large adjustments of velocity to happen over an extended period of time only, as opposed to all at once. The pin failure is also attributed to the fact the pin was responsible for delivering the entire torque load to the wheel from the motor. This problem was alleviated by making use of the slotted section of the motor shaft by inserting a wire into it, and then wedging the motor shaft and wire into the socket head. The pin is then inserted through the wire and existing hole in the motor shaft. The wire takes the brunt of the torque with friction while the pin now serves only as a locking mechanism. Since the implementation of these two solutions, no failures have been experienced in the transmission.

The Omnibot frame and suspension underwent a loading test using the 53 kg Thermo
Figure 6.1: Compression test using 53 kg Thermo F3 Robot Arm

F3 robot arm. The installation length of each spring was measured and recorded. The robot arm was then placed on the center of the Omnibot and the operating length of each spring was measured. It was noted that the installation length of each spring varied depending on the orientation of the wheels and the unequal loading caused by existing on-board systems. Table 6.1 contains the length of the compression on the spring. It should be noted that none of the springs individually compressed further than 30 mm, which was the prescribed maximum compression length of the springs. As the robot arm moves to interact with its environment, the center of mass will shift causing the distribution across the 16 springs to change. It is expected that the last 10-15 mm of available compression will withstand the fluxuation in weight distribution. Variations between lengths in this experiment are caused by the shifting of weight between each side of the wheel, along with the unequal loading caused by a non-balanced platform.
Apart from measuring spring compression, a visual inspection was performed on the frame of the Omnibot. There was no visible strain or warping caused by the weight of the robot arm. Therefore, it was concluded that the mechanical design of the Omnibot will withstand the weight of the robot arm. Several pictures from this experiment can be seen in Figure 6.1.

In general, the Omnibot suspension performed as desired. While in operation, the omni-wheels were viewed to determine if the wheels were losing contact with the floor. The only instance of slippage that was detected originated from sudden changes in velocity. It was concluded that each wheel was indeed remaining in contact.

A test was performed to see if the Omnibot could handle driving over small bumps in the workspace without hindrance. For the most part, the Omnibot traveled seamlessly over a fixed bump as seen in Figure 6.2. It was noted that an omni-wheel would sometimes get stuck when rolling purely on the omni-wheel’s smaller secondary axis. The barrel radius of the secondary axis is much smaller than the primary axis radius which implies that more force is required to roll over small bumps. Rolling diagonally and rolling using purely the primary wheel axis enabled the Omnibot to get over the bump without interrupting its motion. It was also noted that the nuts at the bottom of the suspension system would sometimes scrape the bump.
6.3 Velocity Controller Test Results

The velocity control system was implemented as described in Chapter 4. Before velocity control was possible, hardware connections had to be established and tested independently. One of the most challenging parts of the PID implementation was the setup and installation of the encoder counters. Each encoder counter is interconnected with other encoder counters, which made troubleshooting for the problematic encoder counter difficult. Through extensive debugging and testing, it was discovered that some of the original connections on the PCB had come loose and needed repair, the wiring was initially too thin and was not staying in place during operation, and the PB6 and PB7 pins were reversed inside the microcontroller itself. Once the wiring was re-done with thicker gage wire and the PCB repaired, the encoder counters worked, and a preliminary test could be run to see if the encoder counts matched the actual angular velocity of the wheel.

6.3.1 Encoder Tests

The encoder test was carried out by first marking a wheel to indicate a full rotation that can be seen by the examiner. The motor was set to rotate at a constant angular velocity, and the program was set to print the current motor velocity to a computer. The number of rotations was counted for 30 seconds and an average of the angular motor velocity was taken to compare to the actual motor velocity. The average was necessary because the measured motor velocity was always changing slightly, most likely due to variance in friction contact as the wheel spun. The actual motor velocity was determined as:

\[ v_{\text{actual}} = \frac{Rot}{t} 2\pi \]  

(6.1)

where \( v_{\text{actual}} \) is the actual angular velocity measured in rad/s, \( Rot \) is the number of
Table 6.2: Encoder counter test

<table>
<thead>
<tr>
<th>Test #</th>
<th>Average measured angular velocity (rad/s)</th>
<th>Counted rotations</th>
<th>Actual angular velocity (rad/s)</th>
<th>error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.615</td>
<td>22</td>
<td>4.608</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>4.317</td>
<td>20.5</td>
<td>4.293</td>
<td>0.55</td>
</tr>
<tr>
<td>3</td>
<td>4.451</td>
<td>21</td>
<td>4.398</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>4.462</td>
<td>21.5</td>
<td>4.503</td>
<td>0.91</td>
</tr>
</tbody>
</table>

rotations counted over a 30 second period, and $t$ is the 30 seconds that it took to count the rotations.

Table 6.2 compares the measured velocity with the actual velocity as counted by the examiner. It is noted that the error is below 2% for each of the tests. The error in this experiment was likely caused by human error, since it is not easy to determine the exact position of the wheel after 30 seconds of rotation. Since error is relatively minimal, it was assumed that the velocity of the wheel is accurately reflected by the measured angular velocity using the encoders.

6.3.2 PID Tuning

In order for the motors to be controlled adequately, the PID gain constants had to be tuned. A program was implemented on the Omnibot’s microcontroller to set the command velocity as a square-wave signal (on-off). This input velocity was then processed through the PID control and the output command was sent to the motors. Both the input velocity and measured velocity were sent out using a serial connection to a remote desktop. MATLAB [82] was programmed to receive the serial signals and process them into graphs for analysis.

The PID gain constants were, at first, tuned using the ultimate cycle method by Ziegler and Nichols [83]. The ultimate cycle method tunes the three PID gain constants by finding the critical value of the proportional gain constant $K_{PC}$ and the respective periodic time of oscillations $T_c$. The critical value is found by slowly in-
Figure 6.3: Graphs depicting effect of changes to proportional gain constant

\[ K_p = 0 \]

\[ K_p = 1.00 \]

\[ K_p = 2.00 \]

\[ K_p = 3.00 \]

\[ K_p = 3.25 \]

\[ K_p = 3.50 \]

Figure 6.3: Graphs depicting effect of changes to proportional gain constant
creasing the gain until a constant stable oscillation is seen. Figure 6.3 has six graphs showing the effect on the measured velocity when the proportional gain is changed. The critical value of the proportional gain was found to be 3.50 as it gave a constant stable oscillation. The oscillation time from peak-to-peak during the stable oscillation was measured to be 25.0 ms. The ultimate cycle criteria was then applied as:

\[ K_P = 0.6K_{pc} = 2.10 \]
\[ T_I = T_c/2 = 15 \text{ ms} \]  \hspace{1cm} (6.2)
\[ T_D = T_c/8 = 3.75 \text{ ms} \]

where \( T_I \) represents the integral time constant and \( T_D \) represents the derivative time constant.

The PID gain constants were then determined as:

\[ K_P = 2.10 \]
\[ K_I = \frac{K_P}{T_I} = \frac{2.10}{15} = 0.14/\text{ms} \]
\[ K_D = K_P \cdot T_D = 2.10 \cdot 3.75 = 7.875 \text{ ms} \]  \hspace{1cm} (6.3)

The PID gain constants resulted in the graph shown in Figure 6.4. Clearly, the aggressive nature of the Ziegler and Nichols method lead to an unstable solution that caused the motor to oscillate.

It was clear the Ziegler and Nichols method did not work, so manual tuning was used. This method is similar to the Ziegler and Nichols method in that it starts with finding the ultimate gain. Upon finding the ultimate gain, \( K_P \) was set to half and \( K_I \) was increased until the offset was corrected within a sufficient amount of time. \( K_D \) was increased until the settling time was acceptably short and the overshoot had shrunk sufficiently. This method was carried out and acceptable gain constants were found as \( K_P = 2.40 \), \( K_D = 1.09 \), and \( K_I = 0.02 \). These gains were selected as
they offered minimum overshoot, steady state error, and time to stability, while still reacting quickly to changes. The graph in Figure 6.5 depicts the effect of a step input with these gains applied. Appendix C has the graphs showing the various steps in this tuning method.

Although the wheel was unloaded, error was present in this experiment through the friction in the transmission and suspension system as well as the slack in the transmission that allowed for a small amount of free rotation in the wheel. Also, a linear relation was assumed for the conversion from motor command to motor speed, which is not the case. This increases steady state error and overshoot because the actual velocity and commanded velocity are not the same.

### 6.3.3 Motor Command to Motor Speed Mapping

The motors are controlled by specifying the duty cycle of a PWM signal. The duty cycle can be specified as a percentage but in the program, its specified as a number between 0 and 255. A program was made to slowly step through each integer between 0 and 255 and record the resulting angular velocity. The graph in Figure 6.6 was made using these gains.
It was noted that the relationship between the commanded and actual velocity is non-linear. Although the PID controller does a good job of correcting for this non-linearity, the control would be more accurate if the non-linear relationship was taken into account. The PID controller would also arrive at the desired velocity faster and with less overshoot. Several attempts were made to map the actual velocity to the commanded velocity.

Using cftool, a Matlab toolbox, an equation was formed that matches the positive half of the graph depicted in Figure 6.6. The data from the reverse direction of the graph was made positive to match the plot on the positive end. Although there is some discrepancy between the positive and negative end of the plot, the difference is negligible. Also, the axes were flipped, as the conversion would happen between the PID controller, which is in rad/s, and the motor output, which is a 8-bit number. The resulting equation was:

\[ y = 0.1654e^{0.5154x} + 6.731x \]  

This equation can be seen plotted over the data in Figure 6.7. In order to use this
Figure 6.6: Graph mapping commanded velocity to measured velocity

equation, a 32-bit double variable had to be created in the program. Adding this one calculation slowed down the processing cycle of the program from 7.5 ms to 25 ms. This slowed down the PID controller to an unacceptable rate, which is contradictory of the purpose for mapping this equation in the first place. Since the PID controller was already countering the non-linearity, it was decided to leave the mapping out of the microcontroller program.

In addition to this equation, other equations were tested that lead to similar results. A third and six order polynomial were tested, as well as a look-up table. While the third order polynomial caused unnecessary stepping in the output, the look-up table caused the processor to slow down significantly. The six-order polynomial caused stepping and slowed down the processor. A good solution for mapping the non-linearity that minimizes the impact on the microcontroller processing speed has was not found.

### 6.3.4 Visual PID Control Test

A test was constructed to show the ability of the Omnibot to travel in a straight path. Straight path travel would not be possible if the motor velocities did not match
the commanded velocities. The Omnibot was positioned beside a straight line and instructed to travel at a constant velocity horizontally using only two motors. The motion of the Omnibot was videotaped so the displacement from the line could be viewed as the Omnibot traveled. The Omnibot was also instructed to travel diagonally using four motors. This motion was also recorded. It was viewed that the Omnibot was able to travel 3 m without noticeably drifting from the trajectory. Still images of these motions can be seen in Figure 6.8.

It was difficult to set the initial angle of travel on the Omnibot due to slack in the transmission. If one wheel started spinning before the others, the angle of trajectory would change initially causing the overall run to be discarded.
Figure 6.8: Several images from the visual PID control test
6.4 Cricket Test Results

Preliminary implementation of the Cricket system was conducted to determine the position of the Omnibot through post-processing algorithms. Four listeners were mounted on the Omnibot as shown in Figure 4.14. Each Cricket listener was connected via serial cable to a PC that recorded the position in terms of universal time. The demo program that came with the Crickets was modified for this purpose. After the beacons had been calibrated, the PCs were commanded to record the position data communicated from the listeners along with the time that it was received. The Omnibot was moved inside the workspace to varying positions. Since the listeners were not communicating with each other, it was unlikely they would determine their location at the same time.

Through the use of MATLAB, the data was analyzed and optimized using an optimization algorithm. This algorithm consisted of five steps: find relevant coordinates, pre-optimization filter, optimization, post-optimization filter, and data-imaging.

To find the relevant co-ordinates, the data had to be compiled in the program and mapped by their time of recording.

The pre-optimization filter determined and selected data points that were the most recently given according to the current analysis time. Four Crickets were used, but the algorithm only needed three to determine the 3-D position and orientation of the Omnibot. So long as three of the four listeners reported position within a certain time, the point was considered worth analyzing. This selection process is shown in Figure 6.9. The X’s represent a data point at a certain time, and the circles represent the selected data point given the current time. Note that since Cricket #3 did not report data within the tolerance time, no data was considered. The pre-optimization step also eliminated some data points if they were not a feasible distance between the last recorded point and the current point. In order to save processing time, the selected data points were checked against the previous chosen data points, if they
The optimization step was skipped.

The optimization step used a constrained optimization algorithm to find the best fit of points given the actual distance between the Crickets and the data that was collected during the previous steps. Figure 6.10 demonstrates the shifting of the constrained frame so that it matches as closely as possible the points given by the listeners (the square boxes) while still holding the constrained distances. This optimization can

Figure 6.9: Graph showing relevant data from four Crickets versus time

Figure 6.10: Sketches of the optimization algorithm results for one pose
be mathematically described through an objective function and constraints. The objective function is as follows:

\[
f_i(x) = \sqrt{(x_{Ei} - x_{Ai})^2 + (y_{Ei} - y_{Ai})^2}
\]  

(6.5)

where \(x_{Ei}\) and \(y_{Ei}\) are the coordinates of the estimated node position and \(x_{Ai}\) and \(y_{Ai}\) are the coordinates of the actual node position as defined by the listeners for nodes \(i = 1\) to \(4\). The vector \(x\) contains all of the \(x\) and \(y\) co-ordinates of the estimated nodes. Levenberg-Marquardt optimization [84] was used to optimize all four equations simultaneously. This method minimizes all four functions at once using the assembled vector \(F(x)\):

\[
F(x) = \{f_1(x), f_2(x), f_3(x), f_4(x)\}^T
\]  

(6.6)

This algorithm works by finding an optimal \(x\) such that the sum of the squares of the elements is minimized to \(F(x) = 0\).

The frame shown in Figure 6.11 is a simple sketch of the four listeners on the Omnibot. It is assumed that the link lengths of all four sides \((d_{12}, d_{23}, d_{34}, d_{41})\) are the same length. Likewise, the two diagonals \((d_{13}, d_{24})\) are also assumed to be the same length.
$N_{Ei}$ for $i = 1$ to $4$ symbolize the estimated position of each listener. Each node in the estimated frame must be linked together by the use of constraints. Each of these constraints serve to hold the estimated nodes to the actual measurements between the Crickets. The following constraints were applied:

\[
l_1 = d_{12} = d_{23} = d_{34} = d_{41}
\]

\[
l_2 = d_{13} = d_{24}
\]

\[
d_{ij} = \sqrt{(x_{Ei} - x_{Ej})^2 + (y_{Ei} - y_{Ej})^2}
\]

where $l_1$ and $l_2$ are specified physical dimensions from the listener placement and $d_{ij}$ is the specified distance between nodes $i$ and $j$.

Since the square root is taken in all the magnitude calculations, their will always be two solutions, one of which will be impossible. Therefore, other constraints were also implemented to ensure the location of the four corners of the node frame were clockwise. The following constraints were formed for this purpose:

\[
x_{A2} \geq x_{A1} \quad \rightarrow \quad y_{A3} - y_{A2} \leq 0
\]

\[
x_{A2} < x_{A1} \quad \rightarrow \quad y_{A2} - y_{A3} \leq 0
\]

\[
x_{A3} \geq x_{A2} \quad \rightarrow \quad y_{A4} - y_{A3} \leq 0
\]

\[
x_{A3} < x_{A2} \quad \rightarrow \quad y_{A3} - y_{A4} \leq 0
\]

These constraints only become active when their condition is true. This means that only two of the four constraints will be active at any one time.

The optimization uses these equations to best place the constrained frame as close to the Cricket data as possible. The optimization was programmed to handle a four node optimization and a three node optimization. If only three Cricket location data packets were available, then 3-point optimization was used. Figure 6.12 has examples...
Post-optimization verified that the optimization calculated a feasible solution and that the optimization algorithm did not output any faults. Once the optimization was completed, images were created using MATLAB’s graphing capabilities. An image from this process is shown in Figure 6.13. The image shows the path of the Omnibot as it translates and rotates simultaneously.

It was concluded that the data is easily collected and points generated when the Omnibot is stationary. However, when the Omnibot is in motion, very few feasible solutions are generated. This is likely due to the passive nature of the Crickets, which causes location calculations to be slower. A listener must interpret signals from beacons at separate times before determining its own position. By the time the third beacon measurement has been made, the other beacon data is too old.

### 6.5 Open Source Programming Test Results

The open source program was implemented as described in Chapter 5. A picture of the Omnibot during remote operation can be seen in Figure 6.14. The goal to drive the Omnibot remotely was achieved using the program shown in Figure 5.3. Since open source code was readily available, code was taken and altered for this application from the ROS libraries. The nodes from the pingpong.cpp tutorial program, pinger
and ponger, were used in the construction of OmnibotPinger and OmnibotPonger, respectively. Also, the talker and listener nodes from roscpp_tutorial were used as well. Talker was used in the construction of the OmnibotUI and parts of listener were used in OmnibotPinger. Since these ROS nodes were built using C++, other C++ libraries and headers were added to use the commands that activate, read from, and write to serial ports. All programs can be found in Appendix D.

It was noted that the environment variables ROS_IP and ROS_MASTER_URI must be updated to reflect the location of the master node and the current IP address of the machine the ROS node will be operating from. If these are not set exactly, node communication is not possible. Upon restart of either CPU, the IP address could change, and the environment variables would no longer be valid.

It was difficult to implement communication between the microcontroller and onboard data processor through serial communication. Since the program on the microcontroller runs an interrupt every time a character is received from the serial port, the interrupt function runs quite often. Also, data was being sent out at a regular interval.
within the program. If this write operation ran too fast, the program would act un-
predictably, and data from the output would appear in the input. Through testing, it
was determined that only small amounts of data could be sent every 15 milliseconds.
Any faster and the microcontroller did not function properly.

In terms of response rate, no noticeable difference was seen between the remote driving
operation and direct drive operation. Also, data was successfully passed back from
the microcontroller showing the velocities of the four motors. This proves that nodes
can be configured for data transmission in later applications.

6.6 Discussion

Over all, the implementation of the Omnibot was a success. The physical prototype
was built and tested. Mechanical testing has shown the effectiveness of meeting
the requirements of the Omnibot. This includes the Omnibot’s ability to support
the weight of a 53 kg robotic arm and the ability to handle variations in the floor.
Initially, the Omnibot was having transmission failures, but the prescribed corrective measures have eliminated these failures.

The implementation and testing of the velocity controller proved the Omnibot could move in a controlled fashion and will respond well to future autonomous applications. The Omnibot successfully used all four motors and encoders to translate at prescribed velocities for local control. Testing revealed that PID control across the four motors can achieve straight line motion. The motor amplifiers allowed for a full range of velocity motion which allows for any motion possible.

A preliminary implementation was done to check the feasibility of using a global localization system to track the pose of the Omnibot. An optimization algorithm was designed and executed for post-processing of pertinent position data to determine the Omnibot’s pose. Results have shown that the passive architecture of the localization system was too slow to give adequate feedback when the Omnibot was moving.

An open source robotics software was selected and implemented on the Omnibot for remote communication. It was shown that it is possible to drive the Omnibot remotely and receive pertinent data to the remote machine about the Omnibot’s operation. ROS has proven to be a viable open source software and will be used in future applications for inter-process communication.

Through implementation and testing, the objectives of this thesis were met. All of this testing verifies the development of a functioning ODV platform that will form the basis for a mobile manipulator system and autonomous system.
Chapter 7

Conclusions and Recommendations for Future Work

7.1 Conclusions

The Omnibot was presented in terms of mechanical, electrical, and software design and development. The Omnibot uses a unique omni-directional configuration to achieve holonomic motion while offering a lower center of gravity compared to traditional omni-directional platforms. Its modular frame was built to support a robotic arm as well as many other systems while maintaining stability as it travels. A suspension and transmission system were designed to ensure each wheel stays in contact with the surface while reducing vibrations that occur from rotation of the omni-wheels. Kinematic representations of the Omnibot were developed to allow the Omnibot to move in both local and global control modes. The Omnibot is capable of traveling accurately at specified velocities through the use of four independent PID controllers that correct the velocity of each wheel using encoder feedback. Accurate local control was implemented through the use of PID control. The first step towards global control was also taken through the experimentation with an indoor localization system known
as Crickets. This system was proven to be insufficient given the passive architecture of the Crickets because the data was not updated fast enough for accurate tracking when the Omnibot was in motion. The implementation of driving the Omnibot remotely was proven through the use of an open source robotics software package. Willow Garage’s Robot Operating System (ROS) was selected for development of the remote drive system as well as future implementation of other systems such as the indoor localization system. Overall, all requirements of this thesis were met. The Omnibot Williams serve as the omni-directional platform for hosting autonomous applications and mobile manipulator applications.

7.2 Future Work

The Omnibot is an ongoing project in the Mechatronic and Robotics Systems Laboratory at the University of Ontario Institute of Technology. As such, there are many systems that require implementation before the system can be declared as an operating autonomous robot and/or mobile manipulator. The new systems that were discussed but not implemented in this thesis should be installed and tested. These systems are the collision detection systems, vision system, and an improved indoor global localization system. Although the indoor global localization system was partially implemented, there is much work to be done on increasing accuracy and sampling rates if it is to be used as a form of feedback for driving the Omnibot. Once this system is operational, other more abstract algorithms can be written for autonomous control such as path planning. These systems must also be linked using ROS so data can be easily transferred between systems. It is believed that ROS has great potential, and implementation will become simpler as more tested programs become available. In general, the global control algorithms need to be implemented in order for autonomous travel to be possible.
References


[2] R. Park, “Flight simulation technology is revolutionizing automo-
tive testing,” See also URL www.moog.com/industrial/newsletter


[5] California Institute of Technology NASA Jet Propulsion Labora-
tory, “Mars exploration rover mission,” 2009, See also URL
http://marsrovers.nasa.gov/overview/.

an rov using base force information,” in Proceedings of the IEEE International


Appendix A

Electronic Component Data Sheets

A.1 MicroMo Motor

The following datasheet is available from MicroMo [65]. This datasheet describes several gearheads available from MicroMo. The particular gearhead selected for the Omnibot has the following attributes:

- Series 3257 012 CR
- 12V motor
- 531 mNm Stall torque
- 5,700 rpm no-load speed
### DC-Micromotors

**Graphite Commutation**

For combination with (overview on page 14-15)

Gearheads: 32/3, 36/1, 38/2

Encoders: IE2 – 16 ... 512, 5500, 5540

#### Series 3257 ... CR

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal voltage</td>
<td>U₉</td>
</tr>
<tr>
<td>Terminal resistance</td>
<td>Rₙ</td>
</tr>
<tr>
<td>Output power</td>
<td>P一站</td>
</tr>
<tr>
<td>Efficiency</td>
<td>η一站</td>
</tr>
<tr>
<td>No-load speed</td>
<td>nₙ</td>
</tr>
<tr>
<td>Stall torque</td>
<td>M₂</td>
</tr>
<tr>
<td>Friction torque</td>
<td>Mₚ</td>
</tr>
<tr>
<td>Speed constant</td>
<td>kₑ</td>
</tr>
<tr>
<td>Back-EMF constant</td>
<td>kₑ</td>
</tr>
<tr>
<td>Current constant</td>
<td>kₑ</td>
</tr>
<tr>
<td>Slope of n-M curve</td>
<td>δn/δM</td>
</tr>
<tr>
<td>Rotor inductance</td>
<td>Lₑ</td>
</tr>
<tr>
<td>Mechanical time constant</td>
<td>τₑ</td>
</tr>
<tr>
<td>Rotor inertia</td>
<td>J</td>
</tr>
<tr>
<td>Angular acceleration</td>
<td>αₘₚ</td>
</tr>
<tr>
<td>Thermal resistance</td>
<td>Rₐk / Rₑk</td>
</tr>
<tr>
<td>Thermal time constant</td>
<td>τₑk / τₑk</td>
</tr>
<tr>
<td>Shaft bearings</td>
<td>ball bearings, preloaded</td>
</tr>
<tr>
<td>Shaft load max.:</td>
<td></td>
</tr>
<tr>
<td>– with shaft diameter</td>
<td>5,0</td>
</tr>
<tr>
<td>– radial at 3 000 rpm (3 mm from bearing)</td>
<td>50</td>
</tr>
<tr>
<td>– axial at 3 000 rpm</td>
<td>5</td>
</tr>
<tr>
<td>– axial at standstill</td>
<td>50</td>
</tr>
<tr>
<td>Shaft play:</td>
<td></td>
</tr>
<tr>
<td>– radial</td>
<td>ε</td>
</tr>
<tr>
<td>– axial</td>
<td>δ</td>
</tr>
<tr>
<td>Housing material</td>
<td>steel, black coated</td>
</tr>
<tr>
<td>Weight</td>
<td>242</td>
</tr>
<tr>
<td>Direction of rotation</td>
<td>clockwise, viewed from the front face</td>
</tr>
</tbody>
</table>

#### Recommended values - mathematically independent of each other

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed up to</td>
<td>ωₑmax</td>
</tr>
<tr>
<td>Torque up to</td>
<td>Mₑmax</td>
</tr>
<tr>
<td>Current up to (thermal limits)</td>
<td>Iₑmax</td>
</tr>
</tbody>
</table>

**Graphite Commutation**

For details on technical information and lifetime performance refer to pages 26-32. Specifications subject to change without notice.

Edition 2008-2009

www.faulhaber.com

For options on DC-Micromotors refer to page 62.
A.2 MicroMo Gearhead

The following datasheet is available from MicroMo [65]. This datasheet describes several gearheads available from MicroMo. The particular gearhead selected for the Omnibot has the following attributes:

- 45:1 reduction ratio
- 20 Nm Output Torque
- 90% efficiency
- Sizing pertains to DC-Micromotor #3257
Planetary Gearheads

Series 38A

Housing material: steel
Geartrain material: steel

Recommended max. input speed for:
- continuous operation: 4 500 rpm

Backlash, at no-load:
- radial (10 mm from mounting face): ≤ 0,75°
- axial: ≤ 230 N
- radial (5 mm from mounting face): ≤ 0,03 mm
- axial: ≤ 0,3 mm

Shaft press fit force, max.:
- radial (5 mm from mounting face): ≤ 0,03 mm
- axial: ≤ 0,3 mm

Operating temperature range:
- -25°C ... +90°C

<table>
<thead>
<tr>
<th>Specifications</th>
<th>reduction ratio</th>
<th>backlash</th>
<th>weight</th>
<th>length without motor</th>
<th>length with motor</th>
<th>output torque</th>
<th>direction of rotation</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(absolute)</td>
<td>°</td>
<td>g</td>
<td>L2 (mm)</td>
<td>L1 (mm)</td>
<td>Mmotor (Nm)</td>
<td>(reversible)</td>
<td>%</td>
</tr>
<tr>
<td>4:1</td>
<td>0,50</td>
<td>190</td>
<td>42,2</td>
<td>93,8</td>
<td>99,2</td>
<td>106,2</td>
<td>113,6</td>
<td>139,6</td>
</tr>
<tr>
<td>5:1</td>
<td>0,50</td>
<td>190</td>
<td>42,2</td>
<td>93,8</td>
<td>99,2</td>
<td>106,2</td>
<td>113,6</td>
<td>139,6</td>
</tr>
<tr>
<td>12:1</td>
<td>0,58</td>
<td>260</td>
<td>55,0</td>
<td>106,6</td>
<td>112,0</td>
<td>119,0</td>
<td>126,4</td>
<td>152,4</td>
</tr>
<tr>
<td>16:1</td>
<td>0,58</td>
<td>260</td>
<td>55,0</td>
<td>106,6</td>
<td>112,0</td>
<td>119,0</td>
<td>126,4</td>
<td>152,4</td>
</tr>
<tr>
<td>20:1</td>
<td>0,58</td>
<td>260</td>
<td>55,0</td>
<td>106,6</td>
<td>112,0</td>
<td>119,0</td>
<td>126,4</td>
<td>152,4</td>
</tr>
<tr>
<td>25:1</td>
<td>0,58</td>
<td>260</td>
<td>55,0</td>
<td>106,6</td>
<td>112,0</td>
<td>119,0</td>
<td>126,4</td>
<td>152,4</td>
</tr>
<tr>
<td>36:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>45:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>60:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>80:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>100:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>120:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>160:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>200:1</td>
<td>0,67</td>
<td>330</td>
<td>67,6</td>
<td>119,2</td>
<td>124,6</td>
<td>131,6</td>
<td>139,0</td>
<td>165,0</td>
</tr>
<tr>
<td>240:1</td>
<td>0,75</td>
<td>410</td>
<td>80,2</td>
<td>131,8</td>
<td>137,2</td>
<td>144,2</td>
<td>151,6</td>
<td>177,6</td>
</tr>
<tr>
<td>360:1</td>
<td>0,75</td>
<td>410</td>
<td>80,2</td>
<td>131,8</td>
<td>137,2</td>
<td>144,2</td>
<td>151,6</td>
<td>177,6</td>
</tr>
<tr>
<td>480:1</td>
<td>0,75</td>
<td>410</td>
<td>80,2</td>
<td>131,8</td>
<td>137,2</td>
<td>144,2</td>
<td>151,6</td>
<td>177,6</td>
</tr>
<tr>
<td>800:1</td>
<td>0,75</td>
<td>410</td>
<td>80,2</td>
<td>131,8</td>
<td>137,2</td>
<td>144,2</td>
<td>151,6</td>
<td>177,6</td>
</tr>
<tr>
<td>1600:1</td>
<td>0,75</td>
<td>410</td>
<td>80,2</td>
<td>131,8</td>
<td>137,2</td>
<td>144,2</td>
<td>151,6</td>
<td>177,6</td>
</tr>
</tbody>
</table>

For notes on technical data and lifetime performance refer to “Technical Information”.
Edition 01.02.2007
A.3 MicroMo Encoder

The following datasheet is available from MicroMo [65]. This datasheet describes several encoders available from MicroMo. The particular encoder selected for the Omnibot has the following attributes:

- HEDM 5500 J
- 2 Channel, 1024 line per revolution
- Sizing pertains to DC-Micromotor #3257
Encoders

Optical Encoders

Features:
- 100 to 1024 Lines per revolution
- 2 or 3 Channels
- Digital output

### Series 5500, 5540

<table>
<thead>
<tr>
<th>Lines per revolution</th>
<th>HEDS 5500</th>
<th>HEDS 5540</th>
<th>HEDM 5500</th>
</tr>
</thead>
<tbody>
<tr>
<td>N 100 - 500</td>
<td>100 - 500</td>
<td>100 - 500</td>
<td>1 000 - 1024</td>
</tr>
<tr>
<td>Signal output, square wave</td>
<td>2</td>
<td>2+1 index</td>
<td>2</td>
</tr>
<tr>
<td>Supply voltage</td>
<td>V CC 4,5 ... 5,5</td>
<td>V DC</td>
<td></td>
</tr>
<tr>
<td>Current consumption, typical (Vcc = 5 V DC)</td>
<td>V CC 57</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>Pulse width</td>
<td>P 180 ± 45</td>
<td>180 ± 35</td>
<td>180 ± 45</td>
</tr>
<tr>
<td>Phase shift, channel A to B</td>
<td>Φ 90 ± 20</td>
<td>90 ± 15</td>
<td>90 ± 15</td>
</tr>
<tr>
<td>Logic state width</td>
<td>S 90 ± 45</td>
<td>90 ± 35</td>
<td>90 ± 45</td>
</tr>
<tr>
<td>Cycle</td>
<td>C 360 ± 5,5</td>
<td>360 ± 5,5</td>
<td>360 ± 7,5</td>
</tr>
<tr>
<td>Signal rise/fall time, typical</td>
<td>tr/tf 0,25 / 0,25</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Frequency range (1)</td>
<td>f up to 100</td>
<td>up to 100</td>
<td>up to 100</td>
</tr>
<tr>
<td>Inertia of code disc</td>
<td>J 0,6</td>
<td>up to 100</td>
<td>up to 100</td>
</tr>
<tr>
<td>Operating temperature range</td>
<td>~ 40 ... + 100</td>
<td>~ 40 ... + 70</td>
<td></td>
</tr>
</tbody>
</table>

(1) Velocity (rpm) = f (Hz) x 60/N

(2) HEDS 5540 requires pull-up resistors of 2,7 kΩ between pins 2, 3, 5 and 4 (V CC)

### Ordering Information

Encoder type | number of channels | lines per revolution | For combination with:
--- | --- | --- | ---
HEDS 5500 C | 2 | 100 | DC-Micromotors and DC-Motor-Tachos Series
HEDS 5500 A | 2 | 500 | 2236, 2233, 2251
HEDS 5540 C | 2+1 | 100 | 2338, 2342
HEDS 5540 A | 2+1 | 500 | 2642, 2657
HEDM 5500 B | 2 | 1000 | 3242, 3257, 3557, 3863
HEDM 5500 J | 2 | 1024 | brushless DC-Servomotors Series

Ordering information

For notes on technical data and lifetime performance refer to “Technical Information”.

### Features

These incremental shaft encoders in combination with the DC-Micromotors and brushless DC-Servomotors are designed for indication and control of both, shaft velocity and direction of rotation as well as for positioning.

A LED source and lens system transmits collimated light through a low inertia metal disc to give two channels with 90° phase shift.

The single 5 volt supply and the two or three channel digital output signals are interfaced with a 5-pin connector.

Motors with ball bearings are recommended for continuous operation at low and high speeds and for elevated radial shaft load.

Details for the Motors and suitable reduction gearheads are on separate catalog pages.

### Output signals / Circuit diagram / Connector information

Output signals HEDS, HEDM with clockwise rotation as seen from the shaft end

Connection diagram HEDS 5540 requires pull-up resistors

For notes on technical data and lifetime performance refer to “Technical Information”.

Specifications subject to change without notice.

www.faulhaber.com
For notes on technical data and lifetime performance refer to “Technical Information”.

Specifications subject to change without notice.

www.faulhaber.com
A.4 Victor 884

The following datasheet is available from IFIRobotics [67]. This datasheet describes the various physical properties of the 12 V Victor 884 which was originally used on the Omnibot. A user manual can be found in [67]. The Victor 884 also has the following specifications:

- Standard R/C Type PWM Control Signal
- 6V - 15V Operating Voltage
- 40A Maximum Current
- 3\% Minimum Throttle
- 0.25 lbs
- 6-32 Screw Terminals Power Connectors
A.5 Dual VNH3SP30 Motor Driver Carrier MD03A

The following datasheet is available from Pololu Corporation [68]. This datasheet describes the various properties of the MD03A which was used on the Omnibot. The MD03A also has the following specifications:

- 10 kHz Maximum PWM frequency
- 9A Continuous Current
Dual VN3SP30 Motor Driver Carrier MD03A

If you are looking to drive two high-power motors through one compact unit, these dual VN3SP30 motor driver carriers are perfect for you. With these boards, it’s easy to get a medium-sized, differential drive robot running in no time. The VN3 version is a lower-cost option than its VN2 counterpart.

Overview
The Pololu dual high-power motor drivers are compact carriers for the VN3SP30 and VN2SP30 motor driver integrated circuits from ST. The board incorporates most of the components of the typical application diagram on page 8 of the VN2SP30 datasheet, including pull-up and current-limiting resistors and a FET for reverse battery protection. (The current sense circuit is populated on both versions of the board, but only the VN2SP30 supports current sense.) To keep the number of I/O lines down, the two enable/diagnostic lines on each chip are tied together. All you need to add is a microcontroller or other control circuit to turn the H-Bridges on and off.

Please note that we offer several other products based on these same chips, including single carrier boards for controlling one motor, the qik 2s10v12 dual serial motor controller, the TReX motor controller, the high-power motor controller with feedback, and the Orangutan X2 robot controller. We also have two higher-power (single) motor drivers that can deliver more current over a wider operating voltage range: the high-power motor driver 18v15 and the high-power motor driver 36v9.

In a typical application, the power connections are made on one end of the board, and the control connections are made on the other end. +5 volts must be supplied to the board through the smaller 0.1"-spaced pins; the input voltage is available at those pins as well, but the connection is not intended for currents exceeding a few amps. The diagnostic pins can be left disconnected if you do not want to monitor the fault conditions of the motor drivers. INA and INB control the direction of each motor, and the PWM pins turns the motors on or off. For the VN2SP30 version, the current sense (CS) pins will output approximately 0.13 volts per amp of output current. If you want to add current sensing to the VN3SP30 version, or if you want higher-accuracy current sensing with the VN2SP30 version, please consider our ±30A ACS714 current sensor carrier.

The dual motor driver PCB includes provisions for installing up to three large capacitors to limit disturbances on the main power line. Two 10mm radial capacitors may be mounted between the motor driver ICs, and an axial capacitor may be mounted between the ICs and power connections. It is generally not necessary to use all three capacitors; two radial capacitors are included with each unit. For applications that require a low profile, a single capacitor can be installed on its side as shown in the picture to the right.

VN3SP30 and VN2SP30 Comparison
MOSFET on-resistance (per leg) 34 mΩ 19 mΩ
Maximum PWM frequency 10 kHz 20 kHz
Current sense none approximately 0.13 volts per amp
Over-voltage shutdown none (operates up to 30 V) could be as low as 16 V (19 V typical)
Time to overheat at 20 A* 8 seconds 55 seconds
Time to overheat at 15 A* 80 seconds 150 seconds
Current for infinite run time* 9 A 14 A

*Typical results using Pololu motor driver carrier with 100% duty cycle at room temperature.

Real-world power dissipation considerations
The motor drivers have maximum current ratings of 30 A continuous. However, the chips by themselves will overheat at lower currents (see table above for typical values). The actual current you can deliver will depend on how well you can keep the motor drivers cool. The carrier printed circuit board is designed to draw heat out of the motor driver chips, but performance can be improved by adding a heat sink. In our tests, we were able to deliver short durations (on the order of milliseconds) of 30 A and several seconds of 20 A without overheating. At 6 A, the chip gets just barely noticeably warm to the touch. For high-current installations, the motor and power supply wires should also be soldered directly instead of going through the supplied terminal blocks, which are rated for up to 15 A.

Many motor controllers or speed controllers can have peak current ratings that are substantially higher than the continuous current rating; this is not the case with these motor drivers, which have a 30 A continuous rating and a over-current protection that can kick in as low as 30 A (45 A typical). Therefore, the stall current of your motor should not be more than 30 A. (Even if you expect to run at a much lower average current, the motor can still draw high currents when it is starting or if you use low duty cycle PWM to keep the average current down.)

Schematic of the Pololu Dual High Current Motor Driver Carrier

http://www.pololu.com/catalog/product/707
A.6 Encoder Counter

The following datasheet is available from Avago Technologies [85]. This datasheet describes the properties of the HCTL-2021 Encoder Counters which was used on the Omnibot. More information can be found at [85].
Description
The HCTL-2021/2017 is CMOS ICs that performs the quadrature decoder, counter, and bus interface function. The HCTL-2021/2017 is designed to improve system performance in digital closed loop motion control systems and digital data input systems. It does this by shifting time intensive quadrature decoder functions to a cost effective hardware solution. The HCTL-2021/2017 consists of a quadrature decoder logic, a binary up/down state counter, and an 8-bit bus interface. The use of Schmitt-triggered CMOS inputs and input noise filters allows reliable operation in noisy environments. The HCTL-2021/2017 contains 16-bit counter and provides TTL/CMOS compatible tri-state output buffers. Operation is specified for a temperature range from -40 to +85°C at clock frequencies up to 33MHz.

The HCTL-2021/2017 provides quadrature decoder output signals and cascade signals for use with many standard computer ICs.

The HCTL-2021/2017 is compliant to RoHS directive and had been declared as a lead free product.

Features
- Interfaces Encoder to Microprocessor
- 33 MHz Clock Operation
- High Noise Immunity: Schmitt Trigger Inputs and Digital Noise Filter
- 16-Bit Binary Up/Down Counter
- Latched Outputs
- 8-Bit Tristate Interface
- 8 or 16-Bit Operating Modes
- Quadrature Decoder Output Signals, Up/Down and Count
- Cascade Output Signals, Up/Down and Count
- Substantially Reduced System Software
- 5V Operation (VDD – VSS)
- TTL/CMOS Compatible I/O
- Operating Temperature: -40°C to 85°C
- 16-pin and 20-Pin Launch Pad

Applications
- Interface Quadrature Incremental Encoders to Microprocessors
- Interface Digital Potentiometers to Digital Data Input Buses

Devices

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Description</th>
<th>Pinout</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCTL-2017</td>
<td>33 MHz clock operation. 16-bit counter.</td>
<td>PINOUT A</td>
</tr>
<tr>
<td>HCTL-2021</td>
<td>33 MHz clock operation. 16-bit counter.</td>
<td>PINOUT B</td>
</tr>
<tr>
<td></td>
<td>Quadrature decoder output signals.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cascade output signals.</td>
<td></td>
</tr>
</tbody>
</table>
Soldering and Mounting Considerations

It is recommended to use manual soldering for HCTL-2021/2017 launch pad devices due to the characteristics of the material used in the launch pad design that not allow wave soldering.

Direct mounting on printed circuit board (PCB) only is recommended for HCTL-2021/2017 launch pad devices.

Mounting gap of 1mm between the base of the launch pad and customer’s printed circuit board (PCB) is required.

NOTE: Precaution is required in order to avoid bend or loose pin during product handling.

Package Dimensions with Tolerances

<table>
<thead>
<tr>
<th></th>
<th>Length (L)</th>
<th>Width (W)</th>
<th>Thickness (T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCTL-2017</td>
<td>22.86 ± 0.5 mm</td>
<td>12.70 ± 0.5 mm</td>
<td>1.67 ± 0.25 mm</td>
</tr>
<tr>
<td>HCTL-2021</td>
<td>27.94 ± 0.5 mm</td>
<td>12.70 ± 0.5 mm</td>
<td>1.67 ± 0.25 mm</td>
</tr>
</tbody>
</table>

(dimension in mm)
A.7 HCS12 Microcontroller

The following is a brief description of the microcontroller used on the Omnibot. This description, datasheets, and other information is available through [70]. Also, the pin layout for the CSM12D from the manual is on the following page.

The MC9S12DT256 microcontroller unit (MCU) is a 16-bit device composed of standard on-chip peripherals including a 16-bit central processing unit (HCS12 CPU), 256K bytes of Flash EEPROM, 12.0K bytes of RAM, 4.0K bytes of EEPROM, 2 asynchronous serial communications interfaces (SCI), three serial peripheral interfaces (SPI), an 8 channel IC/OC enhanced capture timer, two 8-channel, 10-bit analog-to-digital converters (ADC), an 8-channel pulse-width modulator (PWM), 89 discrete digital I/O channels (Port A, Port B, Port K and Port E), 20 discrete digital I/O lines with interrupt and wakeup capability, three CAN 2.0 A, B software compatible modules (MSCAN12), and an Inter-IC Bus. System resource mapping, clock generation, interrupt control and bus interfacing are managed by the System Integration Module (SIM). The MC9S12DT256 has full 16-bit data paths throughout. However, the external bus can operate in an 8-bit narrow mode so single 8-bit wide memory can be interfaced for lower cost systems. The inclusion of a PLL circuit allows power consumption and performance to be adjusted to suit operational requirements.

Features

- 16-bit HCS12 CPU
  
  Upward compatible with M68HC11 instruction set
  
  Interrupt stacking and programmer’s model identical to M68HC11
  
  Instruction pipe
  
  Enhanced indexed addressing

- Multiplexed External Bus
• Memory

  256K byte Flash EEPROM

  4.0K byte EEPROM

  12.0K byte RAM

• Two 8-channel Analog-to-Digital Converters

  10-bit resolution

• Three 1M bit per second, CAN 2.0 A/B software compatible modules
**MCU I/O PORT**

Connector J1 provides access to the MC9S12DT256 I/O signals. The figures below show the pin-out for the MCU I/O connector. Only signal XCLS is not available at connector J1.

**Figure 7: Connector J1**

<table>
<thead>
<tr>
<th>VAUX</th>
<th>1</th>
<th>2</th>
<th>PE1/IRQ*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>3</td>
<td>4</td>
<td>RESET*</td>
</tr>
<tr>
<td>PS1/TXD0</td>
<td>5</td>
<td>6</td>
<td>MODC/BKGD</td>
</tr>
<tr>
<td>PS0/RXD0</td>
<td>7</td>
<td>8</td>
<td>PP7/KWP7/PWM7/SCK2</td>
</tr>
<tr>
<td>PP0/KWP0/PWM0/MISO1</td>
<td>9</td>
<td>10</td>
<td>PAD07/AN07</td>
</tr>
<tr>
<td>PP1/KWP1/PWM1/MOSI1</td>
<td>11</td>
<td>12</td>
<td>PAD06/AN06</td>
</tr>
<tr>
<td>PT0/IOC0</td>
<td>13</td>
<td>14</td>
<td>PAD05/AN05</td>
</tr>
<tr>
<td>PT1/IOC1</td>
<td>15</td>
<td>16</td>
<td>PAD04/AN04</td>
</tr>
<tr>
<td>PM4/RXCAN2/RXCAN0/RXCAN4/MOSI0</td>
<td>17</td>
<td>18</td>
<td>PAD00/AN00</td>
</tr>
<tr>
<td>PM2/RXCAN1/RXCAN0/MISO0</td>
<td>19</td>
<td>20</td>
<td>PAD01/AN01</td>
</tr>
<tr>
<td>PM5/TXCN2/TXCAN0/TXCAN4/SCK0</td>
<td>21</td>
<td>22</td>
<td>PAD02/AN02</td>
</tr>
<tr>
<td>PM3/TXCAN1/TXCAN0/SS0*</td>
<td>23</td>
<td>24</td>
<td>PAD03/AN03</td>
</tr>
<tr>
<td>PA7/ADDR15/DATA15</td>
<td>25</td>
<td>26</td>
<td>PJ7/KWJ7/TXCAN4/SCL0</td>
</tr>
<tr>
<td>PA6/ADDR14/DATA14</td>
<td>27</td>
<td>28</td>
<td>PJ6/KWJ6/RXCAN4/SDA0</td>
</tr>
<tr>
<td>PA5/ADDR13/DATA13</td>
<td>29</td>
<td>30</td>
<td>PP2/KPP2/PWM2/SCK1</td>
</tr>
<tr>
<td>PA2/ADDR10/DATA10</td>
<td>35</td>
<td>36</td>
<td>PP5/KWP5/PWM5/MOSI2</td>
</tr>
<tr>
<td>PA1/ADDR9/DATA9</td>
<td>37</td>
<td>38</td>
<td>PS2/RXD1</td>
</tr>
<tr>
<td>PA0/ADDR8/DATA8</td>
<td>39</td>
<td>40</td>
<td>PS3/TXD1</td>
</tr>
<tr>
<td>PB7/ADDR7/DATA7</td>
<td>41</td>
<td>42</td>
<td>PE0/XIRQ*</td>
</tr>
<tr>
<td>PB6/ADDR6/DATA6</td>
<td>43</td>
<td>44</td>
<td>PE2/RW</td>
</tr>
<tr>
<td>PB5/ADDR5/DATA5</td>
<td>45</td>
<td>46</td>
<td>PE3/LSTRB*</td>
</tr>
<tr>
<td>PB4/ADDR4/DATA4</td>
<td>47</td>
<td>48</td>
<td>PE4/ECLK</td>
</tr>
<tr>
<td>PB3/ADDR3/DATA3</td>
<td>49</td>
<td>50</td>
<td>PT2/IOC2</td>
</tr>
<tr>
<td>PB2/ADDR2/DATA2</td>
<td>51</td>
<td>52</td>
<td>PT3/IOC3</td>
</tr>
<tr>
<td>PB1/ADDR1/DATA1</td>
<td>53</td>
<td>54</td>
<td>PT4/IOC4</td>
</tr>
<tr>
<td>PB0/ADDR0/DATA0</td>
<td>55</td>
<td>56</td>
<td>PT5/IOC5</td>
</tr>
<tr>
<td>PM1/TXCAN0/TXB</td>
<td>57</td>
<td>58</td>
<td>PT6/IOC6</td>
</tr>
<tr>
<td>PM0/RXCAN0/RXB</td>
<td>59</td>
<td>60</td>
<td>PT7/IOC7</td>
</tr>
</tbody>
</table>
Appendix B

Electrical Diagrams

B.1 HCS12 Wiring Diagram

The following diagram shows the pin layout of the microcontroller corresponding to the electrically connected elements on the Omnibot. Note that motor 4 has been re-labeled as motor 0.
B.2 Encoder Counter Wiring Diagram

This diagram shows the pin layout of the 4 encoder counter ICs corresponding to the electrically connected elements on the Omnibot. The encoder counter 8-bit output is connected in parallel across the 4 ICs. Also, note that the 10 MHz signal is supplied by a separate IC.

For Encoder Counter
n = 0, 1, 2, 3
Appendix C

PID Tuning Graphs
$K_p = 1.75 \quad K_I = 0 \quad K_D = 0$

$K_p = 1.75 \quad K_I = 0.020 \quad K_D = 0$

$K_p = 1.75 \quad K_I = 0.040 \quad K_D = 0$

$K_p = 1.75 \quad K_I = 0.060 \quad K_D = 0$

$K_p = 1.75 \quad K_I = 0.080 \quad K_D = 0$

$K_p = 1.75 \quad K_I = 0.10 \quad K_D = 0$
$K_p=1.75 \quad K_I = 0.12 \quad K_D = 0$

$K_p=1.75 \quad K_I = 0.060 \quad K_D = 1.00$

$K_p=1.75 \quad K_I = 0.060 \quad K_D = 2.00$

$K_p=1.75 \quad K_I = 0.060 \quad K_D = 3.00$

$K_p=1.75 \quad K_I = 0.060 \quad K_D = 4.00$

$K_p=1.75 \quad K_I = 0.060 \quad K_D = 5.00$
$K_p = 1.75 \quad K_I = 0.060 \quad K_D = 6.00$

$K_p = 1.75 \quad K_I = 0.060 \quad K_D = 7.00$

$K_p = 1.75 \quad K_I = 0.060 \quad K_D = 8.00$

$K_p = 1.75 \quad K_I = 0.050 \quad K_D = 3.00$

$K_p = 1.75 \quad K_I = 0.070 \quad K_D = 3.00$

$K_p = 1.75 \quad K_I = 0.080 \quad K_D = 6.00$
$K_p = 1.75 \quad K_I = 0.080 \quad K_D = 7.00$

$K_p = 1.00 \quad K_I = 0.100 \quad K_D = 7.00$

$K_p = 1.50 \quad K_I = 0.080 \quad K_D = 7.00$

$K_p = 1.50 \quad K_I = 0.100 \quad K_D = 7.00$

$K_p = 2.10 \quad K_I = 0.040 \quad K_D = 7.00$

$K_p = 2.40 \quad K_I = 0.020 \quad K_D = 10.00$
\[ K_p = 2.40 \quad K_I = 0.020 \quad K_D = 1.09 \]

\[ K_p = 2.10 \quad K_I = 0.018 \quad K_D = 10.50 \]
Appendix D

Program Code for the Omnibot

The following sections are final versions of program code for the Omnibot. Below is a brief description of each program:

OmnibotControlBemisv8.c

- Main program for Omnibot MCU. Developed using Codewarrior and Processor Expert. Must be run in conjunction with Events.c

Events.c

- Functions for the Omnibot MCU such as speed control and serial i/o. Developed using Codewarrior and Processor Expert. Must be run in conjunction with OmnibotControlBemisv8.c

OmnibotPinger.cpp

- ROS node for remote desktop. Subscribes to topic joyChatter and publishes commands to topic ping_bus once it receives data from subscribed topic pong_bus.

OmnibotPonger.cpp

- ROS node for onboard data processor. Receives commands from subscribed topic ping_bus, sends commands through serial to MCU, receives data from MCU and publishes data to topic pong_bus.
OmnibotUI.cpp

- ROS node for remote desktop. Interfaces with joystick through serial and publishes commands under topic joyChatter.
D.1 OmnibotControlBemisv8.c

/** ###################################################################
** Filename : OmnibotControlBemisv8.C
** Project : OmnibotControlBemisv8
** Processor : MC9S12DT256BCPV
** Version : Driver 01.11
** Compiler : Metrowerks HC12 C Compiler
** Date/Time : 28/05/2009, 9:08 AM
** Abstract :
** Main module.
** Here is to be placed user’s code.
** Settings :
** Contents :
** No public methods
**
** (c) Copyright UNIS, spol. s r.o. 1997-2005
** UNIS, spol. s r.o.
** Jundrovska 33
** 624 00 Brno
** Czech Republic
** http : www.processorexpert.com
** mail : info@processorexpert.com
** ###################################################################*/

/* MODULE OmnibotControlBemisv8 */

/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "M0PWM.h"
#include "M1PWM.h"
#include "M2PWM.h"
#include "M3PWM.h"
#include "TI1.h"
#include "Serial.h"
#include "Joystick.h"
#include "Button.h"
#include "BUS.h"
#include "EncRead.h"
#include "ModeSwitch1.h"
#include "FC321.h"
#include "MotorDir.h"

/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

void DoAllTasks(void);
void InitTask(void);

void main(void)
{
    //*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!!
```c
***
PE_low_level_init();
/*** End of Processor Expert internal initialization.

***/

Joystick_Start(); // Initialize joystick
InitTask(); // Initialize Tasks
EncRead_PutVal(1);// Set all encoder counter inputs to high

for(;;)
{
  DoAllTasks(); // Run tasks
}
/*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/ for(;;){}
/*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!!/***

/* END OmnibotControlBemisv8 */
/*
** ###################################################################
**
** This file was created by UNIS Processor Expert 2.96 [03.76]
** for the Freescale HCS12 series of microcontrollers.
**
** ###################################################################
*/
```
D.2 Events.c

/** ###################################################################
** Filename : Events.C
** Project : omnibotControlBemisv8
** Processor : MC9S12DT256BCPV
** Beantype : Events
** Version : Driver 01.04
** Compiler : Metrowerks HC12 C Compiler
** Date/Time : 30/03/2009, 11:17 AM
** Abstract :
** This is user’s event module.
** Put your event handler code here.
** Settings :
** Contents :
** Joystick_OnEnd - void Joystick_OnEnd(void);
** Serial_OnError - void Serial_OnError(void);
** Serial_OnRxChar - void Serial_OnRxChar(void);
** Serial_OnTxChar - void Serial_OnTxChar(void);
** Serial_OnFullRxBuf - void Serial_OnFullRxBuf(void);
** Serial_OnFreeTxBuf - void Serial_OnFreeTxBuf(void);
** TI1_OnInterrupt - void TI1_OnInterrupt(void);
** M3Encoder_OnInterrupt - void M3Encoder_OnInterrupt(void);
** M2Encoder_OnInterrupt - void M2Encoder_OnInterrupt(void);
** M1Encoder_OnInterrupt - void M1Encoder_OnInterrupt(void);
** M0Encoder_OnInterrupt - void M0Encoder_OnInterrupt(void);
**
** (c) Copyright UNIS, spol. s r.o. 1997-2005
** UNIS, spol. s r.o.
** Jundrovska 33
** 624 00 Brno
** Czech Republic
** http : www.processorexpert.com
** mail : info@processorexpert.com
** ###################################################################
*/

#include "Cpu.h"
#include "Events.h"
#include "math.h"
#include <stdio.h>
#include <stdlib.h>

#pragma CODE_SEG DEFAULT

/*
****************************************************************************
UPDATED VERSION OF SOURCE CODE
****************************************************************************
*/
CONSTANT AND VARIABLE DECLARATIONS

/*
 *****************************************************************************
CONSTANT AND VARIABLE DECLARATIONS
*****************************************************************************
*/

// PID gain constants
const float KP=2.4;
const float KD=1.09;
const float KI=0.020;

const int MAX_SERIAL_LAG = 1500; // Max lag before corrective action taken
// in microseconds
const float MAX_SPEED = 10; //in rad/s wheel angular velocity

taskId VALUES

Each Task is assigned a value (constant) that is decremented each time
the Timer Interrupt is executed (every 1ms). When the value of a task
reaches zero, then that task is fired and its value is reset after it finishes
executing.

This process is used to Schedule Task Execution within the program.

*****************************************************************************
*/

// Task Scheduling Values (0.1 ms increments)

const int TASK1_MAX = 60; //Joystick ADC
const int TASK2_MAX = 60; // Jacobian Conversion
const int TASK3_MAX = 150; // Serial Transmit for Monitoring, must be
                          // >150 when remotely driving
const int TASK4_MAX = 60; // Encoder Speed Calculation and Speed Control

// Task values that are decremented every 0.1ms
int task1val;
int task2val;
int task3val;
int task4val;
int countVal=0;

// Various other global variables, most of these variables global for
// debugging purposes or cross function communication
bool convertingString=FALSE;
bool sendingData=FALSE;
word encoders[4];
word prev_encoders[4];
float encSpeed[4];
int prevSpeeds[10][4];
float duration;
long int SpeedSum[4];
int SpeedAvg[4];
char SerialIn[128]=0;
int ellenTemp[4];
const char *pos;
int serialCount=0;
float MotorRequest[4];
int RemoteControlVars[5]={KP*1000,KD*1000,KI*1000,0,0};
int ModeSwitch;
float DesiredSpeed[4];
int Mresult[4];

// Gains used in the PID Speed Controller, separated in the event of
// different gains for each motor
float SpeedKd[4]={KD,KD,KD,KD};
float SpeedKi[4]={KI,KI,KI,KI};

// Variables used in PID control
float SpeedError[4]={0,0,0,0};
float prev_SpeedError[4]={0,0,0,0};
float Accel[4]={0,0,0,0};
float SpeedErrorSum[4]={0,0,0,0};

//**********************************************
// Joystick Variables
int xaxist[6], yaxist[6], zaxist[6];
int xaxisd, yaxisd, zaxisd;
int PushButton;
int Mi[4];

//**********************************************

//Serial Output Vars
char Out2Data[20];
byte status[6];
byte outCount=0;
byte HighByte;
byte LowByte;
word temp;
unsigned int killTime=0;
bool MotorsOn=FALSE;
//to create motor response characteristic
int motor_count=0;
/ * 
***************************************************************************** 
FUNCTIONS 
*****************************************************************************/ 
/* 
***************************************************************************** 
MOTOR CONTROL 
*****************************************************************************/ 

// Motor PWM signal generation based on PID corrective signals 
void MotorControl(float MotorIns[4]) 
{

byte n;

if(MotorsOn) // joystick button pressed, use PID velocity to move motors 
{
    // Set the direction of rotation of each motor
    for(n=0;n<4;n++)
    {
        // convert rad/s to -255 -> 255 scale
        MotorIns[n] = 0.5*MotorIns[n]*255/MAX_SPEED;

        //clipping
        if(MotorIns[n] > 250)
        {
            MotorIns[n]=250;
        }
        if(MotorIns[n] < -250)
        {
            MotorIns[n]= -250;
        }

        if(MotorIns[n] < 0) // 0 to -255 go backward
        {
            Mresult[n]=(int)ceil(-MotorIns[n]);
            MotorDir_SetBit(2*n);
            MotorDir_ClrBit(2*n+1);
        }
        else // 0 to 255 go forward
        {
            Mresult[n]=(int)ceil(MotorIns[n]);
            MotorDir_SetBit(2*n+1);
            MotorDir_ClrBit(2*n);
        }
    }
}
}
else  // joystick button not pressed, set all output to zero, and reset
// PID counts
{
    for(n=0;n<4;n++)
    {
        Mresult[n]=0;
        SpeedErrorSum[n]=0;
        SpeedError[n]=0;
        Accel[n]=0;
    }
}
//Clipping to prevent unnecessary motor power while stopped
for (n=0;n<4;n++)
{
    if (Mresult[n]<5)
    {
        Mresult[n]=0;
    }
}
// Set PWM signal for each motor
status[1] =M0PWM_SetRatio8((byte)Mresult[0]);
status[1] =M1PWM_SetRatio8((byte)Mresult[1]);
status[1] =M3PWM_SetRatio8((byte)Mresult[3]);
}

/****************************************************************************
JACOBIAN
****************************************************************************/

// Jacobian conversion from desired platform velocity to desired motor velocities
void Jacobian(void)
{
    int z;

    // shrinks excessive motion as a result of addition of translation plus
    // rotation, limits drive to -255 to 255 scale
    while(xaxisd + zaxisd > 254 ||xaxisd + zaxisd < -254|| yaxisd + zaxisd >
    254 ||yaxisd + zaxisd < -254|| zaxisd - (xaxisd) > 254 ||zaxisd -
    xaxisd < -254|| zaxisd - (yaxisd) > 254||zaxisd - yaxisd < -254)
    {
        if (xaxisd != 0)
            xaxisd = xaxisd - (xaxisd/(abs(xaxisd)));
        if (yaxisd != 0)
            yaxisd = yaxisd - (yaxisd/(abs(yaxisd)));
    }
if (zaxisd != 0)
    zaxisd = zaxisd - (zaxisd/(abs(zaxisd)));
}

// holding the deadband at 0 (accounting for variance in joystick signal)
if (xaxisd < 20 && xaxisd > -20)
    xaxisd = 0;
if (yaxisd < 20 && yaxisd > -20)
    yaxisd = 0;
if (zaxisd < 20 && zaxisd > -20)
    zaxisd = 0;

// Mi = -255 -> 255 scale
Mi[0] = (int) yaxisd + zaxisd;
Mi[1] = (int) xaxisd + zaxisd;
Mi[2] = (int) -yaxisd + zaxisd;
Mi[3] = (int) -xaxisd + zaxisd;

// converted to rad/s with limitation toward maximum angular velocity
inferred, sent to PID control
for (z = 0; z < 4; z++)
{
    DesiredSpeed[z] = Mi[z] * MAX_SPEED / 255;
}

/************************************************************************
CONVERT ENCODER COUNTS TO RADIANS
************************************************************************/
float ConvertToAngle(int en_count)
{
    float angle;
    // converts encoder counts to rad/s
    angle = (en_count * (2.0 * 3.1415) / (45.0 * 4.0 * 1024.0));
    return angle;
}

/************************************************************************
COUNTER IC READ
************************************************************************/
// This function reads in the encoder counts from all 4 encoder counter ICs.
// It must check each IC one at a time by first setting low the OE pin and the SEL pin. The IC will
// then report back the
// first half of a 16-bit number on a 8-bit bus. Then the SEL pin is set to
// high.
// This gives the second half of the number. That number is summed then sent
// to the motor
// velocity calculation. The OE pin is then set to high, turning off the IC's
// output, and the function
// moves to the next IC in a similar fashion.

void CounterRead(void)
{
    int i=0;
    int j=0;
    word tempEnc;

    for(i=0;i<4;i++)
    {
        EncRead_PutBit(j,0); // SEL -> low
        EncRead_PutBit(j+1,0); // OE -> low

        HighByte= BUS_GetVal(); // Read high byte
        temp = HighByte; // Store high byte

        EncRead_PutBit(j,1); // SEL -> high
        temp <<= 8; // Shift to the left by 8 bits

        LowByte= BUS_GetVal(); // Read low byte
        temp = temp + LowByte; // Add low byte to high byte

        EncRead_PutBit(j,1); // SEL -> high
        EncRead_PutBit(j+1,1); // OE -> high

        j += 2; // increment to next IC

        encoders[i] = temp; // Record Encoder Count Value
    }

    // Re-arranging order of values in encoders array
    tempEnc = encoders[0];
    encoders[0] = encoders[3];
    encoders[3] = tempEnc;
    tempEnc = encoders[2];
    encoders[2] = encoders[1];
    encoders[1] = tempEnc;
}

/*
*******************************************************************************

171
PID VELOCITY CONTROL
*****************************************************************
*/

void ReadEncsAndSpeedControl()
{
    byte jo;

    // ******** READ THE COUNTER IC VALUES ********
    CounterRead();
    duration = countVal*(-0.0001); // calculates how long its been in
    // microseconds since the last measurement
    countVal=0; // resets counter
    for(jo=0;jo<4;jo++)
    {

        //******** MOTOR VELOCITY CALCULATION ********
        // This code takes care to monitor for overflow when it occurs in the
        // encoder counters and calculates
        // the measured velocity of each motor using the previous and current
        // encoder count measurement.

        // Counter Overflow for Forward Rotation
        if(encoders[jo] < 10000 && prev_encoders[jo] > 55000)
        {
            encSpeed[jo]=ConvertToAngle(((65535-prev_encoders[jo])+encoders
            [jo]))/duration;
        }
        // Counter Overflow for Reverse Rotation
        else if(encoders[jo]>55000 && prev_encoders[jo]<10000)
        {
            encSpeed[jo]=ConvertToAngle(((encoders[jo]-65535)-prev_encoders
            [jo]))/duration;
        }
        else
        {
            encSpeed[jo]=ConvertToAngle((encoders[jo]-prev_encoders[jo]))/duration;
        }
    prev_encoders[jo]=encoders[jo];

        //******** SPEED CONTROL ********
        // This code serves as the PID control for each motor.
        SpeedError[jo] = (DesiredSpeed[jo]-encSpeed[jo]); // error
        //change in error
        Accel[jo]=(SpeedError[jo]-prev_SpeedError[jo])/(duration*1000);
        // sum of the error
        SpeedErrorSum[jo]=(SpeedErrorSum[jo]+(SpeedError[jo]*duration*1000));

        // PID equation gives the corrective signals
        MotorRequest[jo] = (float)DesiredSpeed[jo]+(SpeedKp[jo]*SpeedError[jo]) +
        (SpeedKd[jo]*Accel[jo]) + (SpeedKi[jo]*SpeedErrorSum[jo]);
prev_SpeedError[jo]=SpeedError[jo]; //for checking change in next cycle

// send corrective signals to motor control function
MotorControl(MotorRequest);

/*
*********************************************************
CONVERT RADIANs TO ENCODER COUNTS
*********************************************************/

// Converts a (radian) angle into an encoder count
int ConvertToEncoderCount(float angle)
{
    int enc;
    enc = (int)ceil(-(angle*(45*1024)/(2*3.1415)));
    return enc;
}

/*
*********************************************************
CONVERT RECEIVED SERIAL DATA INTO COMMANDS
*********************************************************/

// This function takes a string input consisting of commands from the remote
// host and converts them into
// usable integers. Since the input is usually clogged with bad bits from the
// serial transfer, this
// function cleans the data by checking for several things, if the motor
// commands are numbers between
// 0 and 255, if the first character is the letter "R", and if the final
// character is a 0 or 1.
void ConvertSerialToCommands(char * RTStr)
{
    byte xc=0;
    byte jake;

    // skip conversion if the first character of the string is not "R"
    if (RTStr[0]==82)
    {
        convertingString = TRUE; // set this variable to prevent RTStr from being
            // overwritten
        for (jake=0;jake<19;jake++) // shift whole character array over by 1
        {
            RTStr[jake] = RTStr[jake+1];
        }
        // replace space characters (32) with the number "0" (48)
        for (jake=0;jake<13;jake++)
        {
            if (RTStr[jake]==32)
            {
                RTStr[jake] = 48;
            }
        }
    }
RTStr[jake]=48;
}
}
jake =0;
for (jake=0; jake<3; jake++) // for all 3 velocity commands
// (x,y,theta), sum the number
{
    ellenTemp[jake]=((RTStr[jake+(jake*3)]-48)*100)+((RTStr[jake+1+(jake*3)]-48)*10)+((RTStr[jake+2+(jake*3)]-48));
    if (ellenTemp[jake]<256 && ellenTemp[jake] >-1) // if number between 0 // and 255
    {
        RemoteControlVars[jake]=ellenTemp[jake]; // record number
    }
} 
ellenTemp[3]=RTStr[12]-48; // for forth command (PushButton), check if 0 // or 1
    {
        RemoteControlVars[3]=ellenTemp[3]; // record number
    }
    convertingString = FALSE; // allow RTStr re-write
}
/*
***********************************************************************************************
TASKS
***********************************************************************************************
*/
/*
***********************************************************************************************
TASK 1
***********************************************************************************************
*/
// Joystick read ADC
void DoTask1(void)
{
    byte value;
    byte redCode;
    byte io;
    int ji=0;

    for (ji=5;ji>0;ji--) // shift array by one
    {
        xaxist[ji]=xaxist[ji-1];
        yaxist[ji]=yaxist[ji-1];
        zaxist[ji]=zaxist[ji-1];
    }

    PushButton = Button_GetVal(); // check dead man button

174
ModeSwitch = ModeSwitch1_GetVal(); // check Mode Switch

// set this condition to send a 0 PWM signal to motor control, apply motor
// brakes,
// and reset error sum to 0. (see MotorControl function)
if(PushButton || ((ModeSwitch==64) && (RemoteControlVars[3]==1)))
{
    MotorsOn = TRUE;
} else
{
    MotorsOn = FALSE;
}

// check time since last succesful serial transmission
status[5]= FC321_GetTimeMS(&killTime);

while(ModeSwitch == TRUE && (killTime > 2000 || killTime < 0))
{
    // if time is unrealistic, check again until it is
    status[5]=FC321_GetTimeMS(&killTime);
}

// if serial stopped for sufficient time, shut off drive
if (killTime > MAX_SERIAL_LAG )
{
    RemoteControlVars[3]=0;
}

// One drive command system operational
if (((ModeSwitch == 64) && (RemoteControlVars[3]==1))||PushButton == 128)
{
    // Remote Desktop plugged in, Mode Switch switched ON, and Deadman
    // Variable ON
    if ((ModeSwitch == 64) && (RemoteControlVars[3]==1))
    {
        xaxist[0] = RemoteControlVars[0];
        yaxist[0] = RemoteControlVars[1];
        zaxist[0] = RemoteControlVars[2];
    }
    else if (PushButton == 128) // Manual Joystick plugged in and Deadman
    {
    }
}

// Button Pushed
{
    // Check for errors in A2D input and changes the pointer to refer to it
    // Obtain A/D for channel 0, joystick x-axis
    redCode = Joystick_GetChanValue8(0,&value);
    xaxist[0] =value;

    // Obtain A/D for channel 1, joystick y-axis
    redCode = Joystick_GetChanValue8(1,&value);
    yaxist[0] =value;

    // Obtain A/D for channel 2, joystick z-axis
    redCode = Joystick_GetChanValue8(2,&value);
\[
\text{zaxist}[0] = \text{value};
\]

// if commands are greater then almost zero velocity
if(!(abs(xaxist[0]-127)<8) && (abs(yaxist[0]-127)<8) && (abs(zaxist[0]-127)<8))
{
    for (io=1;io<6;io++) // sum all 6 variables in array
    {
        xaxist[0]=xaxist[0]+xaxist[io];
        yaxist[0]=yaxist[0]+yaxist[io];
        zaxist[0]=zaxist[0]+zaxist[io];
    }
    // set first variable to average of all 6 variables
    xaxist[0]=xaxist[0]/6;
    yaxist[0]=yaxist[0]/6;
    zaxist[0]=zaxist[0]/6;
}
else // preliminary crop under 8, send stop
{
    xaxist[0]=127;
    yaxist[0]=127;
    zaxist[0]=127;
}
else // no commands available, send stop
{
    xaxist[0]=127;
    yaxist[0]=127;
    zaxist[0]=127;
}

// convert axes to -255 --> 255, send commands to jacobian
xaxisd = 2* xaxist[0] - 255;
yaxisd = 2* yaxist[0] - 255;
zaxisd = 2* zaxist[0] - 255;

// TASK 2

/task2/
TASK 3

Serial communication, printing variables to serial for debugging and monitoring

void DoTask3(void)
{
    int numSent;
    word numSentS;
    // send a motor desired speed and measured speed every time this function is called
    // convert to integer for faster transmission, sending floats causes program to crash
    numSent = sprintf(Out2Data,"M%i,%5i,%5i\r\n",outCount,(int)(DesiredSpeed[outCount]*1000),(int)(encSpeed[outCount]*1000));
    sendingData = TRUE;
    status[0] = Serial_SendBlock(Out2Data,numSent,&numSentS);
    sendingData=FALSE;
    outCount++; // switch to next motor for next iteration
    if (outCount>3)
    {
        outCount=0; // if too big, set to motor 0
    }
}

 TASK 4

Calculate the actual speed of each motor using the encoder counts and do PID speed control

void DoTask4(void)
{
    ReadEncsAndSpeedControl();
}
/*
 ******************************************************************************************

*******
INITIALIZE TASKS
******************************************************************************************

******* */

// These functions reset the task values to their maximum
void InitTask1(void)
{
    task1val=TASK1_MAX;
}

void InitTask2(void)
{
    task2val=TASK2_MAX;
}

void InitTask3(void)
{
    task3val=TASK3_MAX;
}

void InitTask4(void)
{
    task4val=TASK4_MAX;
}

// this function called at beginning of program
void InitTask(void)
{
    task1val = TASK1_MAX;
    task2val = TASK2_MAX;
    task3val = TASK3_MAX;
    task4val = TASK4_MAX;
}

/*
 ******************************************************************************************
 RUN TASKS
 ******************************************************************************************
 */

// Function that checks the task values and executes a task when its value is
// 0,
// after execution is complete that task's value is reset to its Max value
void DoAllTasks(void)
{
    if(task1val<1)
    {
        DoTask1();
InitTask1();
}
if(task2val<1)
{
    DoTask2();
    InitTask2();
}
if(task3val<1)
{
    DoTask3();
    InitTask3();
}
if(task4val<1)
{
    DoTask4();
    InitTask4();
}
}

/*
***********************************************************************
EVENTS- Interrupt Service Routines
***********************************************************************
*/

/*
** ==============================================================
** Event : Joystick_OnEnd (module Events)
**
** From bean : Joystick [ADC]
** Description :
**    This event is called after the measurement (which
**    consists of <1 or more conversions>) is/are finished.
** Parameters : None
** Returns : Nothing
** ==============================================================
*/
void Joystick_OnEnd(void)
{
    /* Write your code here ... */
}

/*
** ==============================================================
** Event : Serial_OnRxChar (module Events)
**
** From bean : Serial [AsynchroSerial]
** Description :
** ==============================================================
*/
** This event is called after a correct character is received.
** DMA mode:
** If DMA controller is available on the selected CPU and the receiver is configured to use DMA controller then this event is disabled. Only OnFullRxBuf method can be used in DMA mode.
** Parameters : None
** Returns : Nothing
** ===================================================================
*/
void Serial_OnRxChar(void)
{
    if (convertingString==FALSE) // if no conversion is happening
    {
        status[2] = Serial_RecvChar(&SerialIn[serialCount]); //receive the character
        status[4]= FC321_Reset(); // reset serial kill timer
        if(SerialIn[serialCount]=='!') // if the character is an "!
        {
            serialCount=0; // reset the string
        }
        else if(SerialIn[serialCount]==13) // if the character is the "/n"
        {
            ConvertSerialToCommands(&SerialIn); // Convert the string to commands
            serialCount=0; // reset the string
            status[3]= Serial_ClearRxBuf(); // clear the buffer
        }
        else
        {
            serialCount++; // increment the string by 1
        }
    }
}

/*
** Event : TI1_OnInterrupt (module Events)
** From bean : TI1 [TimerInt]
** Description :
** When a timer interrupt occurs this event is called (only when the bean is enabled - "Enable" and the events are enabled - "EnableEvent").
** Parameters : None
** Returns : Nothing
** ===================================================================
*/
void TI1_OnInterrupt(void)
{
    // Decrease the count by 1 for every 100 microseconds counted
}
task1val--;  
task2val--;  
task3val--;  
task4val--;  
countVal--;  // for encoder counters
}

/*
 **  ==============================================================
 **  Event    :  Overflow_OnInterrupt (module Events)
 **  From bean : Overflow [ExtInt]
 **  Description:
 **      This event is called when an active signal edge/level has
 **      occurred.
 **  Parameters : None
 **  Returns    : Nothing
 **  ==============================================================
 */
//void Overflow_OnInterrupt(void)
//{
//}

/*
 **  ==============================================================
 **  Event    :  Serial_OnFullRxBuf (module Events)
 **  From bean : Serial [AsynchroSerial]
 **  Description:
 **      This event is called when the input buffer is full.
 **  Parameters : None
 **  Returns    : Nothing
 **  ==============================================================
 */
void Serial_OnFullRxBuf(void)  
{
}

/*
 **  ==============================================================
 **  Event    :  WDog1_OnWatchDog (module Events)
 **  From bean : WDog1 [WatchDog]
 **  Description:
 **      This event is called whenever the watchdog starts "barking"
 **      (e.g., after a specified period of the last clearing).
 **  Parameters : None
 **  Returns    : Nothing
 **  ==============================================================
 */
#pragma NO_FRAME
#pragma NO_EXIT

void WDog1_OnWatchDog(void)
{
  /* Write your code here ... */

  /*** The following line was generated by Processor Expert. DON'T MODIFY
  THIS CODE!!! ***/
  __asm jmp _EntryPoint; /* Jump to regular startup code */
}

/* END Events */

/*
** ###################################################################
** This file was created by UNIS Processor Expert 2.96 [03.76]
** for the Freescale HCS12 series of microcontrollers.
**
** ###################################################################
*/
D.3 omnibotPinger.cpp

/* Omnibot ROS Program
* omnibotPinger.cpp
* Created by: Steven Bemis
*
* Subscribes to: pong_bus, joyChatter
* Publishes: ping_bus
* Requires: Master node (roscore)
* 
* Some functionality borrowed from ROS tutorials
* 
* This program is run in conjunction with omnibotPonger.cpp to allow
* for back and forth communication. When a message is received, a
* message is sent. The published topic is the commands received from
* the subscribed topic joyChatter. The receiving of the data is
* de-coupled from the sending and receiving of data between the
* pinger and ponger nodes. If no data is received within a certain
* period of time, the command is changed to a shutoff command which
* which serves as a safety pre-caution. Data received from pong_bus
* is displayed to the user.
*/

#include <iostream>
#include "ros/ros.h" //All roscpp nodes will need this
#include "pingpong_cpp/PPBall.h" //This is our autogenerated msg
#include "time.h"
#include "std_msgs/String.h"

#include<string>
#include<sstream>

class Pinger
{
public:
    std::string msg;
    std::string transferMsg;
    std::string msg_; // A string containing our message to echo
    double freq_; // The frequency of operation.

    ros::NodeHandle n_; // A string containing our message to echo
    ros::Publisher ping_pub_; // The frequency of operation.
    ros::Subscriber pong_sub_; // A string containing our message to echo
    ros::Subscriber joy_sub_; // The frequency of operation.

    std::string dirString;
    std::ostringstream oss;
    int killCount;

    // function called at program start-up
    void init()
    {
        // Retrieve internal message parameter, or else set default to 'ping!' 

n_.param("message", msg_, std::string(""));
// Retrieve global freq parameter or else set default to 1.0
n_.param("freq", freq_, 1.0);
// Advertise our output with a subscription callback
// Pinger::serve will be called whenever a new node connects
ping_pub_ = n_.advertise<pingpong_cpp::PPBall>("ping_bus", 10,
boost::bind(&Pinger::serve, this, _1));
// Subscribe to the pong bus
pong_sub_ = n_.subscribe("pong_bus", 10, &Pinger::pongCallback, this);
// Subscribe to keyboard/joystick input
joy_sub = n_.subscribe("joyChatter", 1000, &Pinger::chatterCallback,
this);
// initialize variables
killCount=0;
}

// function called when message received from topic joyChatter
void chatterCallback(const std_msgs::StringConstPtr& msg)
{
  //ROS_INFO("I heard: [%s]", msg->data.c_str());
  // store command data
  transferMsg = msg->data.c_str();
  // reset killCount timer
  killCount=0;
}

// function called when message received from topic pong_bus
void pongCallback(const pingpong_cpp::PPBallConstPtr& in)
{
  pingpong_cpp::PPBall out; // Our output msg
  // print received message to screen
  ROS_INFO_STREAM("Received msg " << in->counter << ": " << in->msg);
  // increase killCount timer by 1
  killCount++;
  // if killCount exceeded limit, set commanded velocities to zero
  // if not, retrieve commands from joyChatter
  if (killCount >50)
  {
    oss << "127,127,127,0";
  }
  else
  {
    oss << transferMsg;
  }
  // convert commands to publishable data
  std::string dirString(oss.str());
  oss.str("");
  out.msg.append(dirString);
  // increment counter
  out.counter++;
  // print message to be sent to screen
  ROS_INFO_STREAM("Sending msg " << out.counter << ": " << out.msg);
  // Publish output
ping_pub_.publish(out);
}

// Function called after initialization to begin
// transfer of data between ping_bus and pong_bus
void serve(const ros::SingleSubscriberPublisher& pub)
{
    pingpong_cpp::PPBall out; // Our output msg

    ROS_INFO("Sending initial message");
    out.msg = msg_;
    out.counter = 1;

    // This will publish only to the Node that just connected to us
    pub.publish(out);
}

int main(int argc, char **argv)
{
    // Initialize ros
    ros::init(argc, argv, "pinger");

    // Create a new instance of pinger
    Pinger p;
    // initialize node
    p.init();

    // Wait for pinger to finish
    ros::spin();

    // Done
    ROS_INFO("Pinger is done");
    return 0;
}
# D.4 omnibotPonger.cpp

/* Omnibot ROS Program
   * omnibotPonger.cpp
   * Created by: Steven Bemis
   *
   * Subscribes to: ping_bus
   * Publishes: pong_bus
   * Interface: Serial (ttyUSB0)
   * Requires: Master node (roscore)
   *
   * Some functionality borrowed from ROS tutorials
   *
   * This program is run in conjunction with omnibotPinger.cpp to allow
   * for back and forth communication. When a message is received, a
   * message is sent. The published topic is the data received from
   * the microcontroller. When data is received from the ping_bus, it is
   * formatted and sent as drive commands to the Omnibot microcontroller.
   */

#include <iostream>
#include "ros/ros.h" //All roscpp nodes will need this
#include "pingpong_cpp/PPBall.h" //This is our autogenerated msg
#include "time.h"

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>
#include <string>
#include <sstream>

class Ponger
{
public:

  struct termios options;
  int fd1;
  int started;
  int ser_count;
  char buffClear[10000];
  int wr, rd;
  std::string outMessage;
  std::string msg_; // A string containing our message to echo
  double freq_; // The frequency of operation.

  ros::NodeHandle n_;  
  ros::Publisher pong_pub_;  
  ros::Subscriber ping_sub_;  

  std::ostringstream oss;
// function called upon program launch
void init()
{
    // Advertise our output
    pong_pub_ = n_.advertise<pingpong_cpp::PPBall>("pong_bus", 10);
    // Subscribe to the ping bus
    ping_sub_ = n_.subscribe("ping_bus", 10, &Ponger::pingCallback, this);
    // Retrieve internal message parameter, or else set default to 'pong!'
    n_.param("message", msg_, std::string("pong!"));
    // Retrieve global freq parameter or else set default to 1.0
    n_.param("freq", freq_, 1.0);
    // set settings for serial port communication
    system("stty -F /dev/ttyUSB0 115200 cs8 -cstopb -parity -icanon hupcl
    -crtscts min 0 time 0");
    // attempt to open serial port (ttyUSB0)
    fd1=open("/dev/ttyUSB0",O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd1 == -1) // if error
    {
        perror("open_port: Unable to open /dev/ttyUSB0 ");
    }
    else // if no error
    {
        fcntl(fd1, F_SETFL, 0);
        printf("Port 1 has been successfully opened and %d is the file
description\n",fd1);
        rd=read(fd1,buffClear,10000);
    }
    // initialize variables
    ser_count=0;
    started=0;
}

// function called everyday data is received from published topic
// ping_sub_
void pingCallback(const pingpong_cpp::PPBallConstPtr& in)
{
    // variable declaration and reset
    char JoyStickData[15];
    char buff[100];
    // read in data from published topic ping_sub_
    const char * WirelessMsgIn = in->msg.c_str();
    int len;
    char OmnibotData[100];
    rd=0;

    pingpong_cpp::PPBall out; // Our output msg

    // convert message to format acceptable for serial writing
    sprintf(JoyStickData, ".!R%s\n",WirelessMsgIn); // letter "R" serves
    // as a check key
    len = strlen(WirelessMsgIn);
    // print data to screen for user viewing
    printf("length: %d\n",len);
printf("Joystickdata: %s",JoyStickData);

//************** WRITING TO OMNIBOT CONTROLLER ***********
wr=write(fd1,JoyStickData,19);

//************** READING FROM OMNIBOT CONTROLLER**********
rd=read(fd1,buff,100);
   // check read in data, one character at a time
for(int i=0;i<rd;i++)
   {
      // if character is "M", indicate the start of the string has been
      // found
      if (buff[i] == 77)
      {
         started = 1;
      }  
      if(buff[i]==10 && started==1) // this happens when a complete
      // message has been received from omnibot
      {
         ser_count = 0;
         started =0;
         // print data to screen
         printf("\nreading message: %s\n", OmnibotData);
         // format data for publishing to other ROS nodes
         oss << OmnibotData;
         std::string outMessage(oss.str());
         oss.str("");
         // Append data to current outbound message
         out.msg.append(outMessage);
         //printf("Sending: %s\n",out.msg.c_str());
         // clear the read in string
         for (int j=0;j<100;j++)
         {
            OmnibotData[j] = 0;
         }
         break;
      }
      // store character to character string
      OmnibotData[ser_count+i]=buff[i];
   }
   if(buff[rd-1]!=10) // this happens when a complete message has been
   // received from omnibot
   {
       ser_count=ser_count+rd;
   }
   // publish the outbound message
   pong_pub_.publish(out);

   // when shutdown request detected, close port
   void shutdown()
   {
      close(fd1);
   }
};
// main program
int main(int argc, char **argv)
{
    // Initialize node
    ros::init(argc, argv, "ponger");

    // Create a new instance of ponger
    Ponger p;
    // Initialize node and variables
    p.init();

    // Wait for ponger to finish
    ros::spin();
    // when finished, shutdown node
    p.shutdown();

    // Done
    printf("omnibotPonger is done!");
    return 0;
}
/* Omnibot ROS Program
omnibotUI.cpp
Created by: Steven Bemis
*
* Subscribes to: none
* Publishes: joyChatter
* Interface: Serial (ttyS0)
* Requires: Master node (roscore)
*
* Some functionality borrowed from ROS tutorials
*
* This program serves as a ROS interface for a custom controlled
* joystick through serial communication. Joystick data is interpreted
* and sent to this node from a program run on a microcontroller.
* Commands are received and then published to topic joyChatter.
*/

#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>

int fd1;
char stuff[255], buff[13];
int wr, rd;

// main function
int main(int argc, char **argv)
{
  // initialize node
  ros::init(argc, argv, "joyTalker");
  ros::NodeHandle n;
  // Publish topic joyChatter
  ros::Publisher pub = n.advertise<std_msgs::String>("joyChatter", 1000);
  // declare and initialize variables
  int motorCom[4];
  motorCom[0]=0;
  ros::Rate r(10);

  // set settings for serial port
  system("stty -F /dev/ttyS0 115200 cs8 -cstopb -parity -icanon hupcl
  -crtսcs min 1 time 1");
  // attempt to open serial port
  fd1=open("/dev/ttyS0",O_RDWR | O_NOCTTY | O_NDELAY);
  if (fd1 == -1 ) // if error

{
    perror("open_port: Unable to open /dev/ttyS0 ");
}
else // if no error
{
    fcntl(fd1, F_SETFL,0);
    printf("Port 1 has been sucessfully opened and %d is the file
description\n",fd1);
}

// while shutdown not commanded (ctrl-c)
while (n.ok())
{
    rd=0;
    // read in current data until 13 characters have been collected
    while (rd < 13)
    {
        rd=read(fd1,stuff,255);
    }
    for (int i=0;i<13;i++)
    {
        buff[i]=stuff[i];
    }
    // print character array to screen
    printf("Bytes sent are \%d \n",rd);
    printf("reading: \%s \n", buff);
    // convert message to string for ROS node communication
    std::string msg;
    std::stringstream ss;
    ss << buff;
    ROS_INFO("%s", ss.str().c_str());
    msg.data = ss.str();
    // publish data under topic joyChatter
    pub.publish(msg);
}
return 0;
}