IMPLEMENTATION OF PARTICLE MODEL CONTROL APPROACH TO A FIXED AXLE UGV

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Nikhil Gupta

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Mechanical Engineering

July 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Implementation of Particle Model Control Approach

to a Fixed Axle

By

Nikhil Gupta

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

ABSTRACT

Gupta, Nikhil, M.S., Department of Mechanical Engineering, North Dakota State University, July 2010. Implementation Of Particle Model Control Approach To A Fixed Axle UGV. Major Professor: Dr. Majura F. Selekwa.

Robotic vehicles are normally modeled as rigid bodies under general motion, combining translation and rotation motions. While such modeling results in motion controllers that are easy to implement, these controllers are also limited in the number of degrees of freedom (DOF) that can be controlled. The robotic vehicle with limited DOF operates well in structured terrain conditions with sufficient stability and friction. When the vehicle is operated in unstructured terrains, such as those that are sandy, snowy, or steep terrains, which might be slippery, such an approach fails to operate well. Since additional applications of robotic vehicles are in unstructured terrains, it is important to find alternative control models that will increase the number of controllable DOF and add more robustness and flexibility to the vehicle's performance.

This thesis proposes the modeling of a vehicle as a system of particles centered at the wheels, with each particle controlled independent of one another in order to achieve the desired vehicle motion. In this work, the Particle Model Control approach was tested on the robotic platform BIBOT-1. The work illustrated the major vehicle kinematics under different steering modes and how the controls for the robotic motion can be formulated on the basis of Particle Modeling. A control system, based on Particle Modeling using decentralized control architecture, was designed and tested on BIBOT-1. The preliminary test results obtained from the trial runs were then analyzed on the basis of root mean square (R.M.S.) error performance factors. Some future work was also suggested in order to gather more results and validate the modeling approach.

ACKNOWLEDGEMENTS

I am deeply indebted to my academic advisor, Dr. Majura F. Selekwa, for his invaluable guidance and contributions, his understanding and patience, and his constructive comments and suggestion in pursuit of my M.S. degree.

I thank my advising committee members, Dr. Annie Tangpong, Dr. Mariusz Ziejewski, and Dr. Jacob Glower for their guidance and support in the completion of this thesis. Without their expert views and reviews, the thesis would be too informal piece of writing.

I also thank Dr. Alan Kallmeyer, chair of the department of Mechanical Engineering, for having confidence in me and providing the financial support to pursue my degree at North Dakota State University.

I sincerely thank all of the faculty, staff, and graduate students at Department of Mechanical Engineering for their help and for providing a nice and wonderful working environment.

Finally, my deepest gratitude goes to my family members: my parents, my sister and my cousin brothers. Their constant encouragement and support have always helped me to overcome the odds during my work. Without their unconditional love and sacrifice, I would not have been able to come so far to continue my studies.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

1. ROBOTS: PAST, PRESENT AND FUTURE

Robots were introduced as a way of relieving humans of jobs that are characterized as dirty, dangerous, and dull (DDD), including jobs that seems to be dangerously inconceivable. Sooner or later, almost every known field such as manufacturing, service, or domestic sector has involved some kind of robot technology [1]. The quest for a risk free and safe work environment for humans has been the driving force behind the evolution of robot technology. This chapter starts by providing a brief overview on the evolution of robotic technologies and their applications and it closes by describing the motivating problem that forms the basis of this thesis.

1.1. Early Developments

In early developments, robots were deployed in industrial operations only; hence, until the 1990s, most of the robotics research was dominated by industrial robots [2], [3]. Some of the early industrial robots were robotic manipulators and guided mobile vehicles. A robotic manipulator is like an arm, consisting of number of links and an end effecter that performs the required task; whereas a guided vehicle is a mobile platform that moves along a predefined path only. In general, these early industrial robots were not flexible enough to cater unpredictable dynamic work environments such as those occurring in most industrial chemical processes [4] and in postal mail handling systems [5]. Instead, they were programmed to rigidly follow predefined work cycles [6]. The success of industrial robots in the 1970s, and in the 1980s, fuelled the interest for application of robots in many other sectors including those characterized by dynamic work environments. This led to the development of more flexible robots in fields such as medical, entertainment, domestic, military and in space exploration missions, among others [7-10].

Flexible robots are characterized by the presence of an improved intelligence that enables them to make reasonable decisions even under unpredictable environments. The two main types of flexible robots that evolved under this development were the autonomous mobile vehicles and the anthropomorphic robots, which are presently known as humanoids [11-14]. The traditional manipulators became parts of these flexible robots where they were employed as robotic arms. Although some of these flexible robots were employed just like the traditional industrial manipulators, by relieving human beings of DDD tasks, for example in rehabilitation [15] and firefighting tasks [16], a significant number of flexible robots were also employed as assistants to human beings in performing certain high skilled jobs. For example, medical robots were employed as assistants to doctors and nurses in performing special surgical operations [17].

Humanoids are robots that have a similar structure as that of human beings with a highly developed artificial intelligence. Because of their human-like physical structure, they tend to improve the Robot-Human interaction in the social world [18]. Today, there is an increased interest in these robots especially in the entertainment [19] and service industry [20]. Despite recent developments in this area of robotics, there are still a great number of research problems to be addressed in the control systems, structural design and advanced intelligence. Typical problems include dexterous motion generation, task planning, humanlike flexibility in structural design and advanced intelligence to better interact with human beings such as in social learning, emotion expression and perception.

Alongside the interest in the growth and development of humanoids, there was an increased interest in autonomous mobile robots also. An autonomous mobile robot is a robotic system consisting of a mobile platform with locomotive elements that can move from one location to another location in space [21]. Such robots can be terrestrial, e.g., the National Aeronautics and Space Administration's (NASA) Sojourner Mars Rover [22], aquatic, e.g., the National Oceanography Centre's Autosub6000 [23], or aerial e.g., the USAF's Predator [24], [25]. In comparison to humanoids, the autonomous mobile robots are simple in structure and in operation, but with a great deal of potential applications. The locomotive elements in the autonomous mobile robots for the aquatic and aerial environments are usually either propellers or screws, although legs can also be used at the seabed. Depending on the type of the terrain, the locomotive element for terrestrial robotic vehicles can be a wheel, track, or leg. Due to the complexity and unpredictability of the terrestrial environments, the selection of an appropriate locomotive element for a particular terrestrial robot is in itself a challenge. Wheels work well and provide high maneuverability in structured terrains, but miserably fail in unstable terrains such as steep, slippery and irregular terrains; whereas tracks work well in all types of terrains, but with limited maneuverability. On the other end, legs can traverse all types of terrains but are relatively slow as compared to wheels and tracks. Despite their disadvantages in unstructured terrains, wheels are still among the highly preferred locomotive elements. They have an advantage of being fast in response, easy in steering and employ relatively simple control systems compared to the other methods.

1.2. Wheeled Ground Robotic Vehicle: Present State

1.2.1. Structure

Autonomous wheeled ground robotic vehicles (also known as Unmanned ground Vehicles [26]) are just like other ordinary ground vehicles with either two or more wheels and are moved by turning the wheels accordingly. The basic structure of a wheeled robot consists of a drive mechanism and a steering control. Although these robots are structurally simple, their control architecture can be relatively complex in comparison to those of other robotic vehicles. The control system, in general, comprises of a low level unit for controlling actuators and a high level unit for controlling the overall kinematics of the vehicle as one body; the high level unit is also responsible for all decision making and command generation tasks of the robot. Depending on the complexity of the robotic system itself and the flexibility in the actuator layout, the high level and the low level controls can be built either as a single integrated unit or as separate independent units, in which case the lower units serve as subordinates of the higher level unit.

1.2.2. Actuation

The main actuators that are responsible for the motion of any robotic vehicle are the traction unit for turning the wheels to cause motion and the steering unit to change the direction of the robot motion. For a typical four wheel vehicle, the structure of the traction control unit depends on whether the drive system is a Two Wheel Drive system (Front or Rear) or a Four Wheel Drive system (All Wheel Drive). In the two wheel drive system, either the front or the rear axle wheels are driven; whereas in the four wheel drive system all of the wheels are driven. Furthermore, these drive systems can have differentially driven wheels [27] on the same axle or independent driven wheels. If the drive is differentially driven, then the traction control becomes relatively easy since the mechanical differential unit tends to equalize the torque applied on each wheel. In the independent wheel drive system, however, the traction control tends to be relatively complex, though it provides more output power and flexibility to the system than the mechanically coupled differential drive system.

The structure of the steering control unit also depends on the type of steering mechanism used to steer the robot. Typical steering mechanisms for a rigid chassis include the skid steering mechanisms [28], the Ackerman steering mechanism [29], [30], and the all wheel independent steering mechanisms [29]. The skid steering action is performed by applying different traction speeds on the wheels at both sides of the robot. The effectiveness of this method depends on the longitudinal and lateral dimensions of the robot. If these dimensions are relatively short then zero turn radius can be achieved, which is very useful in congested areas. However, the differential steering tends to fail miserably if the surface traction is not uniform on the wheels at both sides of the robot, thus compromising the vehicle stability. Additionally, the drive efficiency of the differential steering decreases due to the difference in the power requirement at both wheels, which also limits the speed of the vehicle. Because of these limitations, the differential steering is confined to toy robots or small sized robotic vehicles such as lawn mowers. The Ackerman steering mechanism is the most common steering configuration used in modern cars and trucks. This system is also used in the large-sized robotic vehicles powered by IC engines such as the Experimental Unmanned Vehicle (XUV) being developed by the Army Research Lab [31]. In this steering configuration, the rear wheels may be kept fixed relative to the body frame whereas the front wheels are moved simultaneously relative to each other. The steering effect is produced by changing the direction of rotation of the steered wheels. The Ackerman steering system is simple in its construction and easy to control but it requires space for the wheels to swivel. The other major disadvantage of this steering geometry is that it forces the fixed non-steered wheels to follow the steered wheels, thus producing a side drag that leads to vehicle instability in sharper turns. The all wheel steering system is a modern approach in which all wheels of the vehicle can swivel either simultaneously or independently. Thus a vehicle can traverse in all direction of the plane, providing great maneuverability.

1.3. The Future Of Robotic Vehicles And Motivation

1.3.1. The Future of Robotic Vehicles

Developments in mobile robot technologies have been helpful in many ways, making the world a better and safer place to live. Robots have marked great future potential that can be utilized to completely eliminate human intervention from highly hazardous fields. As more and more improvements in robot technologies evolve, new demands for robot applications will also evolve in all work levels, ranging from the semi-skilled to high-skilled work fields. It is anticipated the futuristic robot will be able to intelligently share the work environment with humans in a more safe and convenient manner. Despite rapid improvements in the current levels of robotic technologies, there are still many challenges that will need to be solved before the bright future of robots can be realized. Among problems that must be solved include control and navigation [32], [33], and perception and intelligence [34-36]. Furthermore, there is an increased interest in making future robots as small as possible without compromising their performance. Among the current directions of robotic research to address these problems include biologically inspired robots, which will be richly equipped with sensors and have the power of making intelligent judgments, risk analysis and can react to unpredicted events [37].

Another major problem to be solved is that of the mobility and manipulation in difficult environments such as cluttered environments, environments with moving obstacles, and in unpredictable terrains such as steep, irregular, unstable and slippery terrains. Presently, the mobility problem is seriously pursued for defense and space exploration robots, which are more likely to encounter such environments in their daily missions. As intelligence, control, mobility, and manipulation problems will be addressed on one side, the other side of robotics research will focus on the development of new materials and manufacturing methods that will lead to small sized and light but powerful robots.

1.3.2. Motivation

Multidisciplinary group effort is needed in tackling the research problems for the futuristic robot; however, Mechatronics will play a major role. One of the problems to be addressed by Mechatronics is that of robot mobility in unstable terrains. This research is motivated by the mobility problem. Poor robot mobility capability has been a limiting factor in the application of robot potential in the most interesting field such as defense and space exploration. Both the US Department of Defense (DoD) and NASA have invested a lot in robotics research to develop mobile robot technology that can work in cluttered, unstructured and unknown terrains with or without the presence of humans [38], [39].

The mobility of the wheeled vehicles on unstable and slippery surfaces, such as snowy surfaces, is a serious challenge since the wheels tend to lose the grip on the surface. The loss of grip on the surface causes the traction as well as steering control failure. Many different systems have been developed so far that include the Anti Braking System (ABS) and the Traction Control System (TCS), which are used to increase the control of the vehicle during the rough conditions. An ABS prevents the wheels from locking up during braking by adjusting the braking pressure relative to the vehicle's speed [40] thus maintaining the steering control, while the TCS optimally distributes the traction force on tires using the differences in friction between the tires and the road surface [41-45]. The introduction of such safety systems have resulted in the low rates of road accidents [46], [47] but have proved to be redundant on slippery surfaces where the ABS excessively increases the braking distance while the TCS decreases the vehicle acceleration [48]. These limitations have made the use of such systems on the robots less common.

2. RESEARCH HYPOTHESIS

The failure of vehicles traversing on slippery surfaces is not just caused by the lack of proper terrain detecting methods, but is also due to the lack of appropriate mobility control methods. Since the robot mobility and kinematics depends on the actuator dynamics, it is important to have the efficient and better control units that can handle the actuator dynamics in order to achieve the desired robot kinematics. Some of the recent research results have shown that by using a control system that optimally coordinates the traction and steering functions, the mobility of the wheeled robotic vehicle can be significantly improved [49], which provides hope for using wheeled robotic vehicles in difficult terrains.

It is the view of some researchers that the mobility problem in these terrains conditions can be improved by imitating the mechanism of motions of the biological system such as reptiles [50], [51] and mammals. Biological systems are known to have an infinite number of DOF that help them adapt to different terrain conditions without significantly compromising their speed and control. Unfortunately the structural rigidity and the current approaches in the development of the robot motion controllers have limited the number of controllable DOF, making it hard to imitate the biological systems. Normally, the robot is modeled as a rigid body [52], which limits the available controllable DOF for each wheel to only two, corresponding to the steering angle and the traction force.

It is thought that one method of improving the mobility of robotic vehicles would be to increase the wheels' controllable DOF to beyond two. This approach is possible by modeling the robot as a system of particles centered at the wheels such that the relative positions between the wheels can also be controlled, i.e., the Particle Modeling approach.

Since the relative position of the wheel adds another three DOF in a three dimensional (3-D) space, the net effect of this approach would be five controllable DOF for each wheel. Thus the Particle Modeling approach is likely to be more effective in controlling the motion of robotic vehicles because of its added DOF, however, this approach has not been tested before.

The hypothesis of this research is that, by modeling the vehicle as a system of independently controlled particles, it is possible to drive the robotic vehicle to effectively track the desired paths. The primary objective of this thesis is to experimentally test the applicability of this approach using a standard four wheel fixed axle robotic vehicle as a starting point. The results of this thesis should offer a clue as to whether it is feasible to proceed with this approach on more complex robotic vehicles with adjustable axles and how to implement it. The next chapter discusses the kinematics of a system of particle and the feasibility of the Particle Model Control approach and its implementation on a fixed axle robotic vehicle.

3. PARTICLE MODELING AND PROBLEM STATEMENT

The Particle Modeling approach proposed in the work for a robotic control design assumes that the wheels of the robot are a system of particles arranged in space. These particles are identical yet independent to each other, sharing a main goal, i.e. to traverse the system on the desired track. This facilitates to increase the vehicle's controllable DOF that can help improve the vehicle's mobility on various terrains. Thus, the main goal of the thesis is to design and implement a control system for a robot which is modeled based on a Particle Model approach; it will be implemented on an existing fixed axle robotic vehicle.

3.1. Particle Kinematics

From elementary mechanics [53], it is known for a given system of n particle, each with mass m_k and position vector $\vec{r}_k$ as illustrated in Figure 1, that there is a relation between the position of the individual particles and the position vector of the center of mass position $\vec{r}_G$ as

$$\vec{\boldsymbol{r}}_k = \vec{\boldsymbol{r}}_G + \vec{\boldsymbol{r}}_{k|G}\,, \tag{3.1}$$

where, $\vec{r}_G = \frac{\sum_i m_i \vec{r}_i}{\sum_i m_i}$, $\sum_i m_i \vec{r}_{k|G} = 0$, and $\vec{\boldsymbol{r}}_{k|G}$ = relative position of k$^{\text{th}}$ particle with respect to the center of mass.

Similarly, the velocity $\vec{v}_k$ of the particle is expressed as

$$\vec{\boldsymbol{v}}_k = \vec{\boldsymbol{v}}_G + \vec{\boldsymbol{v}}_{k|G}\,, \tag{3.2}$$

where $\vec{v}_G$ is the velocity of the center of mass, and $\vec{v}_{k|G}$ is the relative velocity of the particle k with respect to the center of mass. If the positions can be resolved in a Cartesian plane then

$$\vec{\boldsymbol{r}}_k = \begin{bmatrix} \boldsymbol{r}_k \cos\boldsymbol{\theta}_k \\ \boldsymbol{r}_k \sin\boldsymbol{\theta}_k \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_G \cos\boldsymbol{\theta}_G \mp \boldsymbol{r}_{k|G} \cos(\boldsymbol{\theta}_G \pm \emptyset_k) \\ \boldsymbol{r}_G \sin\boldsymbol{\theta}_G \pm \boldsymbol{r}_{k|G} \sin(\boldsymbol{\theta}_G \pm \emptyset_k) \end{bmatrix}, \tag{3.3}$$

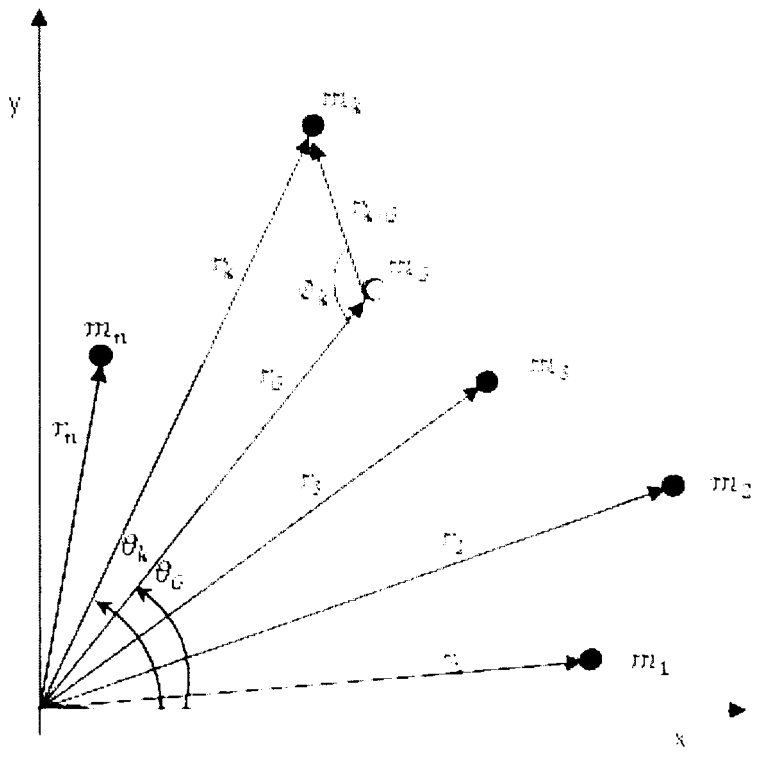


Figure 1: A System of Particles.

where and are magnitudes for the position vectors of particle and the center of mass, respectively and

. **(3.4)**

The angles of vector in the - plane satisfies

, **(3.5)**

. **(3.6)**

If and are angles of the velocity vector directions relative to the - axis, i.e.,

— , **(3.7)**

— , **(3.8)**

and the angle between $\vec{r}_{k|G}$ and the Cartesian plane be defined as $\emptyset_{k|G}$, where

$$\emptyset_{k|G} = \theta_G \pm \emptyset_k \,, \tag{3.9}$$

then the velocity of particle k can also be expressed in the Cartesian plane as

$$\vec{v}_k = \begin{bmatrix} v_k \cos\varphi_k \\ v_k \sin\varphi_k \end{bmatrix} \tag{3.10}$$

$$= \begin{bmatrix} v_G \cos\varphi_G \\ v_G \sin\varphi_G \end{bmatrix} \pm \begin{bmatrix} \dot{r}_{k|G} \cos\emptyset_{k|G} \pm r_{k|G}(\dot{\emptyset}_{k|G}) \sin\emptyset_{k|G} \\ \dot{r}_{k|G} \sin\emptyset_{k|G} \pm r_{k|G}(\dot{\emptyset}_{k|G}) \cos\emptyset_{k|G} \end{bmatrix}. \tag{3.11}$$

At any positions $\vec{r}_k$ and $\vec{r}_G$, this equation shows that if the velocity $\vec{v}_G$ of the center of mass is known, i.e., the magnitude v_G and the direction φ_G with respect to the x- axis, then the velocity $\vec{v}_k$ of each particle can be determined by using $\dot{r}_{k|G}$ and $\dot{\emptyset}_k$. Note that the knowledge of $\vec{v}_G$, $\vec{r}_k$ and $\vec{r}_G$ along with equations (2.1)-(2.9) provides all information required to determine $\vec{v}_k$ in equation (2.11). At constant velocity, the trajectory of the center of mass becomes

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_o \\ y_o \end{bmatrix} + \int \begin{bmatrix} v_G \cos\varphi_G \\ v_G \sin\varphi_G \end{bmatrix} dt \,, \tag{3.12}$$

$$= \begin{bmatrix} x_o \\ y_o \end{bmatrix} + \int \begin{bmatrix} v_k \cos\varphi_k \\ v_k \sin\varphi_k \end{bmatrix} \mp \begin{bmatrix} \dot{r}_{k|G} \cos\emptyset_{k|G} \pm r_{k|G}(\dot{\emptyset}_{k|G}) \sin\emptyset_{k|G} \\ \dot{r}_{k|G} \sin\emptyset_{k|G} \pm r_{k|G}(\dot{\emptyset}_{k|G}) \cos\emptyset_{k|G} \end{bmatrix} dt \,, \tag{3.13}$$

and the orientation of the center of mass with respect to the Cartesian plane becomes

$$\varphi_G = \varphi_{G_o} + \int \dot{\varphi}_G dt \quad , \tag{3.14}$$

$$= \tan^{-1}\left(\frac{\dot{r}_G}{r_G \dot{\theta}_G}\right) + \theta_{G_o} + \int (\dot{\emptyset}_{k|G} \mp \dot{\emptyset}_k) dt \,. \tag{3.15}$$

The basic kinematic equations, i.e., (3.1)-(3.15) defined under Particle Modeling of a system, can be applied in describing the motion of flocks [59-61] as well as of multi-robot formation and cooperation [62-66]. However, the use of these equations can increase the number of controlled DOF, thereby improving the robot performance, which has never been considered for describing and controlling robotic motions. The following work investigates the applications of this Particle Model approach in controlling a robotic vehicle.

3.2. Particle Model Approach To Vehicle Motion Control

Based on the particle kinematics, a -wheeled vehicle can also be modeled as a system of -particles corresponding to the number of wheels; each particle is assumed to be concentrated at the center of each wheel, i.e., a four wheeled vehicle can be treated as a system of four particles as shown in Figure 2.

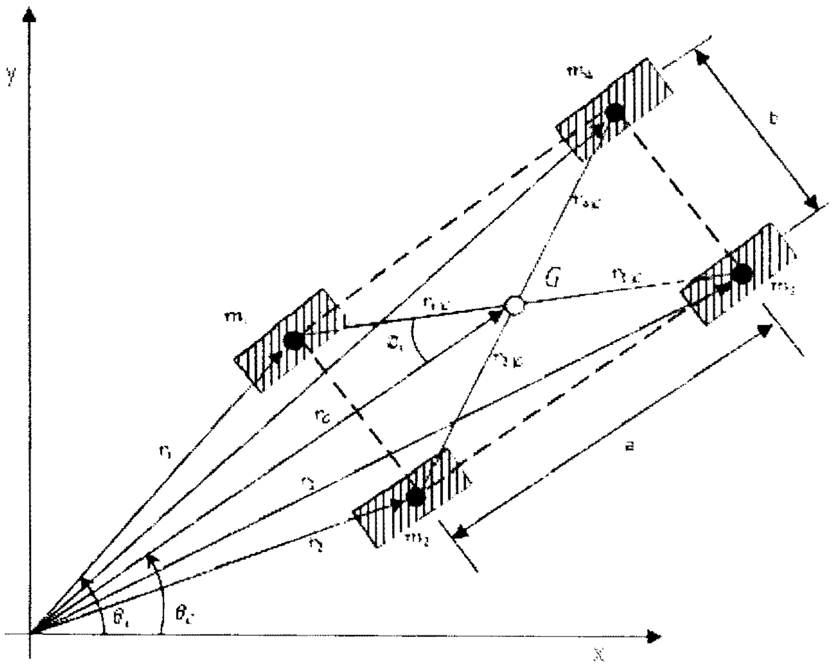


Figure 2: The Particle Model of a Four-Wheel Vehicle.

The motion of the vehicle can be controlled using various present control algorithms, while the overall control system can be illustrated by Figure 3. Here the navigation control system on the vehicle provides the navigational information about the speed of the center of mass, v_G, and its orientation in space, $\emptyset_G$. Along with the navigational information, the current motion state parameters monitored by the system are combined together to generate the control parameters for each wheel treated as a particle. By using equations (2.10) and (2.11) and the combined information, i.e., the navigation and the current motion parameters, the robot motion control algorithm determines the individual velocity for each wheel, $\vec{v}_k$. The wheel velocity vector, $\vec{v}_k$, is then processed by the traction controller and the wheel steering controller to achieve the wheel speed of magnitude v_k and the wheel heading direction φ_k. The continued motion of all wheels each with the velocity vector $\vec{v}_k$ will move the center of mass of the vehicle at a velocity $\vec{v}_G$ as demanded by the navigation algorithm.

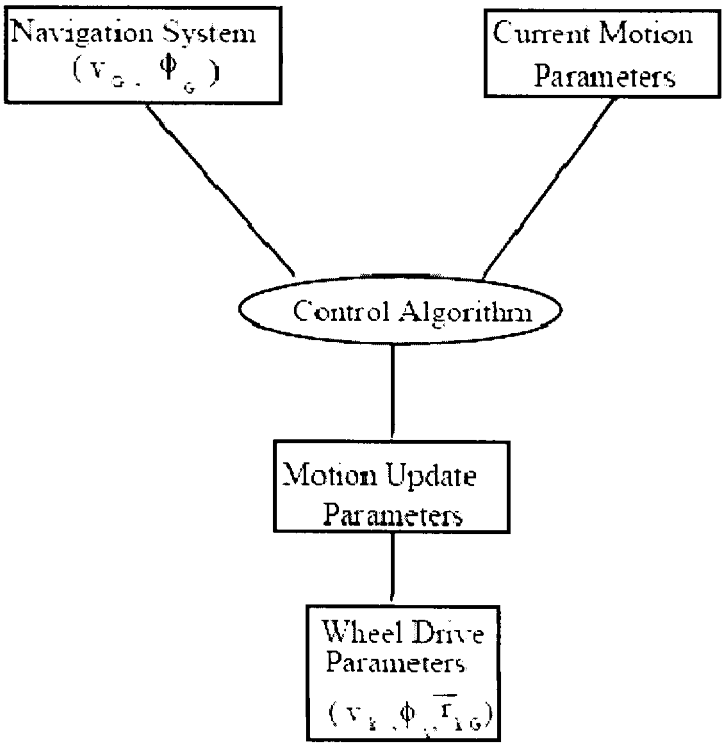


Figure 3: Five-Degrees of Freedom (DOF) Wheel Control Process

If the distance $r_{k|G}$ between each wheel k and the center of mass are not constant, i.e., it can be changed by the control system so that $\dot{r}_{k|G} \neq 0$, then it will create three more control parameters. In addition to the v_k and φ_k, the motion update parameters for each wheel would include $\vec{r}_{k|G}$, which creates a total of five DOF.

3.3. Particle Model Control Approach On A Fixed Axle Robotic Vehicle

When this approach is applied on a fixed axle four wheel vehicle where

$$\dot{r}_{k|G} = 0, \tag{3.16}$$

and

$$r_{k|G} = \frac{1}{2}\sqrt{a^2 + b^2} \quad , \tag{3.17}$$

then the updated motion velocity vector $\vec{v}_{k|G}$ becomes

$$\vec{v}_{k|G} = \pm \begin{bmatrix} r_{k|G}(\dot{\emptyset}_{k|G}) \sin \emptyset_{k|G} \\ r_{k|G}(\dot{\emptyset}_{k|G}) \cos \emptyset_{k|G} \end{bmatrix}, \tag{3.18}$$

with $\emptyset_k$ being a function of θ_G only. The vehicle thus reduces the available DOF to two only. This causes the vehicle to behave as a rigid body; however, the particle kinematics equations can still be applied for the control of its motion. There are five main steering motion configurations stated below for a four-wheeled vehicle that has fixed axles and all wheels steerable and drivable, to which particle modeling can be applied.

Main steering configurations for a four-wheeled vehicle of fixed axles are:

1. Fixed straight steering configuration,
2. Front (or rear) wheel turning steering,
3. All wheel steering general motion configuration,
4. All wheel steering translation motion (parallel) configuration, and
5. All wheel steering zero radius turning configuration.

3.3.1. Fixed straight steering configuration

The fixed straight steering refers to the configuration in which the wheels of the system are always aligned straight to the heading direction of the vehicle. For cornering the vehicle in this steering configuration, the skid steering [54], [55] is implemented; turning the left and the right side wheels at different speeds makes the vehicle turn in a direction depending on the differential velocity vector. In such a steering configuration, if the wheels move with the same forward velocity, then the vehicle will move in a straight line. However, if the speed of the wheels on one side increases as compared to the other side, then the vehicle follows a curved path inward towards the slower wheel. The geometrical configuration for steering is illustrated in Figure 4.

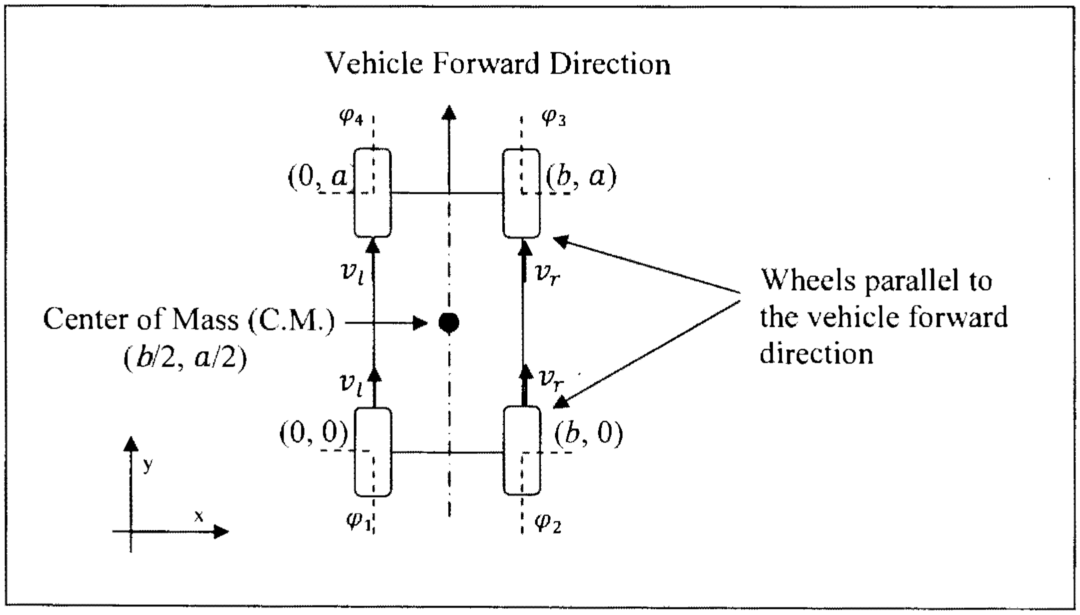


Figure 4: Fixed straight steering configuration.

In this configuration the left hand side wheels have speed $\vec{v}_l$ whereas the right hand side wheels have speed $\vec{v}_r$, such that

$$\vec{v}_1 = \vec{v}_4 = \vec{v}_l \ , \tag{3.19}$$

$$\vec{v}_2 = \vec{v}_3 = \vec{v}_r \,. \tag{3.20}$$

In order to move straight, the velocity $\vec{v}_l$ should be equal to $\vec{v}_r$, or the velocity of each wheel, $\vec{v}_k$, should be equal to each other with $\vec{v}_{k|G} = 0$ for all k such that at any instant

$$\vec{v}_k = \vec{v}_G \,. \tag{3.21}$$

For cornering the vehicle, differential velocity vector $\Delta\vec{v}$ between the left and the right side wheels is required such that,

$$\Delta\vec{v} = \pm(\vec{v}_r - \vec{v}_l) \,, \tag{3.22}$$

where the $\pm$ sign signifies the turning direction which can be either left or right; the velocity of the center of mass becomes

$$\vec{v}_G = \frac{\vec{v}_r + \vec{v}_l}{2} \,. \tag{3.23}$$

In this case, the change in the heading direction of the vehicle φ_G is equal to that of the wheels φ_k, i.e.,

$$\dot{\varphi}_G = \dot{\varphi}_k = \frac{2|\vec{v}_r - \vec{v}_l|}{|\vec{r}_{2|G} + \vec{r}_{3|G}| + |\vec{r}_{1|G} + \vec{r}_{4|G}|} = \frac{|\Delta\vec{v}|}{b} \,. \tag{3.24}$$

At a constant velocity, the trajectory of the center of mass becomes

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_o \\ y_o \end{bmatrix} + \frac{1}{2}\int \begin{bmatrix} (v_r + v_l)\cos\varphi_k \\ (v_r + v_l)\sin\varphi_k \end{bmatrix} dt \,, \tag{3.25}$$

and the vehicle orientation becomes

$$\varphi_G = \varphi_k = \varphi_{G_0} + \int \frac{|\Delta\vec{v}|}{b} dt \,. \tag{3.26}$$

3.3.2. Front (or rear) wheel turning steering

The front wheel steering configuration is the most common configuration used in normal road vehicles. In this configuration the front wheels only are steered with respect to the vehicle center, while the rest of the wheels are kept straight and have to follow the path made by the steered wheels; this causes skidding at the end of the un-steered wheels. The steering geometry depends on the Ackerman steering principal [56], [57] where the inner wheel turns tighter than the outer wheel as shown in Figure 5.

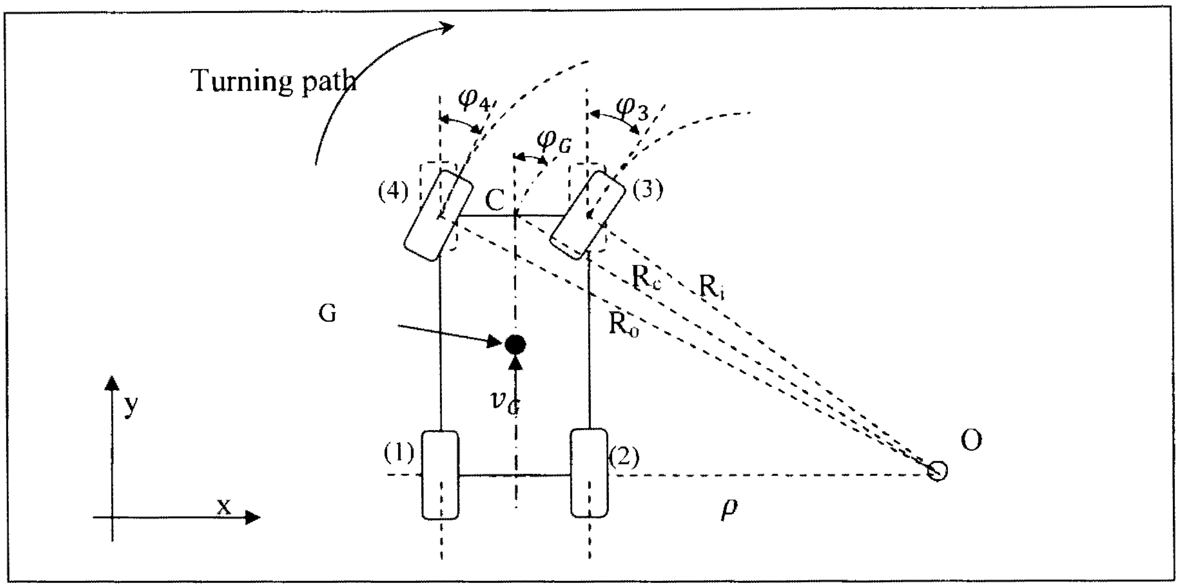


Figure 5: Front wheel steering configuration.

The vehicle can be driven with either two or four wheels; the most general case considered in this discussion is when the vehicle is driven with all four wheels at velocities $\vec{v}_1$, $\vec{v}_2$, $\vec{v}_3$ and $\vec{v}_4$. During the straight line motion, the vehicle behaves, as in the previous case, with all $\vec{v}_k$ assuming equal values and pointing in the direction of motion of the vehicle. For vehicles with low width to length ratio $\left(\frac{a}{b}\right)$ during turning maneuvers, the trajectory of the center of mass can conveniently be approximated by that at the center C of the steered wheels. There is an instantaneous center O of zero velocity at a distance ρ from the inner wheel, i.e., wheel 2, along the axis of non-steered wheels such that the inner and outer wheel angle φ_3 and φ_4 satisfy

$$\varphi_3 = \tan^{-1}\left(\frac{a}{\rho}\right) , \qquad (3.27)$$

$$\varphi_4 = \tan^{-1}\left(\frac{a}{\rho+b}\right) . \quad (3.28)$$

The angle of the center of mass is approximated by

$$\varphi_G = \tan^{-1}\left(\frac{a}{\rho+\frac{b}{2}}\right) . \quad (3.29)$$

The wheel angles for the non steered wheels are assumed to be zero, i.e., $\varphi_1 = \varphi_2 = 0$. For a fixed ratio $\left(\frac{a}{b}\right)$), the steered angles (φ_3 and φ_4) can be computed using the desired heading direction of the vehicle, φ_G, and the equations (2.27)-(2.29) as

$$\varphi_3 = \cot^{-1}\left(\cot\varphi_G - \frac{b}{2a}\right) , \quad (3.30)$$

$$\varphi_4 = \cot^{-1}\left(\cot\varphi_G + \frac{b}{2a}\right) . \quad (3.31)$$

The velocities of the steered wheels, i.e., k = 3, 4, for the front wheel steering can be expressed as

$$v_k = v_G + [\csc\varphi_k - \csc\varphi_G]\frac{v_G}{\csc\varphi_G} . \quad (3.32)$$

However, the velocities of the non-steered wheels, i.e., for k = 1, 2, depend on the location of the wheel since their angles remain fixed in the direction of the vehicle. The velocity for wheel 1, which is in the same side as the steered wheel 4, will be

$$v_1 = v_G + [\cot\varphi_4 - \csc\varphi_G]\frac{v_G}{\csc\varphi_G} . \quad (3.33)$$

Similarly, wheel 2, which is in the same side as the steered wheel 3, will have a velocity as

$$v_2 = v_G + [\cot\varphi_3 - \csc\varphi_G]\frac{v_G}{\csc\varphi_G} . \quad (3.34)$$

3.3.3. All wheel steering general motion

The all wheel steering configuration is a new configuration that steers all wheels of the vehicle. Such a steering configuration is now making ground in the automobile industry since it can steer the vehicle with less turning radius as compared to the front wheel steering configuration, and the rear wheels are made to track the front wheels, thereby avoiding the skidding at their end [58]. The all wheel configuration depends on the dual Ackerman steering principle as shown in Figure 6.

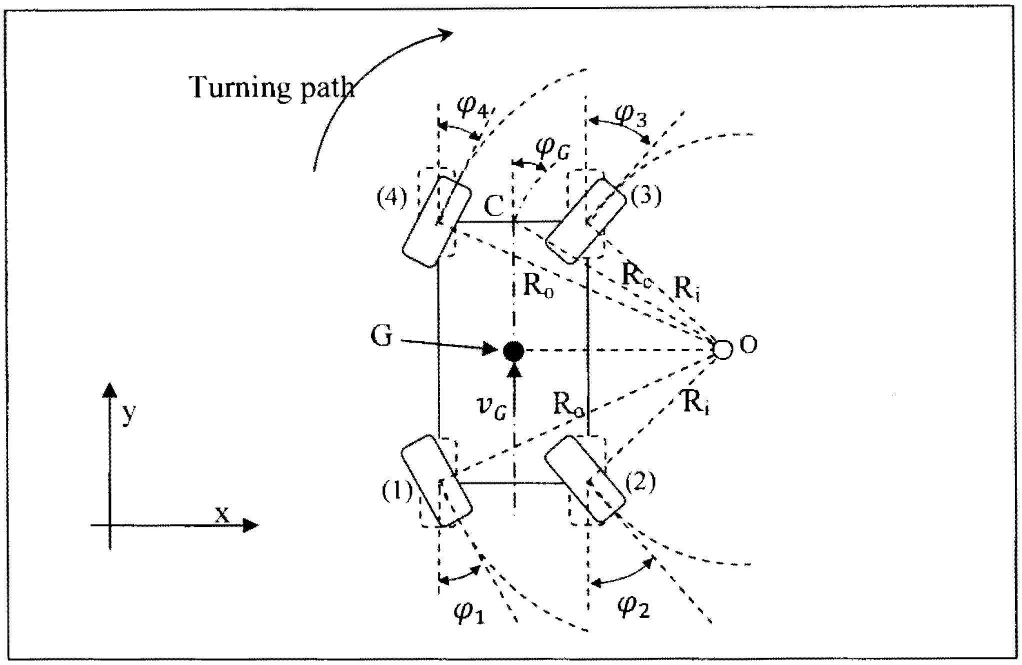


Figure 6: All (Four) wheel steering configuration.

As in the front wheel steering or the fixed straight steering, this configuration too behaves the same way under the vehicle's straight line motion, i.e. all four wheels drive with the same velocity in the vehicle heading direction. However, during cornering, all of the wheels are steered simultaneously as compared to only two wheel steering in the front wheel steering configuration. In this configuration, the instantaneous centre O of zero velocity is located at a distance ρ from the inner wheel, but on the axis of the center of mass, while the steering angle for the outer and the inner wheels remain equal, respectively, such as

$$\varphi_1 = -\varphi_4 \;, \tag{3.35}$$

$$\varphi_2 = -\varphi_3 \;, \tag{3.36}$$

where the wheel angle φ_1 and φ_2 satisfy the equations as

$$\varphi_1 = \tan^{-1}\left[\frac{a}{2(\rho+b)}\right] \ , \tag{3.37}$$

$$\varphi_2 = \tan^{-1}\left(\frac{a}{2\rho}\right) \ . \tag{3.38}$$

The angle of the center of mass is approximated by

$$\varphi_G = \tan^{-1}\left(\frac{a}{b+2\rho}\right) \ . \tag{3.39}$$

Using the equations (2.37)-(2.39), the wheel steering angle can be correlated to the vehicle steering angle as

$$\varphi_1 = \cot^{-1}\left(\cot\varphi_G + \frac{b}{a}\right) \ , \tag{3.40}$$

$$\varphi_2 = \cot^{-1}\left(\cot\varphi_G - \frac{b}{a}\right) \ . \tag{3.41}$$

The velocity of the steered wheels, i.e., of all four wheels, can be expressed the same as was done before in the front wheel steering configuration, such as

$$v_k = v_G + [\csc\varphi_k - \csc\varphi_G]\frac{v_G}{\csc\varphi_G} \ , \tag{3.42}$$

for k=1,2,...,4.

3.3.4. All wheel steering translation motion (parallel)

If the vehicle is at a very high speed and needs to make a lane change then the steering configuration described in the previous sub-sections may not work well, in particular they can produce a rolling effect, which destabilizes the vehicle. In order to maintain the vehicle stability in such situations the parallel movement of the vehicle, which can be provided by the parallel steering configuration, is highly preferred. In such a configuration, all wheels steer at the same angle with respect to the turning angle of the vehicle center making a parallel shift without inducing any roll in the vehicle. Since all wheels are steered, the vehicle can be made to travel in any direction relative to its axis as illustrated in Figure 7.

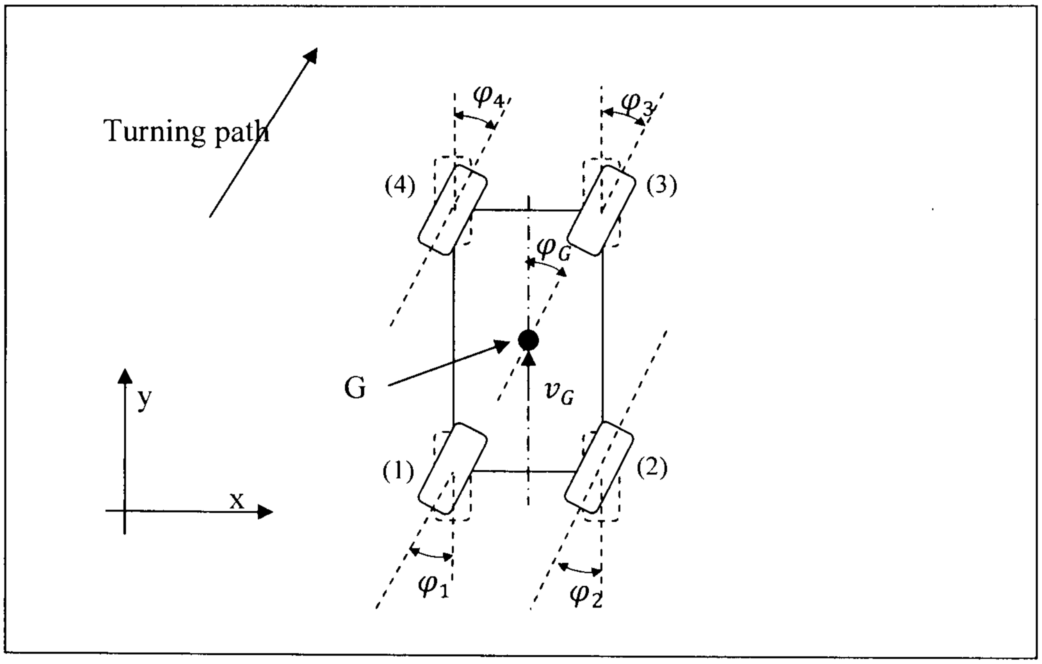


Figure 7: Parallel steering configuration.

The common feature of this motion is that all wheels have the same velocity as that of the center of mass, i.e.,

$$\varphi_1 = \varphi_2 = \varphi_3 = \varphi_4 = \varphi_G \;, \tag{3.43}$$

$$v_1 = v_2 = v_3 = v_4 = v_G \;. \tag{3.44}$$

During the motion, the vehicle heading direction does not change, i.e.,

$$\dot{\varphi}_G = 0 \;, \tag{3.45}$$

therefore,

$$\vec{v}_{k|G} = \mathbf{0} \;. \tag{3.46}$$

The vehicle trajectory can easily be shown as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \int \begin{bmatrix} v_k \cos\varphi_k \\ v_k \sin\varphi_k \end{bmatrix} dt \;, \tag{3.47}$$

for all $k = 1,2..,4$.

3.3.5. All wheel steering zero radius turning

To turn the vehicle in a cluttered terrain or on a narrow path like tunnel, it is preferred to let the vehicle turn at the axis passing through its center of gravity. This minimizes the space required for a successful turn. The maneuver is thus known as the zero radius steering. The steering configuration for zero radius turning is illustrated in Figure 8..

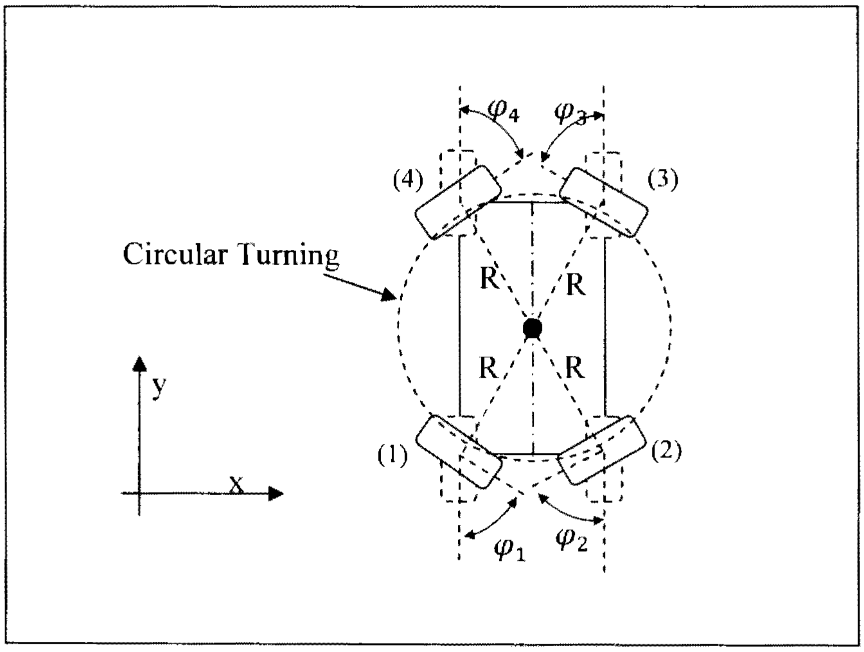


Figure 8: Zero Radius steering configuration.

In this steering basically the center of mass remains stationary while the vehicle changes direction continuously, as

$$\vec{v}_G = \mathbf{0} \ , \qquad (3.48)$$

$$\dot{\varphi}_G \neq \mathbf{0} \ . \qquad (3.49)$$

Here all wheels are subjected to velocities and steering angles of equal magnitudes but in different directions. Diametrically opposite wheels are steered at equal angles but opposite velocities such as

$$\varphi_1 = \varphi_3 = -\varphi_2 = -\varphi_4 \ , \tag{3.50}$$

$$\vec{v}_1 = -\vec{v}_3 \ , \tag{3.51}$$

$$\vec{v}_4 = -\vec{v}_2 \ , \tag{3.52}$$

where

$$|\varphi_k| = 90° - \tan^{-1}\frac{b}{a} \ , \tag{3.53}$$

and

$$v_1 = v_2 = v_3 = v_4 = v_G \ , \tag{3.54}$$

providing the vehicle orientation as

$$\dot{\varphi}_G = \frac{2v_k}{\sqrt{a^2+b^2}} \ . \tag{3.55}$$

4. THE EXPERIMENTAL ROBOT PLATFORM

The problem task for implementation of the proposed control approach includes building a control system hardware that treats the vehicle as a system of particles and test its performance in various path geometries. The test vehicle used in the work for the implementation of such kind of control system was a pre-built robotic platform dubbed BIBOT-1, and the control framework employed was a decentralized control architecture that allowed each wheel to be controlled independently.

This chapter describes the robot platform that was used in this research. It discusses the structural details and limitations of the robot and states the constraints and the scope of this work.

4.1. BIBOT-1 Structure

The BIBOT wheeled robot project is a Mechatronics project in the Department of Mechanical Engineering at NDSU; it intends to develop flexibly reconfigurable robots that can be used for research in a variety of terrain conditions. The development of BIBOT-1 robot started fall 2006 with the aim to create a robot that could be operated in slippery terrains at high speeds. The basic structure of the robot was designed by a team of several undergraduate students as the part of their senior design project [67]. The final mechanical structure of the robot consisted of a chassis with an edge to edge length of 37 inches and a width of 10 inches; the wheel to wheel base was 23 inches long and 26 inches wide. The robotic vehicle had four wheels, each using an independent suspension system. Each wheel was driven by an in-wheel hub motor coupled with a stepper motor to steer it. This made the vehicle an all wheel drive, all wheel independent steering system unit as shown in Figure 9.

The first control system for this robot was designed and built by another team of undergraduate students from the Electrical Engineering Department at NDSU as part of their design projects [68]. The controller consisted of two microcontrollers: the PIC16F877A and the PIC10F200. The major function of the PIC16F877A microcontroller was to read the sensors and drive the DC hub motors, whereas a PIC10F200 microcontroller served as the low end controller for generating the grey codes needed in controlling the stepper motors.

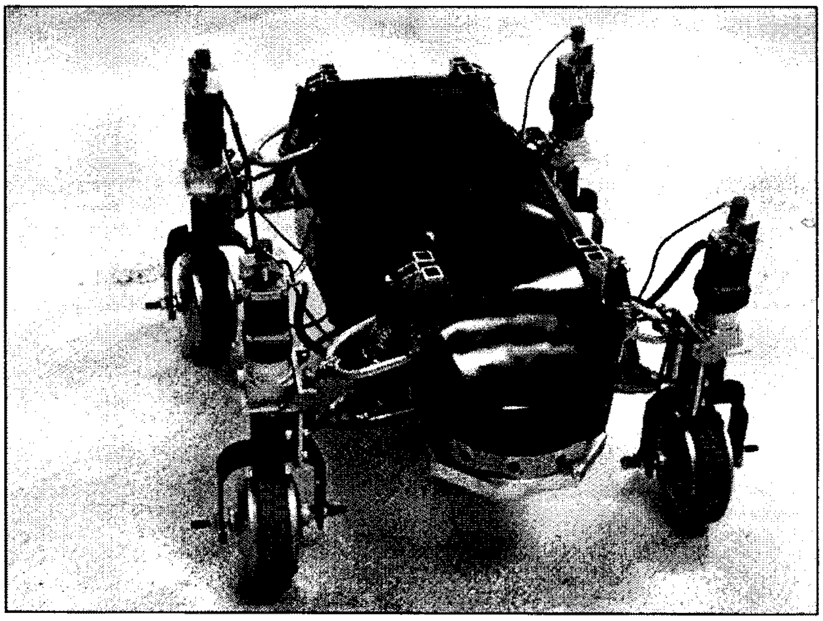
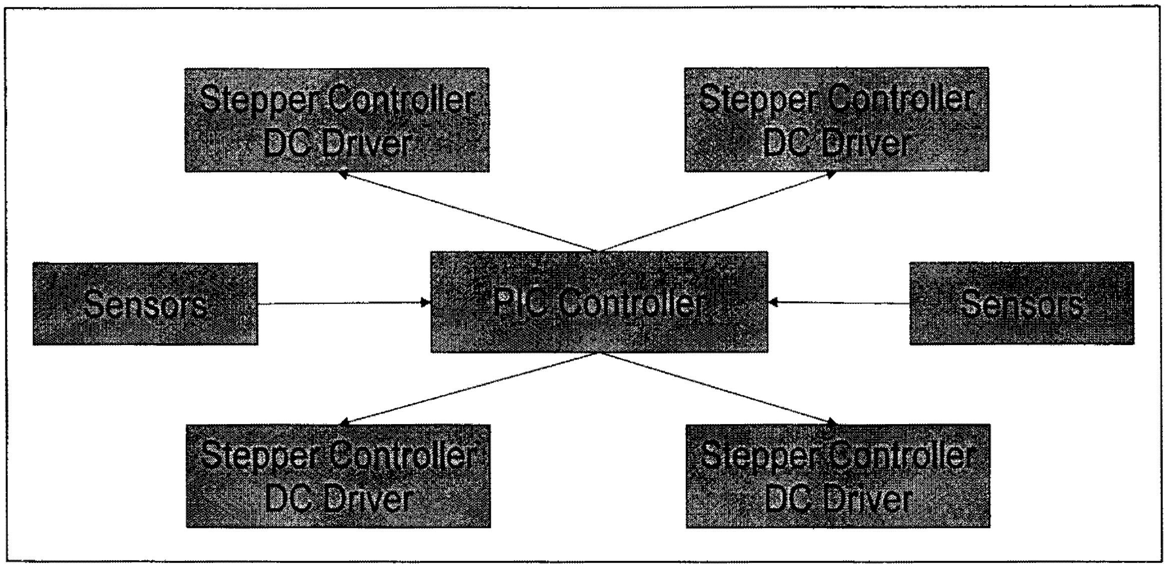


Figure 9: The BIBOT-1 robotic vehicle

In construction, this control system had two sections. The first section was the vehicle control unit (VCU), which acted as the central controller unit managing the other parts of the control system. The second section had four units acting as the local control units, which control the motors on each wheel. These local units were known as the motor control units (MCU). Potentiometers attached to the steering stepper motors provided the feedback data needed for the steering controller. Although the robot was equipped with sonar sensors that could be used to detect obstacles, there was no speed feedback information for both the VCU and the MCU. Figure 10 shows the structure of the initial controller, and the actual control boards for both the VCU and the MCU are shown in Figure 11 and Figure 12.



Stepper Controller DC Driver
Stepper Controller DC Driver
Sensors
PIC Controller
Sensors
Stepper Controller DC Driver
Stepper Controller DC Driver

Figure 10: Block diagram for the initial controller [68].

The VCU, which was the workhorse of the whole control system, was capable of reading 16 analog sensors and could provide control commands to four MCUs only; it provided all motion commands that were implemented by the individual MCUs. This controller organization provided little or no flexibility in the drive structure of the robot. Part of the purpose of employing Particle Modeling on a robotic vehicle was not only for creating independent wheel controllers for each wheel, but also to have the ability to add a wheel and change the base lengths of the vehicle. With the existing controller it would have been difficult to achieve this. In particular, the addition of wheels would be a complicated effort for the controller, and the MCUs were not as independent as they should be.

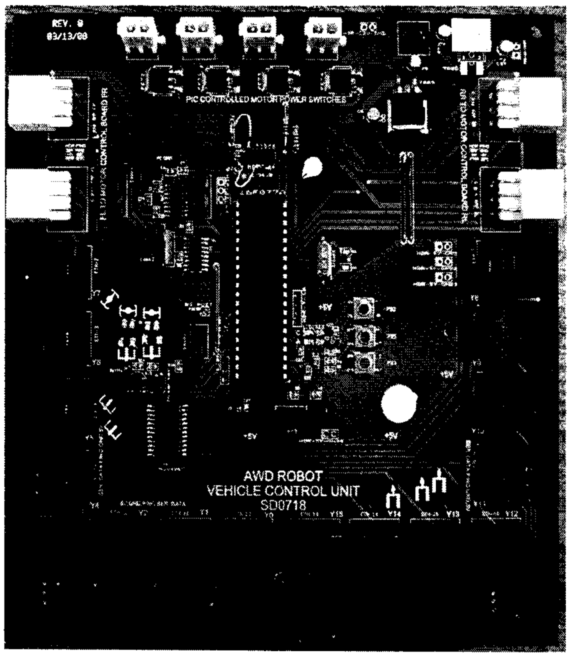


Figure 11: Vehicle Control Unit (VCU) [68].

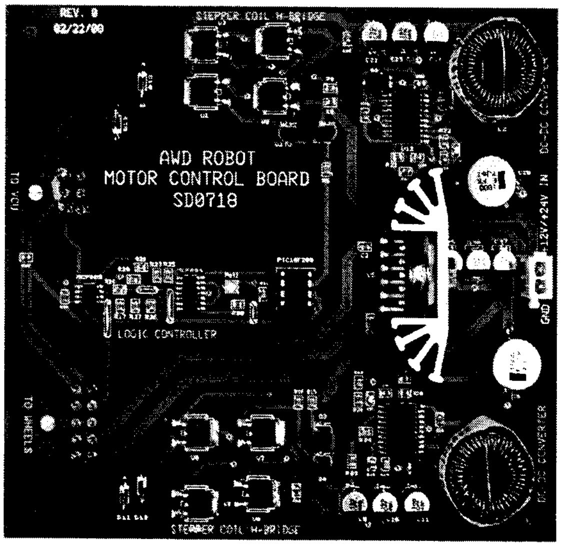


Figure 12: Motion Control Unit (MCU) [68].

Despite the structural limitations, there were also hardware limitations with the existing controller. The MCUs were capable of driving the DC hub motors at 12 volts DC only, which was another limitation since it failed to cruise the robot at sufficiently high speeds by almost 50%: the hub motors are rated at 24Volt DC. Additionally, the stepper motor driving function of the MCU was not capable of controlling the current flow in the stepper motor coils. This was not beneficial for the robot especially since adverse operation conditions could cause high currents to flow through the stepper motor and possibly damage them. This controller was also unable to provide sufficient holding torque for the wheels to be kept in their respective positions.

Out of the concerns raised above, this initial controller was deemed incapable of providing the needed control functions for BIBOT-1.

4.2. Thesis Constraints And Scope Of Work

The work covered in this thesis aimed to develop a new control system for BIBOT-1 using Particle Modeling approach that could provide the minimal tracking errors and satisfy the constraints as follows:

1. Robot's speed: It was necessary for the controller to steer the robot at as high speeds as possible. Robot maneuverability at high speeds was one of the subjects that would be studied using BIBOT-1. Hence, since the hub motors that drive the wheels were rated at 24Volts DC, it was important for the new control system to provide that level of voltage supply to these motors.
2. Scalability: The studies may require the robot to have more wheels than the current four, especially on the robot mobility over unstable surfaces. As such, it was

important for the controller to be able to accommodate any number of robot wheels and steering as necessary.

3. Performance Reliability and Equipment Safety: The use of stepper motors in driving the robot may have introduced some unintended constraints. The most notable constraint was that of being able to firmly hold the wheels in the intended direction, which was accomplished by holding the stepper motor phases at a steady voltage. However, this could result in the motor drawing damagingly higher electrical currents than the rated values. It was necessary for the controller to provide safety mechanisms that would guard the motor against over currents.
4. Wheel Independence: Another objective of the Particle Modeling was to provide each wheel with an independent controller. The existing system did not provide that feature as the control commands for all wheels were developed by a single VCU.

One of the major requirements for the application of the Particle Modeling approach is the availability of appropriate sensors on the robot to provide all information about the vehicle motion and the wheel parameters. Unfortunately, BIBOT-1 did not have all of the necessary sensors, such as the proximity sensors for obstacle detection and speed sensors. Therefore, some of the robot parameters, especially the speed, were assumed to always be constant. This degraded the quality of the controller since it was not possible to ensure that the speed remained constant. This is the limitation that the research had to live with.

5. CONTROL SYSTEM DESIGN

The design and development of the control system involved two major tasks. The first task was to design and develop the hardware, and the second task was to develop the control system's core software (Firmware). Robotic intelligence was not considered during this development; however, the core software was built with a platform on which the intelligence code could be easily added in the future. This chapter discusses how both the hardware and the core software were developed using the Particle Model kinematic analysis of the vehicle as presented in Chapter 3.

5.1. Hardware Design

5.1.1. Architecture

Controlling all steering and traction actuators sometimes poses a complex and non-linear problem. To effectively control them in robotic vehicles, several control algorithms have been studied and proposed as can be seen in the literature [69-71]. Most of these algorithms are based on a rigid body model and are known to require a high computational speed in order to generate the various control signals at a proper time without delay. For such requirements, different control architectures have been proposed [72]. The first one is the Centralized control architecture [73] in which there is one central controller that combines both high level and low level controls. Although the centralized control system is simple in its architecture, it reduces the speed of the system by limiting the number of commands it can execute over a particular time interval. Another drawback is its un-scalability; it is not possible to add or remove any actuator units from the system after it has been built and is ready to run. Since the complexity of the system increases exponentially with the increase in the actuator units, it is evident that the centralized control architecture fails when the complexity of the system increases.

Another popular control architecture is the Hierarchical control architecture [74], [75], which decomposes the large and complex system into small modules that operate hierarchically from high level to low level control functions. Thus, an application of the control effort becomes easier and faster as compared to the centralized control system. However, this architecture also faces some limitations and drawbacks resulting from the arrangements of the sub modules; to avoid the misinterpretation of the sensory data, it demands more processing time as the information further increases [76].

Similar to the centralized architecture, another major limitation of the hierarchical architecture is its inflexibility and in-expandability. As scalable robotic systems become the order of the day, the need for the control architecture that eliminates these limitations favors the decentralized or distributed control architecture. In decentralized control architecture [77-82], the low level control units are resident local to the actuators. Local drive units are controlled by the local controllers, which get the information from a central controller to run the system in harmony. The central controller reads the sensors and provides the required information to move the vehicle in the desired direction. This information is shared by the lower level control units, which processes it according to their respective positions in the system and generates appropriate actuation signals, thus, it uses the concept of parallel computing [76]. Such a system not only reduces time delay in the information flow and in the generation of the control, but also improves the scalability factor of the system.

In accordance with the performance objectives discussed earlier and the above mentioned analysis, it was decided to use the decentralized control architecture [83-85] for this robot. Under the decentralized control architecture, as illustrated in the Figure 13, the system is broken down into simple controllers, each of which performs specific local tasks independent of the other controllers, though they may share or change certain control information through some form of a data communication bus. Typically, each controller can have its own set of sensors that are local to that controller, and the measured data may be shared by the other controllers.

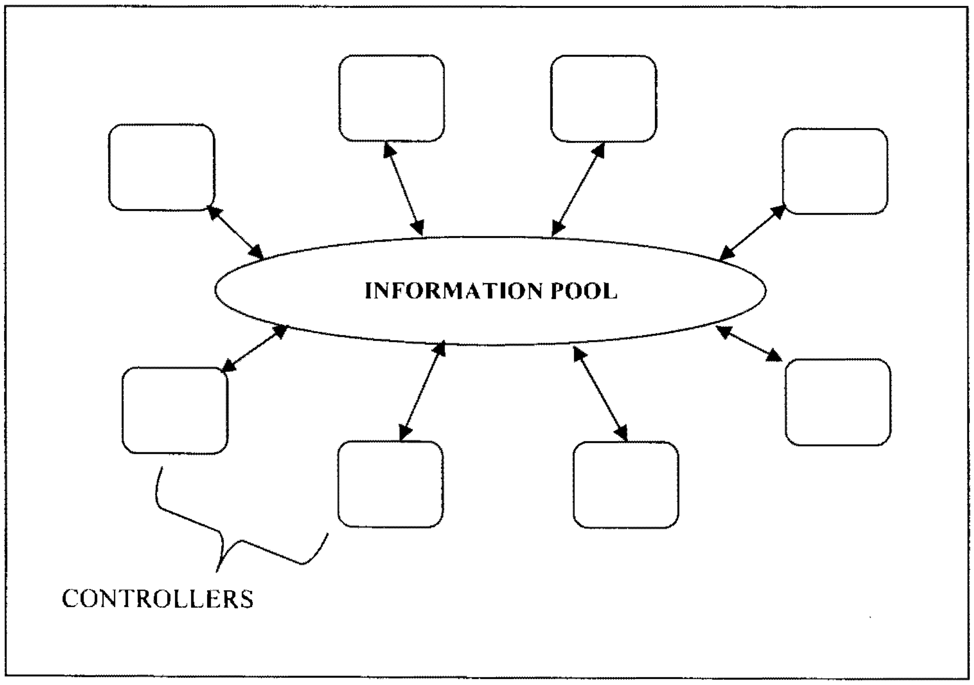


Figure 13: Decentralized control architecture.

On this robot, it was decided to have five such controllers, one of which was referred to as the Vehicle Body Controller (VBC). This controller was dedicated to monitoring the motion of the robot body as a whole with the ability to change the information in the information pool (data bus). This controller would determine the direction, speed and the steering mode of the robot. The remaining controllers were assigned to the wheels of the robot; one for each wheel, known as a Wheel Control Unit (WCU). The WCU could read the information from the data bus without making any changes on it and at the same time continue with its own sensor data to determine the wheel traction speed and the steering angle. The VBC would use the body sensor data and its navigation software, which will be the part of the robot intelligence (currently not fully developed), to determine the robot heading direction, speed and the steering mode. This information is then broadcasted to the information data bus for all WCUs to read. The overall arrangement of the controllers is illustrated in Figure 14.

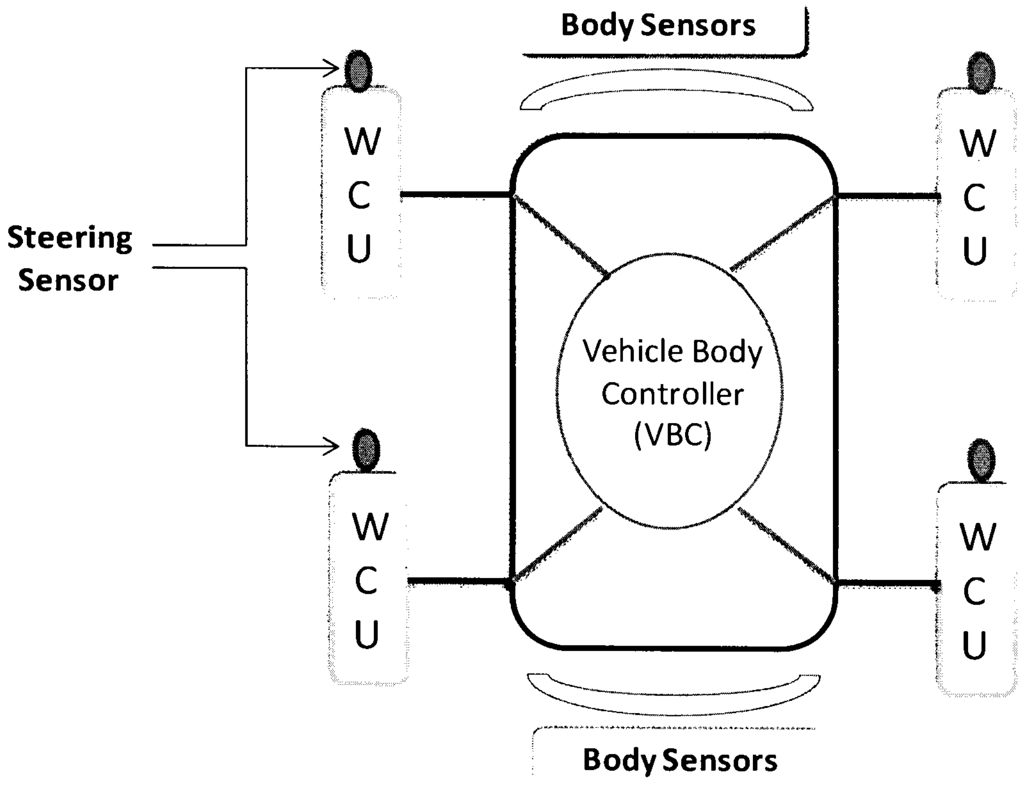


Figure 14: Structure of the Decentralized control system implemented on the four-wheel fixed axle vehicle.

The information passed to the data bus by the VBC would be influenced by the physical environment of the robotic system in which it resided as well as the goal of the system itself. The WCUs would respond by combining the VBC data at the data bus and their local sensors to develop their local motor control signals, consistent with the desired robot motion. The processing of the information by the individual WCU would depend on its particular location on the robot along with the information it received from its respective sensors.

This control structure has a number of advantages for this system. Although it is characterized by the fast processing of information with minimal time delay in the flow of data between the VBC and the WCUs, which is an advantage by itself, the main advantage is the fact that any number of WCUs can be added without compromising its performance, i.e., system scalability. These features are consistent with the structural demands of the Particle Model structure, wherein each particle is assumed to be independent; so having its own independent controller is highly desirable.

5.1.2. Hardware Planning

Once the control architecture was decided, the next step was to decide how to get the control process working. There were two things that needed to be considered prior to designing and building the hardware for these controllers. The first was the drive electronics, which includes the drives for the traction and the steering control. Recall that DC hub motors were used for the generation of the traction, whereas stepper motors were used for steering. The speed of the DC hub motors, which is directly proportional to the supply voltage, can be controlled either using linear amplifiers or the switching amplifiers

[86-88]. Though linear amplifiers tend to be simple in design, they have low efficiency as compared to switching amplifiers; this can cause an early drain out of the batteries. The efficiency of the switching amplifiers is nearly 100% if the power switches used in the driver are properly designed with less energy loss, which can happen in the form of heat dissipation. The reason for having such high efficiency is that, ideally, when the power switch is ON, it conducts all of the current without any potential drop across it, and when it is OFF, there is no current flowing, thus providing no energy loss. The voltage output in the switching amplifiers is controlled using a sequence of pulses whose DC component is responsible for driving the motor at that instance. The input pulses can be either Pulse Width Modulated (PWM) (Figure 15), Pulse density Modulated (PDM) (Figure 16), or delta sigma modulated (Figure 17) signals. The advantage offered by the switch amplifiers makes them a clear choice for robot applications, where the battery life needs to be optimized.

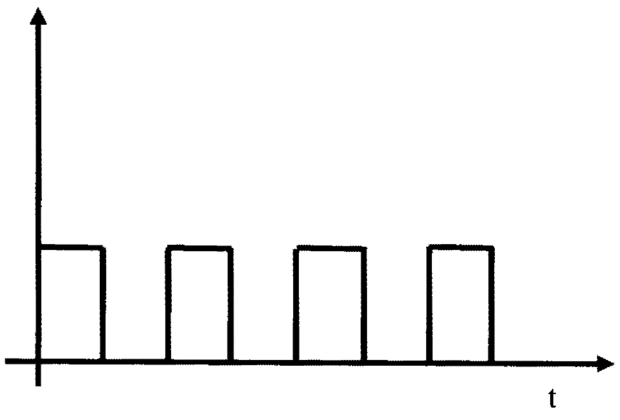


Figure 15: 50 % Duty Cycle PWM signal.

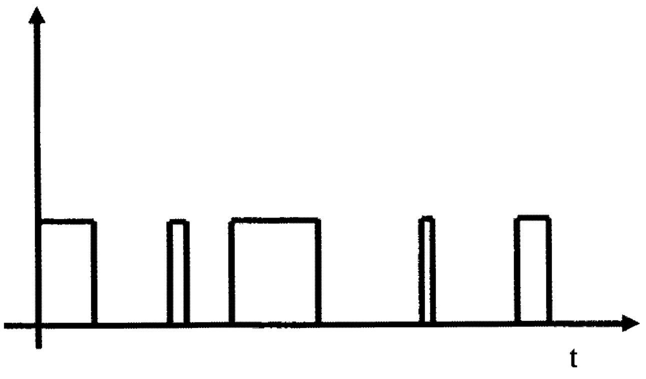


Figure 16: Pulse Density Modulated signal.

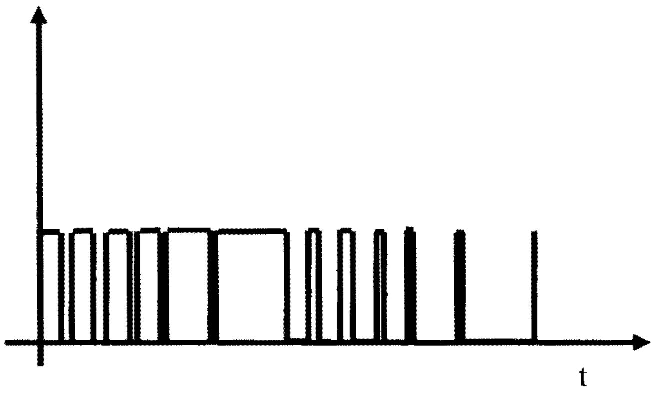


Figure 17: Sigma Delta Modulated signal.

With the drive electronics chosen, the second issue to be considered was the control section, which generates the various control signals such as timing and sequences of pulses to operate the drive amplifiers. The control section can be designed and developed by using either a combination of logic gates and some special purpose chips for information sharing, or a microcontroller that can perform both tasks of information sharing and the generation of control signals. Although the use of logic gates eliminates the need for programming as required in the case of the microcontroller usage, they tend to increase the design complexity. For example, the process of reading sensors and processing the sensor information alone may require several components such as comparators, oscillators, flipflops etc. In order to maintain the simplicity of the control section, a microcontroller based design was considered. It was also decided that the switching signal required for the switching driver be a PWM signal, which can easily be generated by any microcontroller.

5.1.3. Hardware Building Blocks

5.1.3.1. Power Electronics

Although not required by the VBC, power electronics was the most important part of the WCU because it was responsible for driving the traction DC hub motor and the steering stepper motor. The power electronics modules were made using H-bridge based driver units [86], [89] for both the DC motor and the stepper motor on the WCU. The selection of the appropriate drive components for each motor depended on the power ratings of the motor in question. The traction DC motor has current ratings that vary from 0.8 amps at no load condition to a maximum of 58 amps at stall when operated at a voltage of 24 volts, whereas the stepper motor is rated with 2.1 amps in bipolar series connection. There was no off the shelf drivers that could provide such tremendous power ratings at a reasonable rate. It was decided therefore to custom build the drivers for both motors.

a) The DC motor driver

The DC motor driver was designed on the principle of the H-bridge that can be used as switching amplifiers. The first step in the design process was the selection of appropriate power transistor switches that could be used in the H-Bridge circuitry. The high power N-channel MOSFETs, rated for 80 amps maximum at the voltage level of 55 volts, were selected; these power switches were found to be ideal for the design since they could not only handle the power requirement of the motor but also provide a reasonable factor of safety in the design. The H-bridge driver circuit was completed by arranging the power transistor switches in the fashion as shown in Figure 18, where the transistors are labeled as Q_1, Q_2, Q_3 and Q_4. Since, the N-channel MOSFET opens fully only when the gate voltage is higher than the source voltage, to address the problem of switching the high side MOSFETs in the H-Bridge circuit, two IR21094 driver chips, which are half bridge drivers, were used. These driver chips addressed the switching requirement for the N-channel MOSFETs by setting up appropriate gate voltages on the high side and the low side MOSFETs in the H-Bridge. This driver chip also induces a programmable dead time in-between the control signal in order to avoid any accidental opening of two power switches on the same side of the circuit at same time which would cause a short circuit.

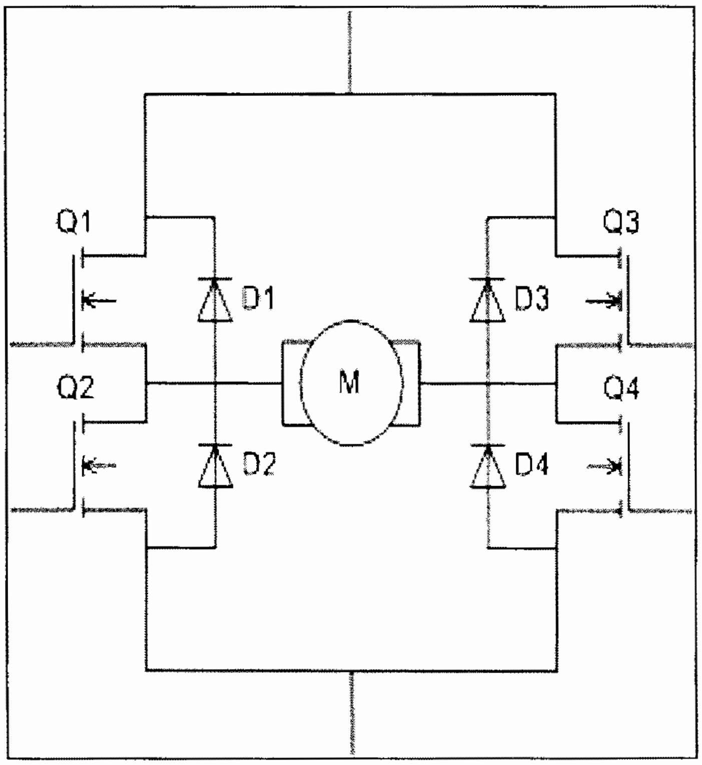


Figure 18: H-Bridge configuration.

The PWM control signals for the H-bridge driver are complementary to each other; these signals are received by the two IR21094 chips, which convert the signals into the appropriate MOSFET gate voltages to open and close the power switches. The speed and the direction of rotation is controlled using a phase locked loop approach, in which a special case of PWM signal controls both the motor speed and the direction by using a single variable i.e., the duty cycle of the input PWM signal. In this approach, a duty cycle of less than 50 % causes the opening of one power transistor, say on the high side, more than the corresponding low side transistor on the same branch of the bridge, say the left branch. Since an inverted signal also was fed to the second pair of bridge transistors, it causes the opening of the low side transistor in the right branch longer than the high side transistor in the right branch. The average opening of these transistors allows the electric current to flow from left top to right bottom causing the motor to run in one direction as shown in Figure 19. The opposite is also valid i.e., a duty cycle of more than 50 % causes the motor to reverse its rotation shown in Figure 20. The duty cycle of 50% is the boundary between two opposite direction rotation of motor; at this value the motor stops, since it causes opening of either high side transistor switches or the low side transistor switches only. The speed control function is achieved by using the average time duration for the opening of the transistors. The greater the difference of the duty cycle from 50%, the greater will be the speed. In order to stop the motor at any time instance, a break input signal was also available on the drive. The generation of these control signals will be discussed in detail later. As usual a tank circuit comprised of two capacitors of different values laid in parallel was also placed across the power supply line in order to absorb any ripples produced in the supply line. The final circuit for the motor driver is shown in Figure 21.

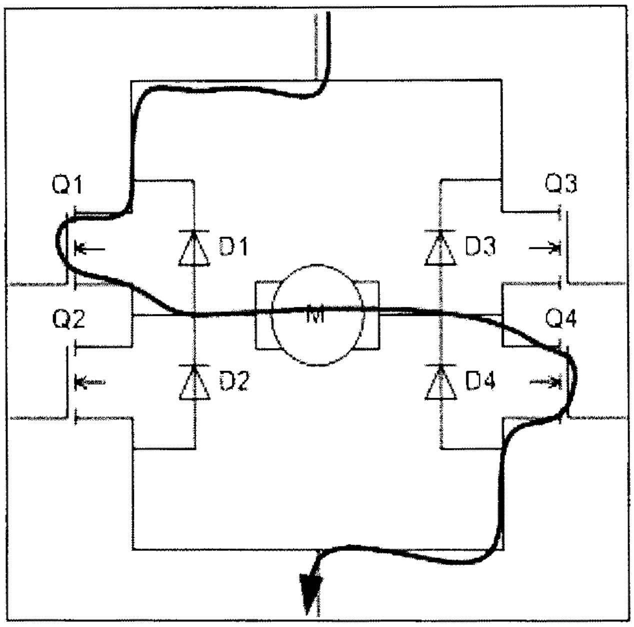


Figure 19: Current flow in H-Bridge when duty cycle < 50%.

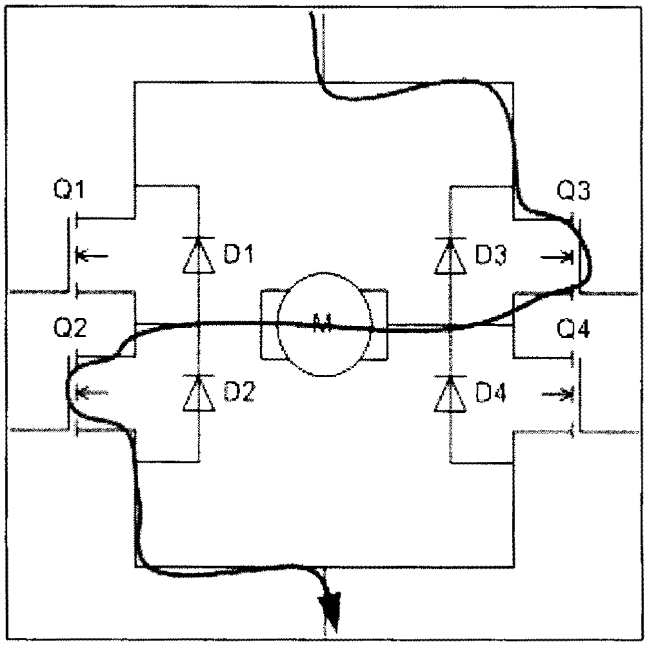


Figure 20: Current flow in H-Bridge when duty cycle > 50%

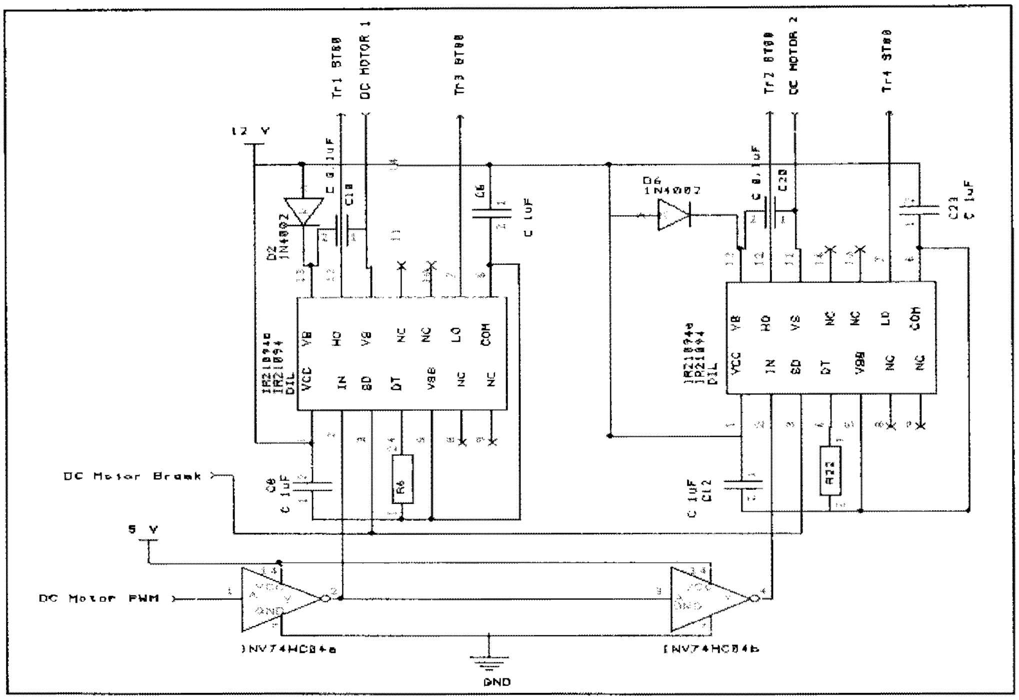


Figure 21: Circuit diagram for DC Hub motor driver.

b) The Stepper Motor Driver

Since, stepper motors have some unique performance requirements that must be achieved for efficient performance, the design of the stepper motor driver tends to be more complex. In order to run the stepper motor at high speed with the rated torque, the current flow in the motor coils plays a significant role. In order to achieve the rated torque, the motor must be driven at the rated current values. As the speed of the motor increases, however, the winding inductance and the back electromotive force (E.M.F) generated in the coils during rotation tend to resist the flow of current by providing less time for the current grow to the maximum value; hence, the maximum attained current may fail reach the rated value. This, in effect, decreases the torque provided by the motor. As the speed increases further, less will be the time for the growth of the current in the motor and the motor cease to produce torque at all. So in order to maintain the torque at a high speed, a high voltage source is required than rated value that helps to achieve the rated current in the coils at less time. However, a check on the current flowing in the coils must be provided since at low speeds the coils can draw more current than rated values from the high voltage source, which may result in breakdown of the motor. Several solutions are available to address this problem; however the most popular approach is by using a chopper circuit across the motor coils [86]. A chopper circuit can be programmed to work at a specific current value and if the current rises beyond that value it cuts off the power supply and wait till the current in the coils goes below that value. Fortunately there are some off-the-shelf chips readily available in the market that serves for this purpose; one of them is the LMD18245 chip. This chip is a complete half bridge chip that contains power MOSFETs along with the appropriate circuitry for activating the MOSFETS, and a chopper circuit which can drive up to 3 amps at 55 volts. With minimum passive components such as resistors and capacitors, the chip can be used to run a single coil of the stepper motor; a pair is required to run the stepper motor in bipolar configuration. The chip uses a four bit chopper circuit that can be programmed, as specified by the manufacturer, using a variable value resistor to limit the current value from 0-3 amps. Thus, the stepper motor driver was designed and built using the LMD18245 chip. Again, the control signal requirement for this drive were two PWM signals also (discussed in the next section on Firmware development) and a braking signal. The circuit diagram for the drive can be seen in Figure 22.

5.1.3.2. Microcontrollers

The heart of each of these control units was an array of one or more microcontrollers that provided the required control signals. A thorough study was conducted to select appropriate microcontrollers for the developed control system. Among

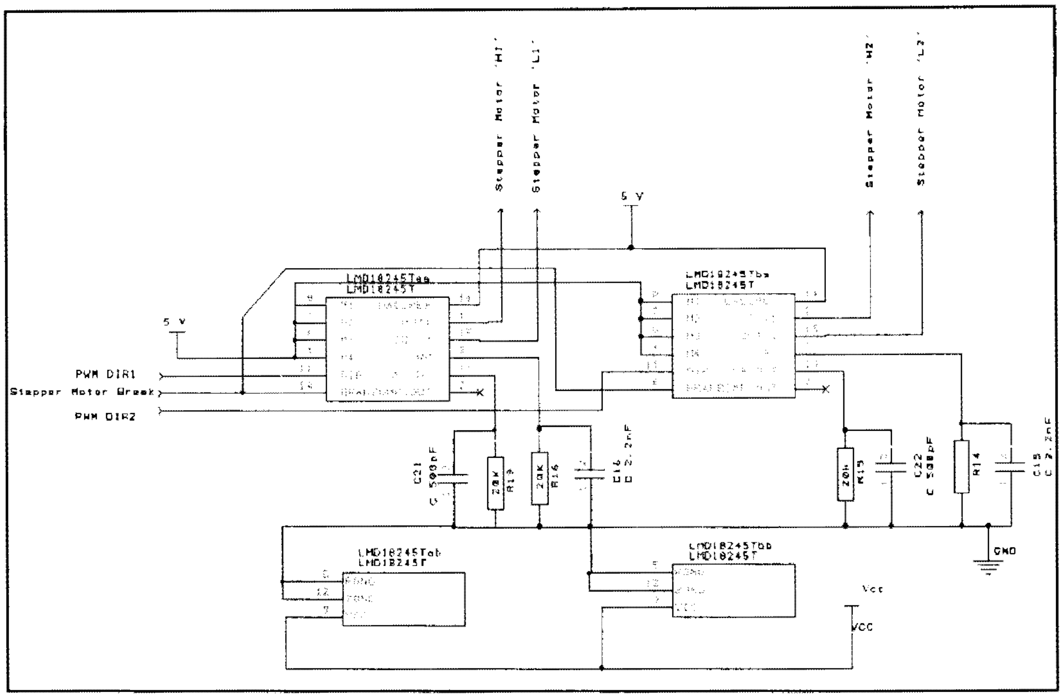


Figure 22: Circuit diagram for Stepper motor driver.

the possible candidates were the PIC and dsPIC family by Microchip, the HCS12 family by Freescale, the C200^{TM} series from Texas Instrumentation and the XE166 family from Infineon Technologies. The deciding factors in the selection of a particular microcontroller, as shown in Table 1, included the cost, the processing speed, the communication interface and the available on board facilities such as analog to digital converters (ADC) and the Pulse Width Modulation (PWM) modules. After a recursive evaluation of the available options, the dsPIC33FJMC802 microcontroller made by Microchip [90] was selected for both the VBC and the WCU; this microcontroller was not only easy to handle, program and mount on the board, but also provided various levels of security and remappable peripheral pins, which added further to the flexibility of the designed controllers. Although, the Freescales's MC9S12DP512 in the 68HCS12 family, having additional features like large memory, more number of input/output pins [91], was also another best option, they are rather found to be relatively costlier and are sold in surface mount packaging only, which was hard to handle in the lab; hence the option was discarded.

Microcontroller	Unit Price ($)	Package	ADC channels	Dedicated PWM channel	Communication	Availability
dsPIC33FJ	6.94	DIP	6x10b/12b	6	SPI/I^{2}C/UART	Yes
HCS12	10.28	QFP	8x10b	8	SCI/SPI/I^{2}C	Yes
C200	10.20	LQFP	5x10b	7	SCI	No
XE166	6.68	LQFP	9x10b	-	SPI/I^{2}C/UART	No

Table 1: Comparison table for the selection of the microcontroller.

Microchip's dsPC33FJ128MC802 microcontroller is a newly launched chip in dsPIC33FJ family by the company; it was released in 2008 and is specifically designed for the motor controls [92]. This chip is a 16 bit high performance microcontroller containing a digital signal processing (DSP) functionality and a processor capable of 24 bit instruction word with variable length opcode field. The DSP engine of the dsPIC33FJ128MC802 features a high-speed 17-bit by 17-bit multiplier, a 40-bit arithmetic logic unit (ALU), two 40-bit saturating accumulators and a 40-bit bidirectional barrel shifter capable of shifting 40-bit value up to 16 bits right or left, in a single cycle. Using the on chip Phase-Locked Loop (PLL), the microcontroller can be programmed to run on its maximum operating frequency of 40 MHz.

The dsPIC microcontroller alone was not sufficient for providing all kinds of control signals needed by the system; therefore, another low end and cheap 8 pin PIC10F206 microcontroller made by the same Microchip company was used along with the dsPIC, for generating the PWM control signals for the stepper motor driver. The chip has only 33 instructions sets written in assembly language; it is easy to use comprising of only 4 input-output programmable pins.

5.1.3.3. Supporting electronics

The microcontroller and the drive circuits described so far need other supporting electronics for them to work properly. The supporting electronics for this purpose were: a) the power supply unit, b) a target board, and c) the optical isolation circuitry for the inductive load motors.

a) Power Supply Unit

The power supply unit is required to serve as a common power house to supply various voltage levels to the different parts of the system. The unit is divided in two sections; one for microcontrollers and other for the motor drives, both having their own independent power source. To save on space with the possibility for a future increase in power requirements due to added WCUs, the power supply unit for the controller section was placed on the VBC board. For the motor drives' power supply unit, a separate circuit board was designed to handle the high current flow and for the safety requirements of the system. For the safety purpose, the board has a kill switch, to off the complete power supply for the motor drives, along with N channel power MOSFET switch's, one for each WCU, to switch off the unregulated battery power supply to the respective WCU at any time. The board circuit is illustrated in Figure 23.

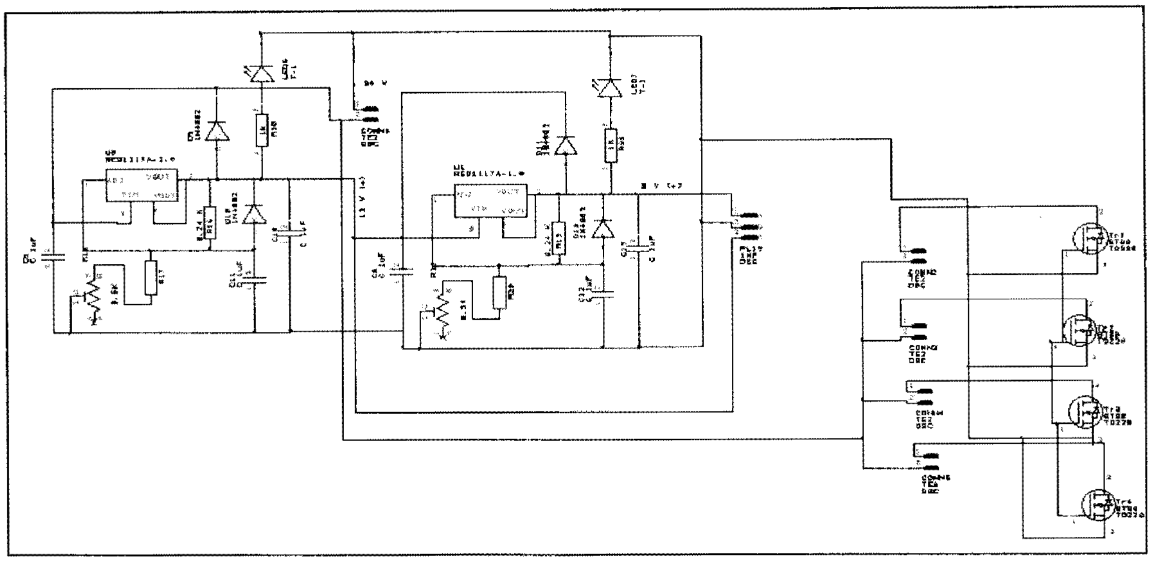


Figure 23: Circuit diagram for the motor drives' power supply.

The needed voltage supplies ranged from 3.3 volts to 24 volts. The off-the-shelf variable voltage regulator LM317 [93] chips were used in building voltage regulator units for producing the required voltage levels of 3.3 V, 5V and 12 V; however the 24 V supply was unregulated and supplied from the two 12 V batteries connected in series. The 3.3 V power supply was required for the control section comprises of microcontrollers, while the rest of the voltage levels were associated with the motor drives' section. A total of seven voltage regulators were used on the system, out of which five were used to supply the 3.3 volts to power the controller section for the VBC and the WCUs, one regulator was used for providing 5 volts to power the isolator circuits (discussed later on) on all WCUs and one regulator was used for 12 volts to power up the DC motor driver chips on all WCUs.

The LM317 chip is a three terminal positive voltage regulator operating from 3 volts to 40 volts, providing full overload protection capabilities such as current limit, thermal overload protection and safe area protection. The chips were used in a standard circuit as shown in Figure 24 providing an output current capability of up to 1.5 amps.

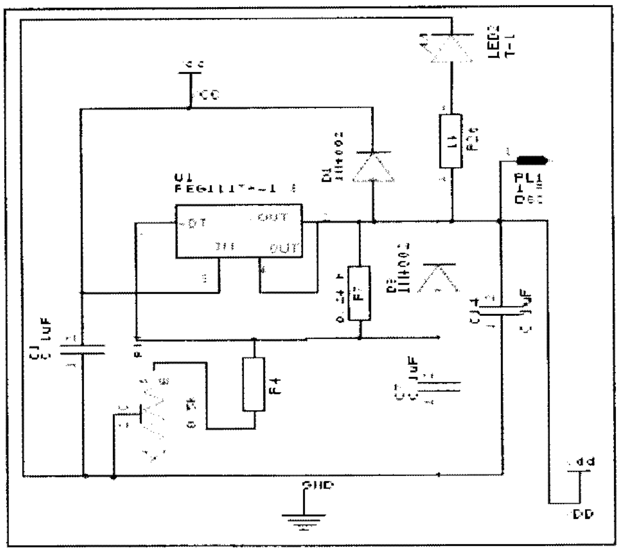


Figure 24: Circuit diagram for the voltage regulator

b) Target Board

The next supporting circuit was the target board used for programming the dsPIC33FJ microcontrollers. The target board is a special interface for connecting the microcontroller to the programmer. It should be able to provide the power and the connectivity requirements for the microcontroller, which includes voltage supply, bypass capacitors at voltage input pins of microcontroller and a 6 pin Ethernet connector jack. These are needed to power the microcontroller and connect it to the programmer in order to accept a new program on it. The target board also provides the on board debugging facility by allowing access to the unused pins of the microcontroller. Along with these requirements, a set of eight light emitting diodes (LEDs) was also placed on the board to help in the troubleshooting. The complete assembled board is shown in Figure 25, along with its circuit diagram shown in Figure 26.

Figure 25: Target board for dsPIC33FJ programming.

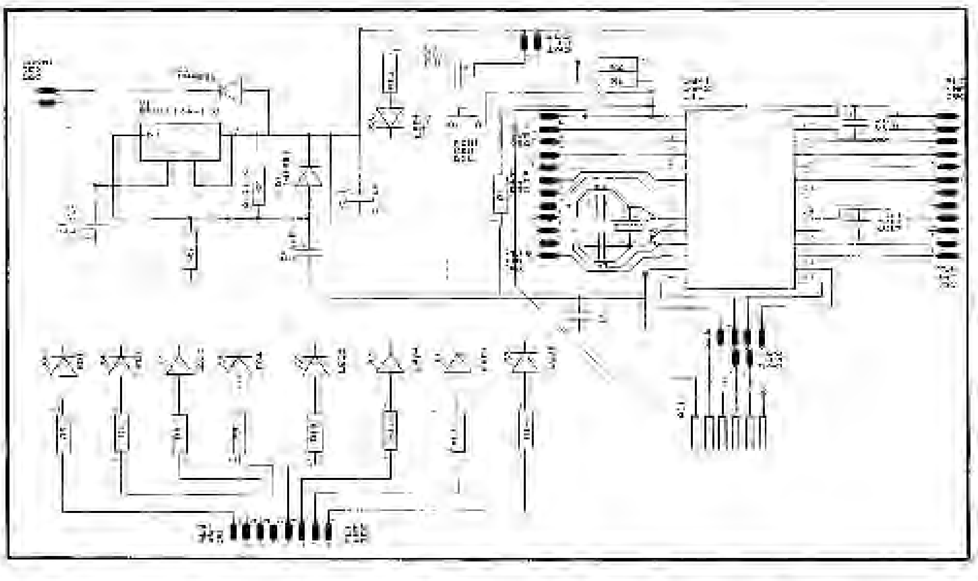


Figure 26: Circuit diagram for target board

c) Optic Isolation Circuit

Optic isolation was needed in order to reduce on board signal interferences and thus maintain stability of the controller [86], [94]. While testing the DC motor simultaneously with the stepper motor, it was found that when the DC motor was run, it generated some noise signals that flowed back to the microcontroller and distorted the stepper motor controller signals, thereby hindering the performance of the stepper motor. So it became necessary to completely isolate the drive power supply from the control section. An on board optic isolation circuit was designed and built using phototransistor opto-couplers and Schmitz trigger buffers. These isolators uses a photo emitter diode and a photo receiver transistor to have a one way flow of signal without having any physical connectivity in between the two ends as shown in Figure 27; the Schmitz trigger is required to make the output signal crisp. The complete on board circuit is shown in Figure 28. On each WCU a set of five such isolator circuits were used for all control signals of the drives.

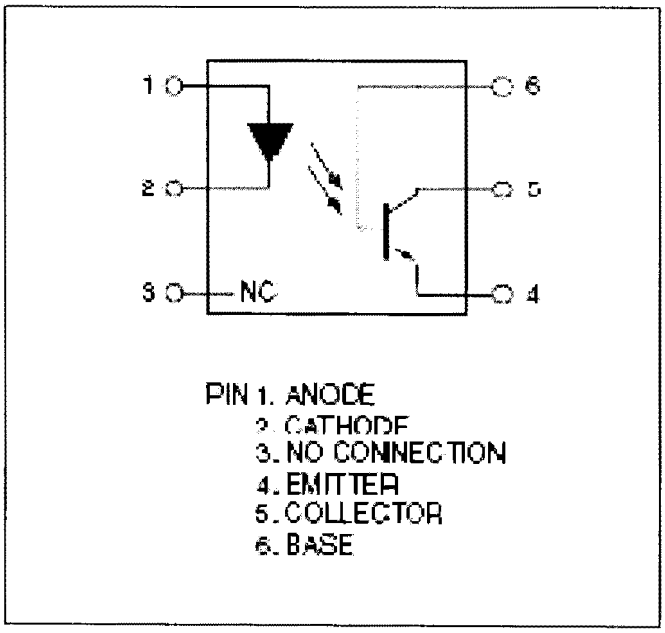


Figure 27: Internal circuit diagram for optic isolator

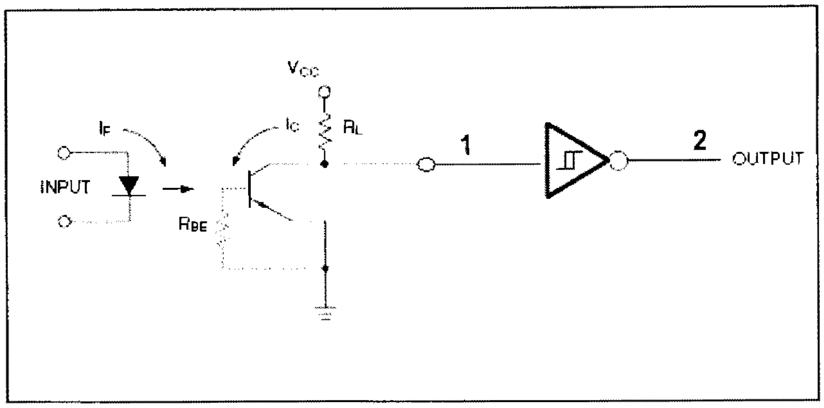


Figure 28: Optic isolator circuit used on board.

5.1.4. VBC

The VBC was the only decentralized controller that could read the vehicle body sensors and update the information on the data bus accordingly. The controller was designed and built using the building blocks that included a dsPIC33FJ128MC802 microcontroller and a 3.3 volt voltage regulator. The controller had the ability to read six analog sensors and one serial peripheral interface (SPI) using four wire configuration. At the writing of this thesis, the VBC had not been connected to any sensor because no appropriate sensors could be obtained for this purpose in a timely manner. It is hoped that in the future the VBC will have a three axis accelerometer and an array of sound navigation and ranging (SONAR) sensors will be connected to enable monitoring the vehicle kinematics and detecting obstacles in the environment. For that reason, a provision was also provided by identifying and assigning some free remappable pins of the microcontroller that will be used for connecting these sensors. This could help to use any useful sensors with different communication interface like universal asynchronous receiver/transmitter (UART) mode, etc. The implementation of the VCU design can be seen in Figure 29 and the complete VBC board is shown in Figure 30.

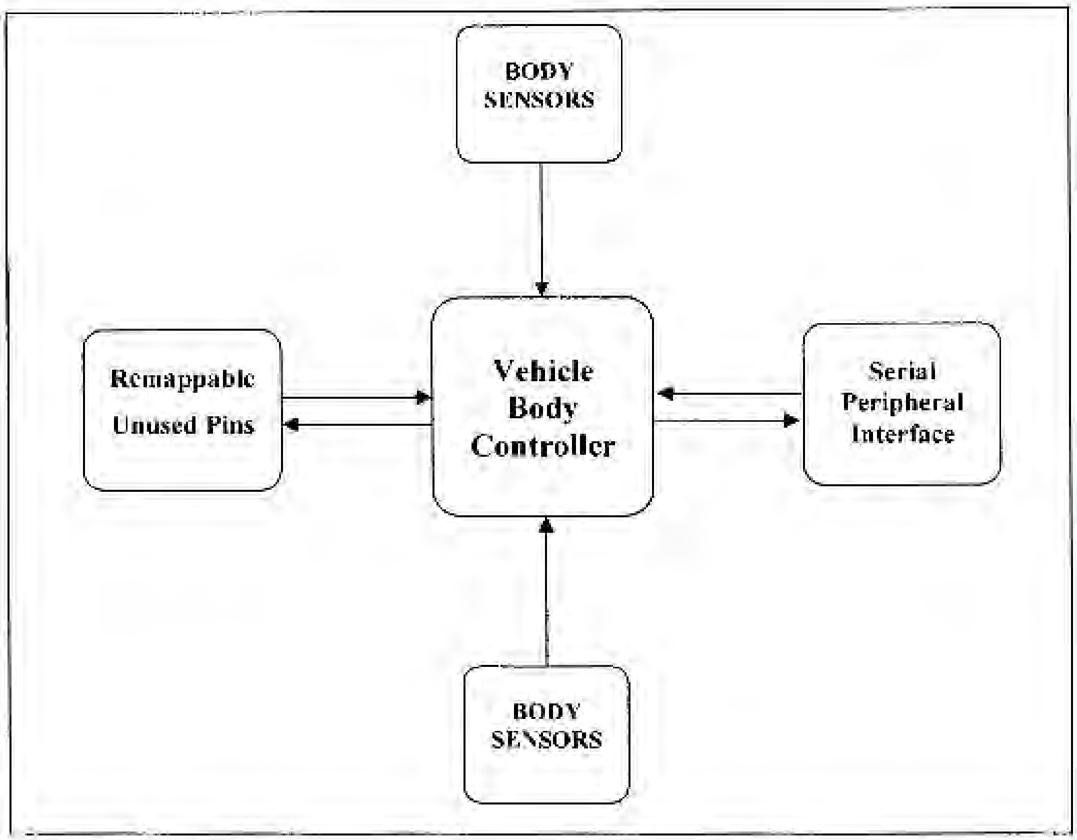


Figure 29: Block diagram showing the VBC design.

Figure 30: Completely assembled VBC.

5.1.5. WCU

The WCU system is the workhorse controller for the robot as it is responsible for making the robot do the actual intended movements. There were four such controller units on this robot, one on each wheel, connected to the common data bus. The WCUs were designed to operate independently.

Each WCU had a traction DC motor driver and a steering stepper motor driver along with a common dsPIC33FJ microcontroller for synchronizing the activity of both drivers and to provide their control signals. Since, the WCU was comprised of these high power drivers whose control section was highly prone to unwanted disturbances generated by the DC hub-motors, special care was needed in their design. As discussed earlier, it was necessary to completely isolate the power section of the drivers from the control section, and to check any cross talk between the drivers, by using optic isolators.

The on board microcontroller was made to read two analog sensors and one SPI communication channel that reads the information from the data bus. Of the two analog sensors, one was for indicating the steering angle and the other was intended for reading the wheel speed; at this stage the hub motor used on the robot lacks the speed sensors, thus a provision was made to be able to easily connect this speed sensor to the microcontroller, once it becomes available. Since the speed sensor could be a wheel encoder, provision was made to read those encoders and convert the output to analog that could be easily read through the microcontroller through the analog interface. The controller structure showing these local sensors is shown in Figure 31; the final WCU circuit board showing component layout is shown in Figure 32.

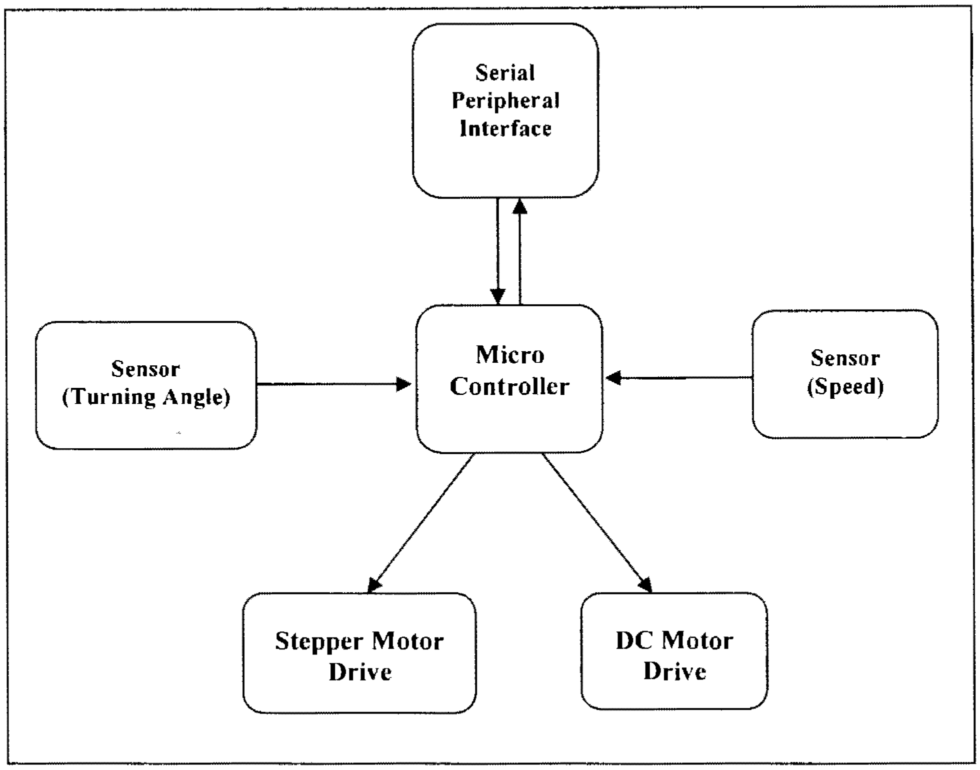


Figure 31: Block diagram showing the WCU design.

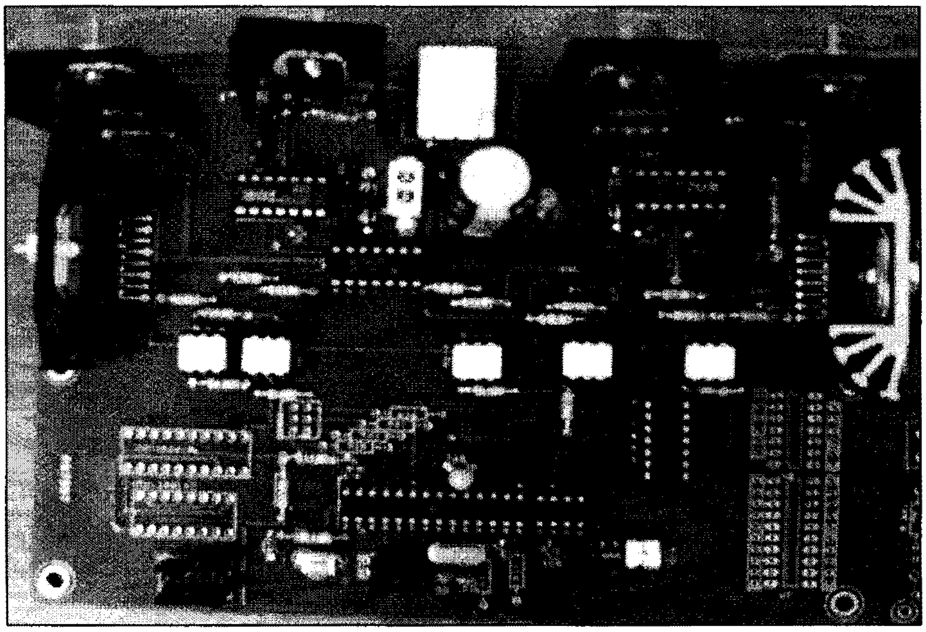


Figure 32: Completely assembled WCU.

5.2. Firmware Design

After the electronics circuit boards for the controllers were developed, the next task was to program the microcontrollers. The development of the proper codes or the firmware on the microcontroller was needed in order to not only synchronize both drives on the individual WCU, but to also control the information flow from VBC to the data bus. For the development of the dsPIC33FJ microcontroller firmware a fully integrated development environment i.e., MPLAB IDE was provided by the manufacture. This software included an assembler, compilers and supporting linkers, and library files along with the MPLAB SIM simulator.

The manufacturer for the dsPIC microcontroller recommended two options for programming and debugging the firmware: 1) to use a separate programmer and a debugger and 2) to use an emulator only. The emulator is a kind of device that has both features of a programmer and a debugger combined as single hardware, and supports a wide range of microcontrollers. For this project, it was decided to use the MPALB REAL ICE emulator which was found to offer significant advantages over competitive emulators including low-cost, full-speed emulation, real-time variable watches, trace analysis, complex breakpoints, a ruggedized probe interface and long (up to three meters) interconnection cables.

The development of the firmware for the designed control system included programming and debugging the codes for the VBC and the WCU microcontrollers. Code development was modularized with modules for configuration of the microprocessor, initialization of variables and supporting functions. Structurally these modules tend to be common for both of the VBC and the WCU controllers but differ in respect to their variables and functionality.

- Microprocessor Configuration: To maximize the application flexibility and reliability, and minimize costs by making use of minimal external components, it was necessary to first optimally configure the bits of the microcontroller. The configuration bits on chip controls the clocking speed, remapping ability of the pins (flexibility), code protection and code guard security.
- Initialization of variables: Variables were needed to store the sensor data as well as the processed data; the important data include analog input values of the sensors, information regarding the speed, steering angle values, and steering modes. They were also used to provide the certain duration looping of some certain module.
- Initialization of the functions: To ease the coding of the microcontroller, the on chip program was divided into a number of sub routines defined as functions, which included initialization of ports for input and output functionality, initialization of the ADC and the PWM module, remapping of port pins for the SPI module, reading of the sensors and updating of the variables. The ADC module was programmed to read the analog signals at a speed of 8 KHz; the module was interrupt driven. The SPI communication interface was programmed to work at a baud rate of 5 MHz; the three wire SPI setup was used for this case: one wire for the clock signal, one wire for the data and the third wire for the handshaking synchronization signal. The SPI was also interrupt driven. Since, both processes were independent of each other the default interrupt sequence preferences were used.

5.2.1. The VBC Firmware Development

The key objective of the VBC firmware is to provide the microcontroller on the VBC with the ability to read the sensors, process the sensor information and feed it to the data bus. Therefore, the program had modules for initializing the microcontroller, reading the sensors, manipulating the sensor data using the artificial intelligence and feeding the manipulated data to the information pool.

The microcontroller initialization process included initializing the analog to digital conversion channels for reading the sensors, and initializing the SPI mode of communication to feed the information pool with the processed data. Since, at the time of developing this control system there were no sensors connected to the VBC, some predefined fixed values were assigned to the sensor variables. Also, in cases where processing the sensors data needed an Artificial Intelligence, a simple system was created to mimic an artificial intelligence module. The full artificial intelligence system will be developed in future. Additionally, there was a small demo code created to help run the robot using the tethered controller. The flowchart for the VBC firmware can be seen in Figure 33. The data bus for this system was defined as the SPI module that connects the VBC microcontroller and all WCUs microcontrollers; the VBC microcontroller was configured as the master for the SPI communication

5.2.2. The WCU Firmware Development

The WCU firmware was required to provide the microcontroller with the ability to read the two analog sensors and the vehicle data available in the information pool along with generating the control signals for the DC motor driver and the stepper motor driver. The program also was modularized with modules for initializing the microcontroller, reading of the sensors and the information from the information pool, and for providing the control signals for the motor drivers as required.

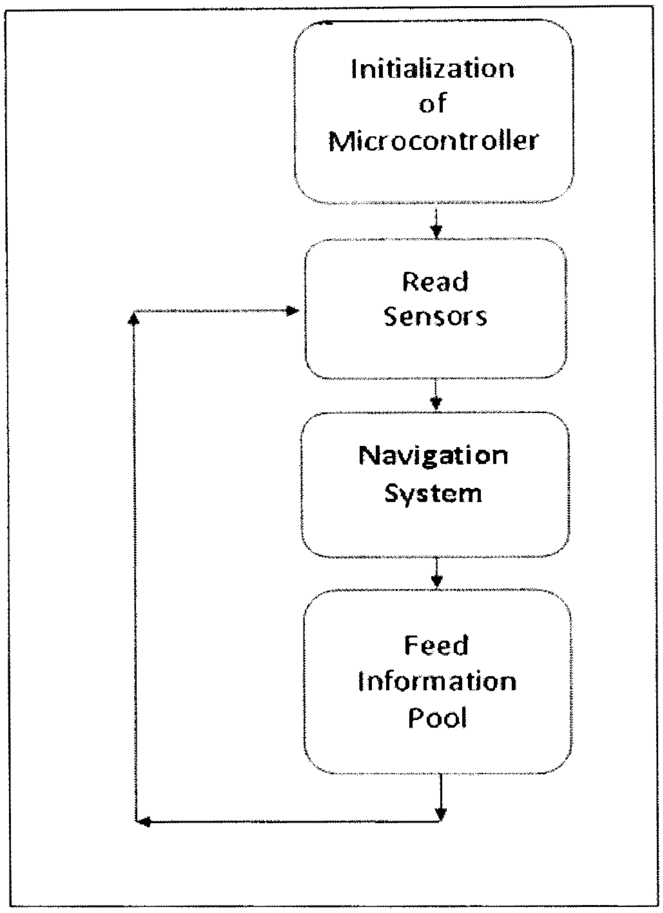


Figure 33: Flowchart for the VBC.

The coding started with the initialization of the microcontroller to generate the PWM signals, read the analog sensors and the information from information pool using SPI mode of communication. The controls for the DC motor drive and the stepper motor drive require the PWM signals and the two digital output signals for the brake control on each drive. For controlling of the drives one PWM signal was needed for the DC motor drive and two PWM signals for the stepper motor. The DC motor PWM signals were generated directly by the dsPIC33FJ, while those for the stepper motor were generated by using a small sized PIC10F206 microcontroller, which was controlled by the dsPIC33FJ. The PIC10F206 chip had two inputs from the dsPIC33FJ: one to select the motor direction and other to select the motor speed. The stepper motor was run using half stepping approach to maximize the resulting torque and efficiency; it also improved the holding torque of the motor and decreased the stepping angle. The PWM pattern for the motor half stepping forms cycles of two bit gray code as shown in Table 2.

Stepper Motor Coils				Electric Angle
A	$\bar{A}$	B	$\bar{B}$	
1	0	0	0	0
1	0	1	0	45
0	0	1	0	90
0	1	1	0	135
0	1	0	0	180
0	1	0	1	225
0	0	0	1	270
1	0	0	1	315

Table 2: Gray code for half wave stepping mode of the stepper motor.

The ADC module was used to read two analog signals one for the turning angle analog sensor and other for the speed. While to connect to the data bus, the microcontroller was programmed as the Slave using the SPI module. At the time, since there were no speed sensors on board, but for the future consideration provision was made in the coding also to read those sensors as analog input. To calibrate the speed of the motors and to program the microcontroller accordingly, an external sensor was used, thus providing a fixed speed control for the traction of the robot. The flow chart for the WCU firmware can be seen in Figure 34.

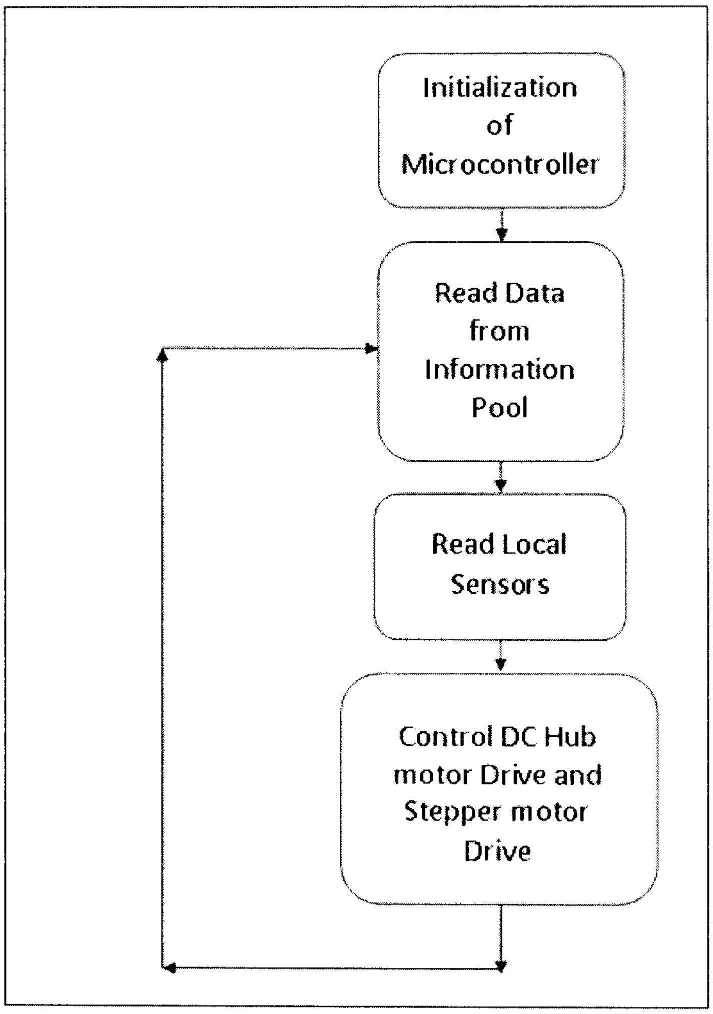

Initialization of Microcontroller
Read Data from Information Pool
Read Local Sensors
Control DC Hub motor Drive and Stepper motor Drive

Figure 34: Flowchart for the WCU.

5.3. Some Lessons Learned From This Effort

The design and development of the power drives and their synchronization on board went through number of trials because of performance problems that resulted from the design itself. The first major problem that was tackled in this process was caused by insufficient opening of the power transistor switches and the second one was due to the presence of crosstalk between the two high power drives designed on board.

5.3.1. Power Transistor Switches

The N-channel MOSFETs power switches require gate voltages of 2-10 volts in reference to the source voltage in order to fully open. When the high side power transistor is ON, the source voltage becomes the supply voltage, i.e., 24 volts, so to keep the switch ON, the gate must be held at a voltage of about 34 volts, which could not be provided by the normal PWM signals.

A number of solutions were available for this problem, which included the floating supply gate drive, the transformer coupled drive, the charge pump drive and the bootstrap circuit [95]. In the floating supply gate drive approach, an isolation circuit powered with a separate power supply is used to provide the switching action to the high side power transistors. The transformer coupled drive uses a high frequency gate transformer with dual secondary winding to control the switching action on both the high and the low side power transistor switches. One of the dual winding on the secondary side of the transformer is connected to the high side transistor, whereas, the second winding is connected to the low side transistor. The charge pump drive circuit uses capacitors to store charge which eventually does voltage multiplication and provide it during the switching gate of the high side transistor. Normally, the charge pump tends to use fewer components than the floating supply gate drive in providing the switching action. The bootstrap circuit uses a single capacitor that charges and discharges in order to provide the required voltage for the opening of the high side transistor. These available solutions with their features are summarized in Table 3.

High Side Gate driver solution for N-Channel MOSFET	
Floating Supply Gate Drive	Cost Impact of isolated supply is significant. The components required tend to be relatively expenses with limited bandwidth and noise sensitive.
Transformer Coupled Drive	Limited switching performance
Charge Pump Drive	Turn ON time tend to be long for switching application. Inefficiency in voltage multiplication.
Bootstrap Drive	Simple and inexpensive. Uses capacitor to provide high side voltage.

Table 3: High side gate driver solutions for N-Channel MOSFET.

After analyzing these methods, it was decided to use the bootstrap circuit [96] especially due to its advantage of being simple and inexpensive. Its typical structure is shown in Figure 35. The power supply V_{BS} for the high side transistor switch is kept floating. When the low side goes ON and the high side goes OFF, it also completes the loop for the V_{DD} to be connected to the ground, thereby charging the bootstrap capacitor through the bootstrap resistor 'R_{BOOT}' and the bootstrap diode 'D_{BOOT}' in series. When the high side turns ON and the low side turns OFF, the bootstrap capacitor discharges through the V_B and V_S loop, generating additional voltage on top of V_{DD} enough to fully open the high side transistor gate. Since, a diode is placed in between the power supply V_{DD} and the capacitor in reversed bias, when the bootstrap capacitor is discharged, it blocks the charge from flowing through V_{DD}.

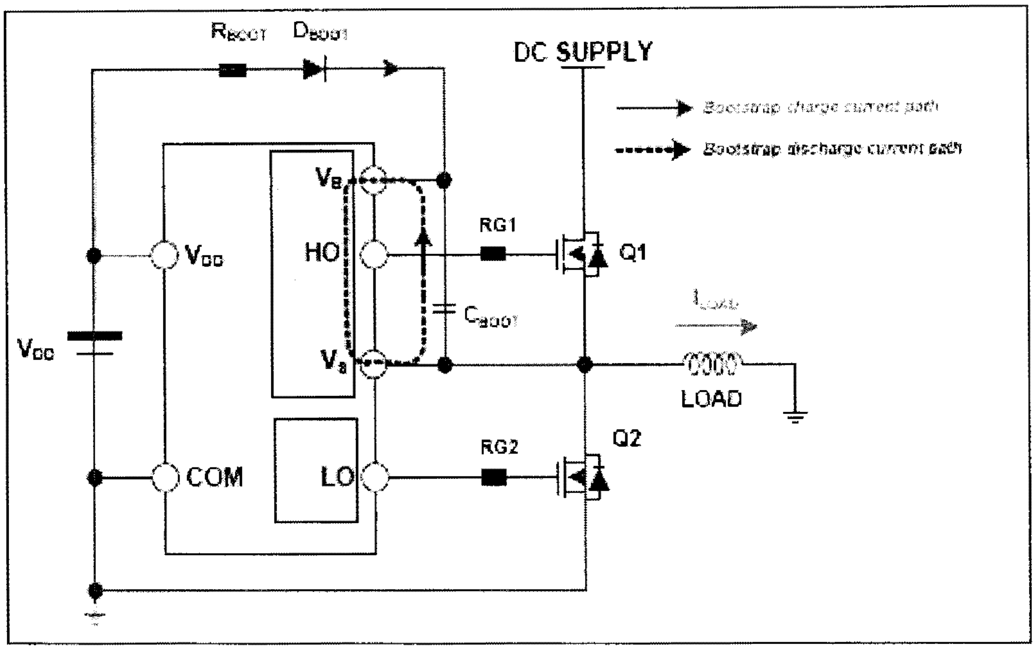

RBOOT
DBOOT
DC SUPPLY
Bootstrap charge current path
Bootstrap discharge current path
VB
HO
VCC
RG1
Q1
CBOOT
ILOAD
VCC
VS
LOAD
RG2
Q2
COM
LO

Figure 35: Bootstrap circuit diagram [95].

The selection of an appropriate capacitor (C_{BOOT}) for efficient performance of the bootstrap circuit depends on a number of factors, which include the gate charge on the power transistor used (Q_{gate}), the drop down voltage across the bootstrap diode (V_f), the supply voltage of gate driver (V_{DD}), the minimum gate source voltage (V_{GSMIN}) required to fully open the gate, and the current leakage in the circuit (I_L). If Q_{TOTAL} is the sum of the desired gate charge and the charge that will be lost in during the switching of the gate i.e.,

$$\mathbf{Q_{TOTAL} = Q_{gate} + I_L{*}\,t_{ON}} \,, \qquad \mathbf{(5.1)}$$

where t_{ON} is the high side switch ON time,

and ΔV_{BOOT} is the difference between the driver supply V_{DD} and the minimum voltage (V_{GSMIN}) required to open the gate along with the drop down voltage across the bootstrap diode i.e.,

$$\mathbf{\Delta V_{BOOT} = V_{DD} - V_f - V_{GSMIN}} \,, \qquad \mathbf{(5.2)}$$

then the value for bootstrap capacitor can be calculated as:

$$C_{BOOT} = \frac{Q_{TOTAL}}{\Delta V_{BOOT}} \quad . \tag{5.3}$$

During the circuit testing, it was found that even with an appropriate value of the bootstrap capacitor, its relative placement on the printed circuit board, length of connecting conductors, and the diode's recovery time could make the bootstrap circuit fail. If the capacitor is placed at some distance from the high side power supply, it may fail to provide the needed charge, therefore failing to open the gate. The reason was found to be due to the effects of the induced resistance between the capacitor and the driver chip, which forces the capacitor to discharge fast, without reaching to the specified voltage level. Similarly, when the capacitor was placed at some distance from the bootstrap diode, it resulted in loss of the capacitor charge due to voltage drop across the length of the conductor. Other problems were related to the geometry of conductors; if the on the board conductors are not straight enough, run in loops, it may induces some parasitic inductance in the circuit which can cause unwanted inductive effects in the circuit. Thus, to have an effective bootstrap circuit, the placements of the bootstrap capacitor and the diode are critical; normally the two components should be in a close vicinity of the high side power supply connected using straight and short conductors.

5.3.2. Cross talk in the Power drives

The problem of existence of crosstalk between the two motor drives on the WCU resulted in abnormal behavior of the motors, especially the stepper motor. It was observed that when the stepper motor ran alone, there was no considerable noise and the control signals were as anticipated as shown in Figure 36 and Figure 37. In these figures, the blue line was the braking signal which, when pulled HIGH, stops the motor; the other two signals were the PWM drive signals to provide the driving sequence for the stepper motor.

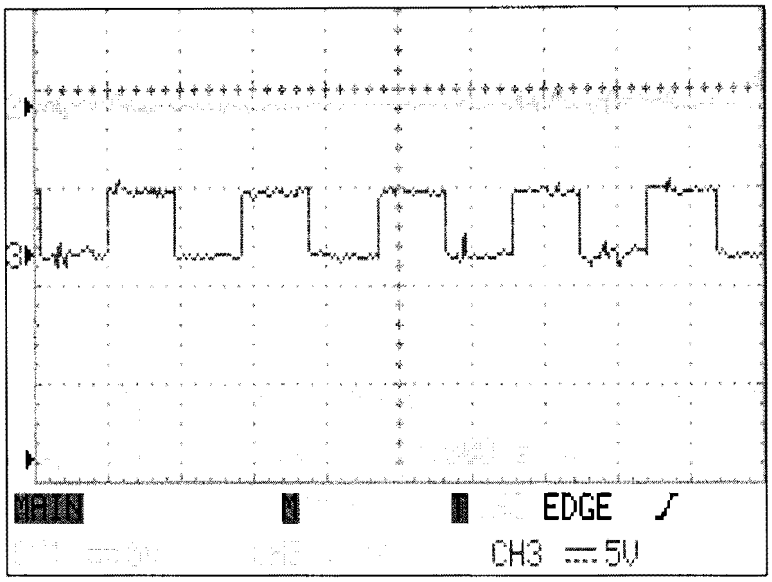


Figure 36: Control signals for the Stepper motor drive when in motion.

When the DC hub motor ran, it generated severe noise that distorted its control signals as shown in Figure 38. The yellow signal was the braking signal for stopping the motor when pulled LOW and the blue signal was the PWM signal to drive the motor. The distortion in the signals indicated that the noise coming from the DC hub motor was flowing back to the control side.

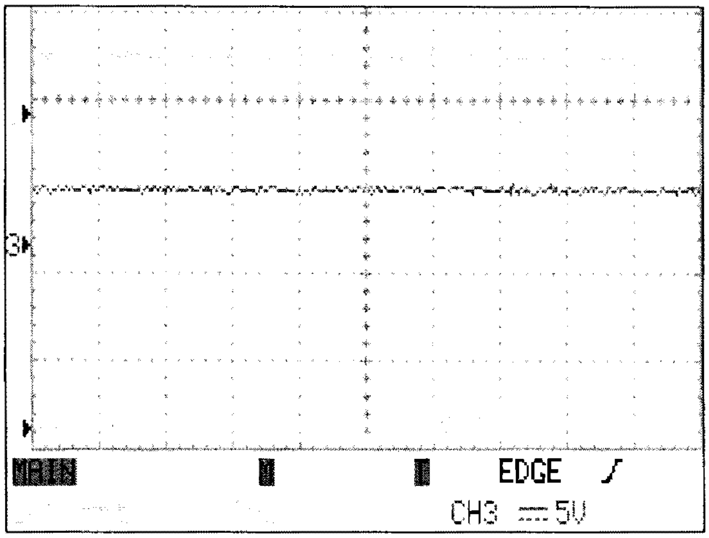


Figure 37: Control signals for the Stepper motor drive when brake applied.

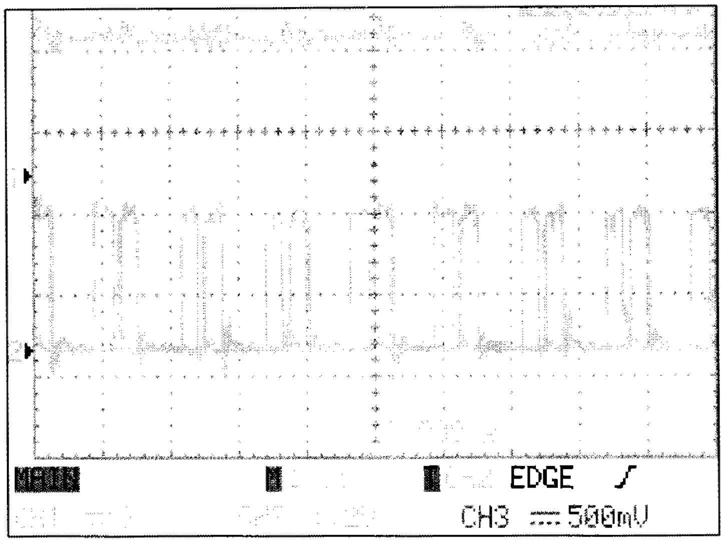


Figure 38: Noise in the DC motor drive signals.

When both, the DC hub motor and the stepper motor were driven, the noise generated by the DC motor severely affected the stepper motor control signals and destabilized the stepper motor drive as shown in Figure 39. The figure shows the control signals for the stepper motor drive when the DC Hub motor was running; the stepper motor braking signal was applied but, due to noise flowing back from the DC Hub motor to the controls section, the break signal output alternated between LOW and HIGH values causing the stepper motor to run despite the braking signal; this noise was stochastic in nature, hence the stepper motor response was very erratic. Interestingly, it was found that by testing the WCU prototype using the spiral cage DC motor instead of the of the DC hub motor, less noise was generated. This indicated that the Hub motors have a tendency to generate more noise than normal since, the inner coils were completely isolated to the ground. Any distortion therefore, flowed back into the circuit instead of going back to the ground.

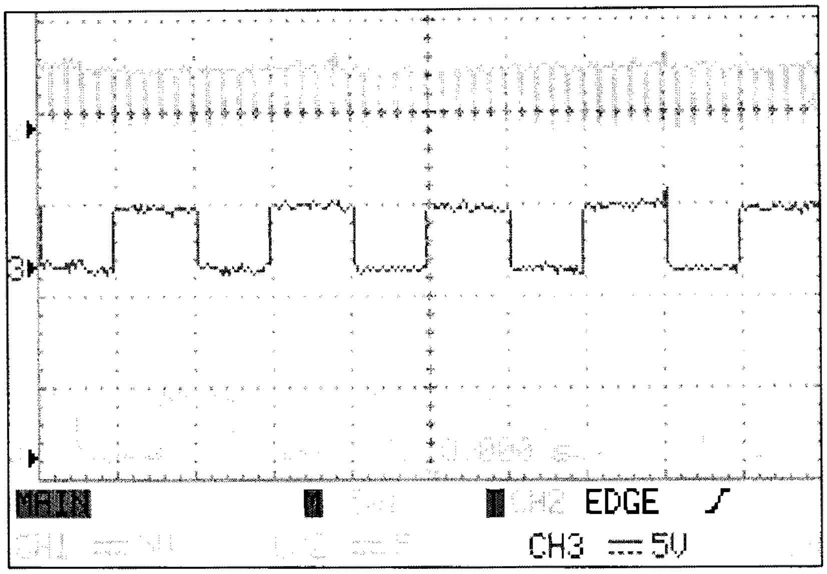


Figure 39: Destabilized control signals for the Stepper motor drive.

Numerous elements can contribute to crosstalk in a circuit; these include the wire inductance, presence of internal noise within the circuit itself, and induction of noise from the externally connected components. Similarly, many solutions are possible, but probably the simplest solution is to keep the wires short or to use proper ground and power planes at both sides of the PCB in order to reduce the parasitic inductances. The other solution is to use the bypass capacitors [86], which dampen the noise strength on the board. These solutions work well with low strength noises; however, if the noise level is considerably high, then complete isolation of the noise generating side from the side affected by that noise must be done by completely isolating both supply voltages and grounds from the two sides.

To address the WCU crosstalk problem, several methods were investigated. The first method was by grounding the DC motor using bypass capacitors. This approach was not feasible, since the hub motors have hidden inner motor that could not be reached for grounding through the bypass capacitors. The second approach was to isolate the drive power from the control section using the opto-isolator chips. In this approach, the diode side of the opto-isolator chip was connected to the control side and the transistor side was connected to the drive side using the 5 volts voltage regulator. The isolation of power only was ineffective, although the intensity of the noise was lowered down; the cross talk was still present. It was found later that the noise from the DC motor was flowing back to the power source through both the power side and the ground side, therefore by isolating the power side only did not eliminate the noise that reached the control side through the common grounding. Finally, a complete isolation of the drive, which included isolating both the ground and the power sides between the motor drives and the controller, was done. This approach introduced a separate low voltage power supply to drive the voltage regulator for the control section. Two 10.5 volts 600 m Amps toy batteries, connected in parallel, were used for this purpose. This, along with appropriate grounding and powering of both sides of the PCB using the ground and the power planes, eliminated most of the noise.

6. EXPERIMENTAL RESULTS

After designing the proposed control system for BIBOT-1, the robot was made to track the desired path to check for the effectiveness and the feasibility of the approach. Although BIBOT-1 can have five steering modes as discussed in Chapter 3, only four modes were tested: 1) the front wheel steering, 2) the all wheel steering, 3) the parallel steering and 4) the zero radius turn steering. The straight run steering was not tested due to the lack of appropriate speed sensors as mentioned previously. This chapter discusses the results of these experimental runs.

In all test cases, the VBC was preprogrammed to provide the vehicle movement information, i.e., the turning angle (φ_G), the speed (v_G) and the steering mode. The steering modes were coded in the VBC using binary codes as shown in Table 4. Although, this information is intended to be generated by the robot itself while tracking the path using the robot's navigation system, at the time of this experimentation, the navigation system for the BIBOT-1 was not fully developed. On the other side, each WCU was pre-programmed to independently react to the information provided by the VBC on the data bus depending on its own position on the robot; different wheels had different reactions.

To track the robot path during the experimental runs, the robot was made to move on a gridded floor of 6x6 in^2 grid size using a marker attached at the robot's center of mass. The path tracked by the robot was graphed in an excel file using the x-y coordinates marked on the gridded floor. The paths that were perceived to be simple, were tested with three runs, while those that were perceived to be relatively complex were tested with up to five runs so as to have a better estimate for the performance measure of the control system with 95% of confidence. The experimental results obtained along with the desired paths are shown in

Figures 40-46. In all experimental runs the straight run speed of the robot was kept constant at 21 inch/sec. The desired path was plotted using the standard kinematic relationships between the speed, time and distance covered.

Binary Code	Steering Mode
0000	Fixed Straight Steering
00010	Front Wheel Steering
0100	All Wheel Steering
1000	Zero Radius Turn Steering
1100	Parallel Steering

Table 4: Binary codes for the selection of steering mode.

The performance measure of the robot was based on R.M.S. error of the robot deviation from the desired path, which was defined as,

$$\varepsilon = \sqrt{\frac{\int_0^{t_f}[f(x,y)-f_a(x_a,y_a)]^2 dt}{t_f}} , \qquad \textbf{(6.1)}$$

where t_f is the total time duration of the desired path, $f(x,y)$ is the desired path coordinate, while $f_a(x_a, y_a)$ is the actual path coordinate tracked by the robot. However the error was generally computed as,

$$\varepsilon = \sqrt{\frac{\sum_{i=1}^{n}[(x-x_a)_i^2+(y-y_a)_i^2]}{n}} , \qquad \textbf{(6.2)}$$

where $(x - x_a)$ and $(y - y_a)$ are the x and the y axis deviations from the desired path, respectively and n is the number of sample points. The less the value of ε, the better is the performance.

The robot was tested on two types of paths: the U-turn path shown in Figures 40-42 and the zigzag motion shown in Figures 43-46. In the U-turn path the robot was tested for the front wheel steering, all wheel steering and the zero radius all wheel steering. For both, the front wheel steering and the all wheel steering, the robot was first made to follow a straight path for a distance of 174 inches followed by a right turn of 30°. The front wheel steering had to complete the turn over a distance of 160 inches, while the all wheel steering had to complete the turn over a distance of 63 inches. After completing the right turn, the robot followed another straight line path of 174 inches before completing the U-turn mission. The path tracking results for the U-turn motion under the front wheel steering are shown in Figure 40, while those for the all wheel steering are shown in Figure 41. As seen from these results, the robot was able to successfully track the paths with the minimal errors. The tracking error ε was found to be 11.15 $\pm$ 6.53 inches for the front wheel steering and 2.82 $\pm$ 0.35 inches for the all wheel steering. These errors may be caused by the lack of the speed sensors and the assumption that the surface provides no slip, among other possible reasons. It was assumed that the surface on which the robot was tested was not polished enough, providing no slip condition, which could have resulted in the stated error.

The zero radius turning steering had a straight path of 132 inches followed by another straight path of the same length on the return run. Since the zero radius turn steering had a fixed steering angle governed by the vehicle geometry, all that needed was a specification of the final turning angle, which in this case was 180° defined by the turning speed v_k and the time taken to complete the turn. The result obtained for this motion is shown in Figure 42. Again, the robot tracked the desired path successfully with a tracking error of only 1.73 $\pm$ 1.43 inches for the zero radius turn steering. In the steering configuration, there was a sudden change in the rotational direction of the rear wheels during the turning, which caused some slip in the vehicle motion. This, along with the lack of speed sensors and the low friction surface, could be the possible reasons of error in the experimental run.

In the zigzag paths, the robot was tested on two forms of zigzag motions: the zigzag path defined by straight line segments and the zigzag path defined by continuously changing directions. The robot path under the front wheel steering mode for the simple zigzag path with straight line segments is shown in Figure 43, and that of the corresponding all wheel steering parallel motion is shown in Figure 44. This path was made of three straight line segments of 66 inches, 54 inches and 30 inches, separated by a left and right turn of steering angle of 23^0 in both cases. Again the robot was able to successfully track the desired path with small tracking errors ε of 5.78 $\pm$ 1.72 inches for the front wheel steering and 6.12 $\pm$ 0.86 inches for the all wheel steering parallel turn. The error in the experimental run was again accounted for due to the surface condition and lack of speed sensors as discussed previously.

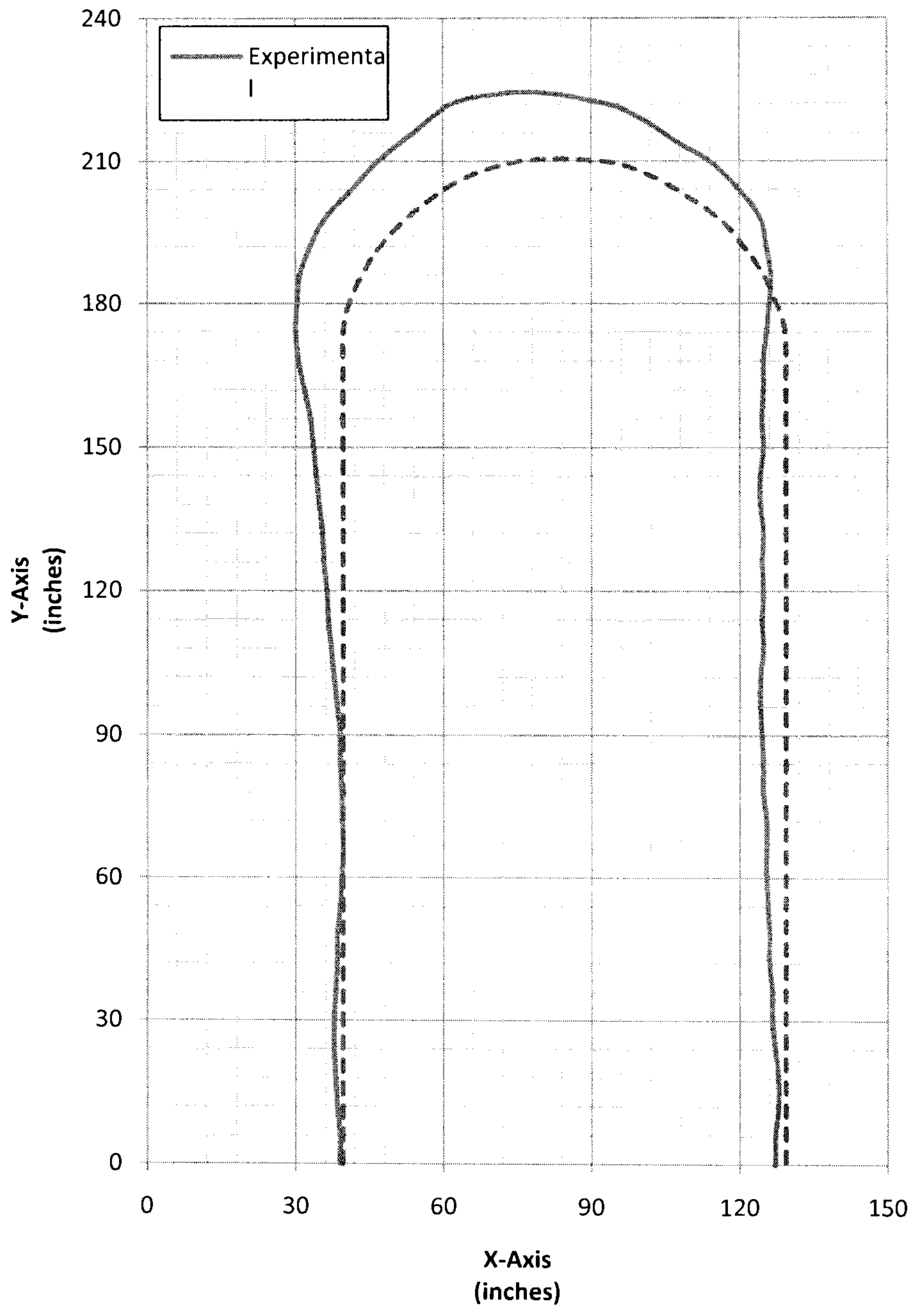


Figure 40: U-Turn in the Front wheel steering mode.

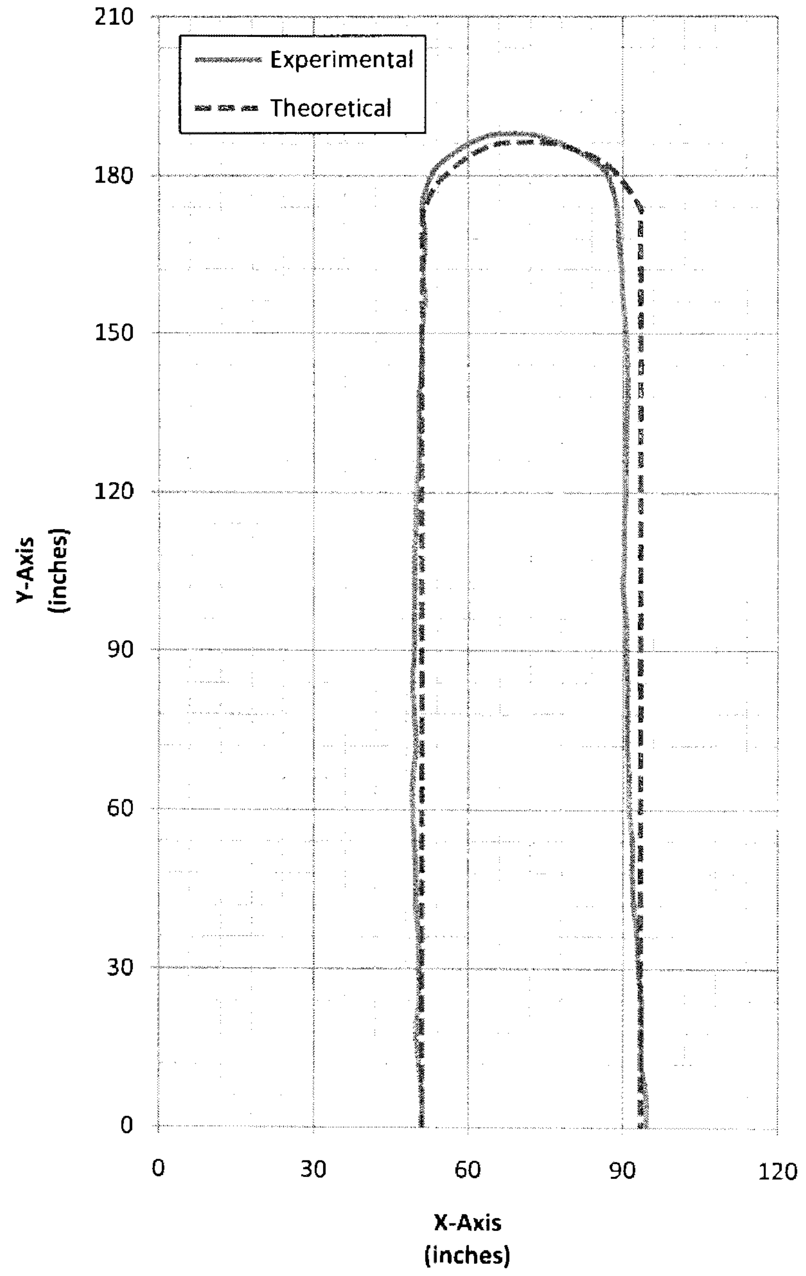


Figure 41: U-Turn in the All wheel steering mode.

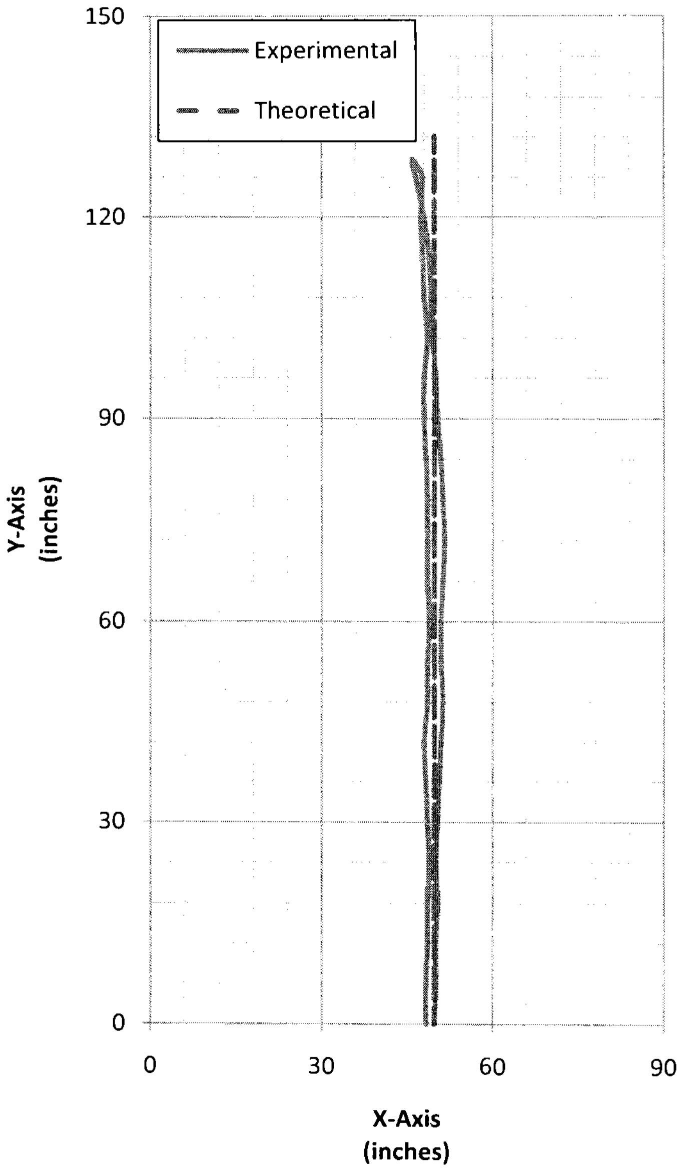


Figure 42: U-Turn in the Zero radius turning mode.

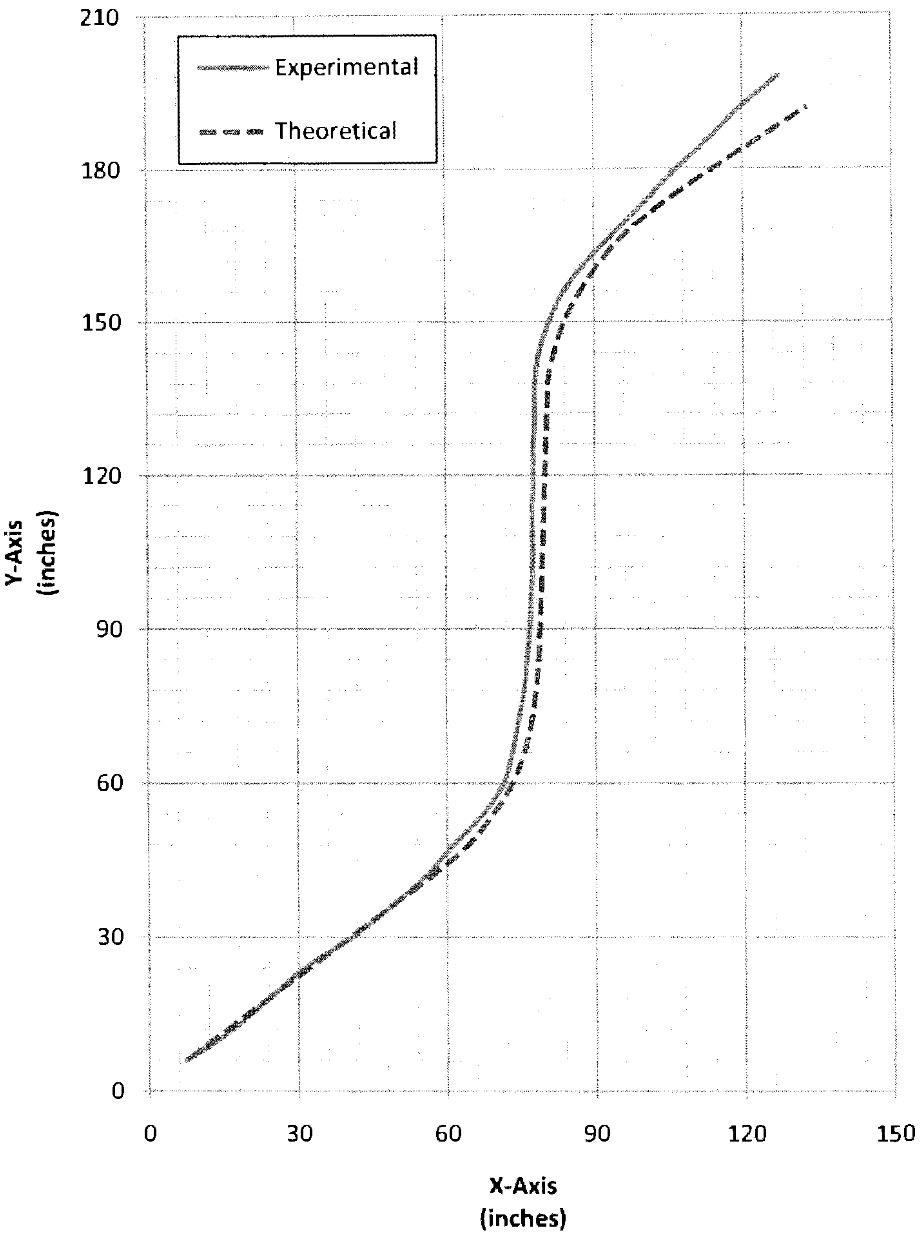


Figure 43: Zigzag motion in the Front wheel steering mode.

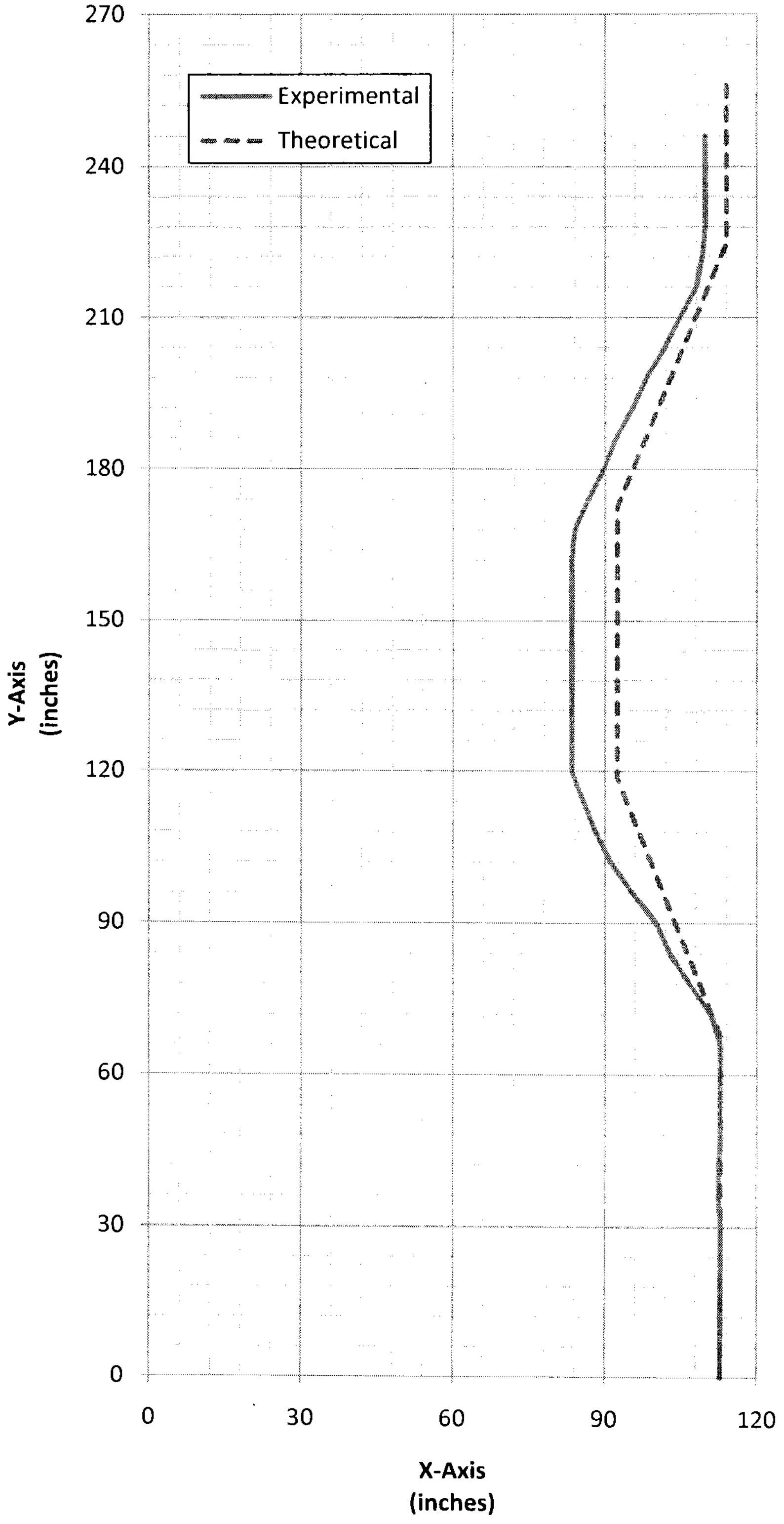


Figure 44: Zigzag motion in the Parallel steering mode.

For the continuous zigzag path, the robot directional angle was continuously changed from 23° to -23° with a separation of a straight line segment of 21 inches. The result for the front wheel steering is shown in Figure 45 and for the all wheel steering is shown in Figure 46. Again, the robot was able to successfully track the desired path. The tracking error ε computed for the front wheel steering was 10.89 ± 4.99 inches and for the all wheel steering was 34.49 ± 5.87 inches, which was relatively high as compared to others. The reason for the high level of tracking error can be attributed due to the accumulation of the localization error. Since the robotic vehicle lacks the appropriate speed sensors, it is possible that at repeated turns, the exact speed may have been either higher or lower than the expected speed, hence introducing localization errors that kept accumulating over the duration of the test.

The results obtained on tracking various paths as discussed above confirm the feasibility of the Particle Model approach for robotic vehicles. Although the experimental robot platform lacked the necessary sensors, it was still able to steer itself and follow the desired paths with little tracking errors.

These path tracking results are comparable to those obtained by other research efforts [97-99]. For example, [97] used a model predictive controller on a generic car robot model treated as a rigid body. The robot was made to track zigzag path geometries similar to the one shown Figure 44, and ADAMS simulation results showed path tracking R.M.S. errors that ranged from 9.824 inches to 26 inches, which are in the same range as those obtained in this thesis.

Based on these results, it is possible to control and drive a robot by using a Particle Model approach. Although these experimental results were limited to 2 DOF controls, the flexibility of the Particle Model approach, which allow up to 5 DOF, makes it more appealing for complex robot maneuvers. Further tests are highly recommended on vehicles with adjustable axles to study the performance of this approach in all 5 DOF.

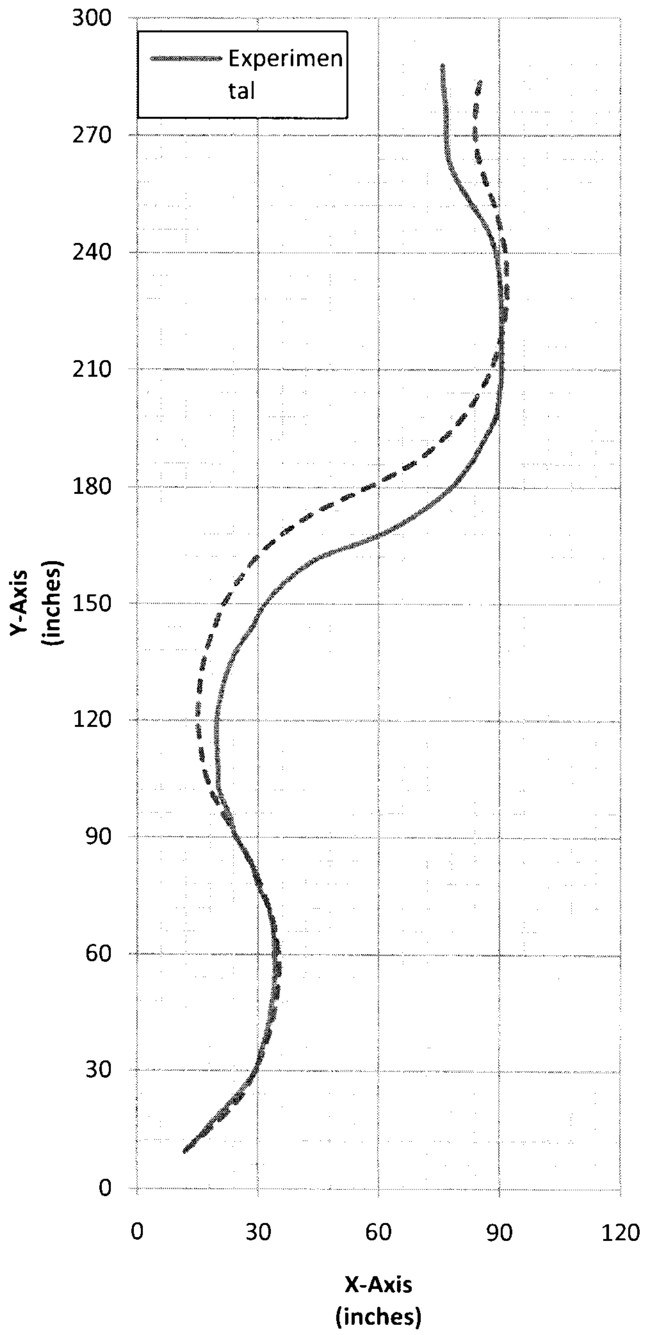


Figure 45: Zigzag motion in the Front wheel steering mode.

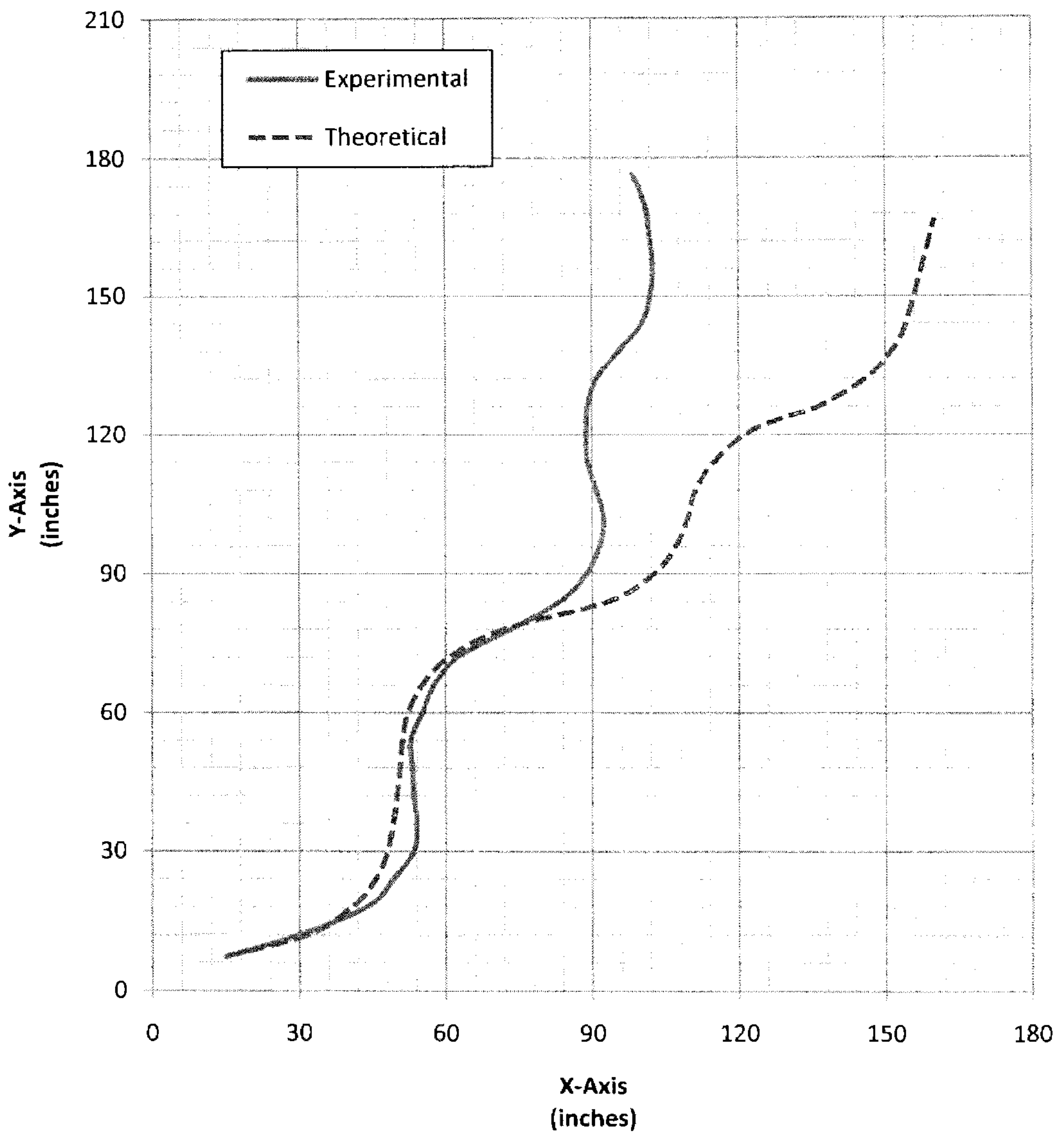


Figure 46: Zigzag motion in the All wheel steering mode.

7. SUMMARY AND CONCLUSIONS

This thesis has presented the initial results, showing the applicability of the Particle Model approach in developing robot controllers. Although the approach assumes that the robot wheels are on adjustable axles such that the wheel position is variable with respect to the center of gravity of the robot, this research used an existing fixed axle robot, so the method was not fully tested as it would have required. After treating wheels as a team of particles, decentralized control architecture was applied and each wheel had its own controller independent of other wheels. The robot was loaded with four steering modes, and the performances of these modes were compared under similar path geometries.

The experimental results on this fixed axle vehicle have shown that the Particle Model approach works and based on the results of [97], this approach compares well with the standard solid modeling approach. This proves that it is feasible to use this approach on wheeled robotic vehicles. To apply this approach, a decentralized control structure must be adopted with each wheel controlled independently. The next task should be to go ahead and test the proposed approach on more complex vehicles with adjustable axles. For the accounted errors it is thought that some of these tracking errors were caused by the inherent robot localization errors, which were caused by the lack of speed sensors, and were not related to the controllers themselves. The assumption of no slip, the smooth surface and a constant battery supply could also have introduced performance errors.

The Particle Model Control approach shows a promising future use for the path tracking problem of robots on unstable surfaces. Still further future experiments need to be done using the control approach on robotic systems equipped with all appropriate sensors and adjustable axles, on a variety of terrains such as snowy, sandy, and icy terrains.

REFERENCES

[1]. Press information by International Federation of Robotics, October 2007 [Online: February 4, 2010], Available at: http://www.worldrobotics.org/modules.php?name=News&file=article&sid=3

[2]. E. Garcia, M. A. Jimenez, P. G. D. Santos, M. Armada, The Evolution of Robotics Research, IEEE Robotics and Automation Magazine, vol. 14, issue 1, pp. 90-103, March 2007.

[3]. S.Y. Nof., Book: Handbook of Industrial Robotics, volume 1, John Wiley and Sons, 1999.

[4]. S. Dunkerley, M. J. Adams, A general robot control system for automatic chemical analysis, Laboratory Automation and Information Management, vol. 33, issue 2, pp. 93-105, December 1997.

[5]. S. Tansey, O. Holland, A system for automated mail portering using multiple mobile robots, Proc. IEEE 8th International Conference on Advanced Robotics, 1997, pp. 27-32, July 1997.

[6]. J. Trevelyan, Redefining Robotics for the New Millennium, The International Journal of Robotics Research, vol. 18, pp. 1211-1223, December 1999.

[7]. I.F. of Robotics, "Service Robots", 2010 [Online], Available: http://www.ifr.org/service-robots/products/

[8]. S.D. Pippo, G. Colombina, R. Boumans, P. Putz, Future potential applications of robotics for the International Space Station, Robotics and Autonomous Systems, vol. 23, issue 1-2, pp. 37-43, March 1998.

[9]. W. Korb, R. Marmulla, J. Raczkowsky, J. Mühling, S. Hassfeld, Robots in the operating theatre-chances and challenges, International Journal of Oral Maxillofacial Surgery, vol. 33, issue 8, pp. 721-732, December 2004.

[10]. I. Ulrich, F. Mondada, J.-D. Nicoud, Autonomous vacuum cleaner, Robotics and Autonomous Systems, vol. 19, issues 3-4, pp. 232-245, March 1997.

[11]. K. Tanie, Humanoid robot and its application possibility, Proc. IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, MFI 2003.

[12]. P. Gupta, V. Tirth, R.K. Srivastava, Futuristic Humanoid Robots: An Overview, First International Conference on Industrial and Information systems, 2006.

[13]. J.-K. Yoo, B.-J. Lee, J.-H. Kim, Recent progress and developments of the humanoids robot HanSaRam, Robotics and Autonomous Systems, vol. 57, issue 10, pp. 973-981, October 2009.

[14]. K. Hirai, M. Hirose, Y. Haikawa, T. Takenaka, The development of Honda Humanoid Robot, Proc. IEEE International Conference on Robotics and Automation, Leuven, Belgium, May 1998.

[15]. R.M. Mahoney, Robotic Products for Rehabilitation: Status and Strategy, Proc. International Conference on Rehabilitation Robotics, 1997.

[16]. QinetiQ's firefighting robots, 2010 [Online: February 4, 2010], Available at: http://www.qinetiq.com/home/newsroom/news_releases_homepage/2007/4th_quarter/fire_rovs.html

[17]. D.B. Camarillo, T.M. Krummel, J.K Salisbury, Robotic technology in surgery: past, present and future, The American Journal of Surgery, 188: 2S - 15S, 2004.

[18]. A. D. Santis, B. Siciliano, A.D. Luca, A. Bicchi, An atlas of physical human-robot interaction, Mechanism and Machine Theory, vol. 43, issue 3, pp. 253-270, March 2008.

[19]. H.H. Lund, Adaptive Robotics in the Entertainment Industry, Proc. IEEE International Symposium on Computational Intelligence in Robotics and Automation, IEEE Press 2003.

[20]. R.D. Schraft, G. Schmierer, Book: Service Robots, A K Peters, Ltd. 2000.

[21]. R. Siegwart, I. R. Nourbakhsh, Book: Introduction to Autonomous Mobile Robots, MIT Press 2004.

[22]. J. R. Matijevic, J. Crisp, D. B. Bickler and rest of Rover Team, Characterization of the Martian Surface Deposits by the Mars Pathfinder Rover, Sojourner, Science Magazine, vol. 278, pp. 1765- 1768, December 1997.

[23]. S. M. Phail, Autosub6000: A Deep Diving Long Range AUV, Journal of Bionic Engineering, vol. 6, issue 1, pp. 55- 62, March 2009.

[24]. S.H.H. Young, Gallery of USAF Weapons, Air Force Magazine, May 2008.

[25]. T. Fong, C. Thorpe, Vehicle Teleoperation Interfaces, Autonomous Robots, vol. 11, issue 1, pp. 9- 18, July 2001.

[26]. D.W. Gage, UGV HISTORY 101: A Brief History of Unmanned Ground Vehicle (UGV) Development Efforts, Unmanned Systems Magazine, vol. 13, issue 3, 1995.

[27]. W.A. Moir, Patent: Differential Gearing for Automobile Driving Axles, Serial no. 438313, February 1922.

[28]. J.Y. Wong, C.F. Chiang, A General theory for skid steering of tracked vehicles on firm ground, Proc. of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering, vol. 215, issue 3, pp. 343-355, 2001.

[29]. F. Large, S. Sekhavat, C. Laugier, E. Gauthier, Towards robust sensor-based maneuvers for a car-like vehicle, Proc. IEEE International Conference on Robotics and Automation, vol. 4, pp. 3765-3770, April 2000.

[30]. M.A. Sotelo, Lateral Control Strategy for Autonomous Steering of Ackerman-Like Vehicles, Robotics and Autonomous Systems, vol. 45, issue 34, pp. 223-233, December 2003.

[31]. A. Lacaze, K. Murphy, M. D-Giorno, Autonomous Mobility for the DEMO III Experimental Unmanned Vehicles, Proc. AUVSI 2002, Orlando, Florida, pp. 8- 12, July 2002.

[32]. A. Stentz, Optimal and Efficient Path Planning for Partially-Known Environments, Proc. of IEEE International Conference on Robotics and Automation, May 1994.

[33]. O. Hachour, Path planning of Autonomous Mobile Robot, International Journal of Systems Application, Engineering and Development, vol. 2, issue 4, 2008.

[34]. A. Lazaro, I. Serrano, J.P. Oria, Ultrasonic circular inspection of object recognition with sensor-based integration, Sensors and Actuators A: Physical, vol. 77, issue 1, pp. 1-8, September 1999.

[35]. F. Castelli, An Integrated Tactile-Thermal Robot Sensor with Capacitive Tactile Array, IEEE Transactions on Industry Applications, vol. 38, issue 1, pp. 85-90, 2002.

[36]. M. Kam, X. Zhu, P. Kalata, Sensor fusion for mobile robot navigation, Proc. IEEE, vol. 85, pp. 108-119, 1997.

[37]. R. Jarvis, Intelligent Robotics: Past, Present and Future, International Journal of Computer Science and Applications, vol. 5, issue 3, pp. 23-35, 2008.

[38]. Defense Advanced Research Projects Agency (DARPA) Closed and Open Solicitations [Online: June 20, 2010], Available at: http://www.darpa.mil/openclosedsolicitations.html

[39]. NASA Jet Propulsion Laboratory Mobility and Robotic Systems, Section 347 [Online: June 20, 2010], Available at: http://www-robotics.jpl.nasa.gov/

[40]. J.Y. Wong, Book: Theory of Ground Vehicles, John Wiley & Sons, Inc., New York, Third Edition, 2001.

[41]. H. Leiber, All-Wheel Drive Automotive Vehicle Traction Control System, US Patent Number 4589511, May 1986, Robert Bosch GmBH, Stuttgart, FRG.

[42]. H.W. Bleckmann, H. Fennel, J. Graber, and W.W. Seibert, Traction Control System with Teves ABS Mark II, SAE Technical Papers, 1986, paper No. 860506.

[43]. Y.A. Ghoneim and Y-K. Chin, Vehicle Traction Control System, US Patent Number 5025882, June 1991, General Motors Corporation, USA.

[44]. Y. Hori and Y. Toyoda, Traction Control of Electric Vehicle: Basic Experimental Results Using the Tes EV "UOT Electric March", IEEE Transactions on Industry Applications, vol. 35, issue 5, pp. 1131-1138, 1998.

[45]. H. Lee and M. Tomizuka, Adaptive Vehicle Traction Force Control For Intelligent Vehicle Highway Systems (IVHSs), IEEE Transaction on Industrial Electronics, vol. 50, issue 1, pp. 37-47, February 2003.

[46]. E. Hertz, J. Hilton , D. Johnson, Analysis of the Crash Experience of Vehicles Equipped with Antilock Braking Systems (ABS), Proc. of 15th ESV Conference, number 96-S9-O-03, pp. 1392-1395, 1996.

[47]. C. Farmer, New Evidence Concerning Fatal Crashes of Passenger Vehicles Before and After Adding Antilock Braking System, Accident Analysis and Prevention, vol. 33, pp. 361-369, 2001.

[48]. J. Benton, Prius Shuts Down in the Snow, Reader Complains, Consumer News, January 2007 [Online: June 23, 2010], Available at: www.consumeraffairs.com/news04/2007/01/prius_snowbound_html.

[49]. B.C. Besselink, J.H. Fielke, Improving Tractive Efficiency By Integrating the Steering and Drive Systems of Four-Wheeled Vehicles, In Proc. of 2003 ASAE Annual International Meeting, Las Vegas, Nevada, USA, pp. 27-30, July 2003.

[50]. J. Gray, The Mechanism of Locomotion in Snakes, Journal of Experimental Biology, vol. 23, issue 2, pp. 101-123, December 1946.

[51]. B.C. Jayne, Kinematics of Terrestrial Snake Locomotion, Copeia, vol. 4, pp. 915-927, December 1986.

[52]. N. Sarkar, X. Yun and V. Kumar, Control of mechanical systems with rolling contacts: Applications to mobile robots, International Journal of Robotics Research, vol. 13, issue 1, pp. 55-69, February 1994.

[53]. H. Baruk, Book: Analytical Dynamics, McGraw Hill, 1999.

[54]. K. Kozlowski, D. Pazderski, Modeling and Control of a 4-Wheel Skid Steering Mobile Robot, International Journal of Applied Mathematics and Computer Science, vol. 14, issue 4, pp. 477-496, 2004.

[55]. L. Caraccilo, A.D. Luca, S. Iannitti, Trajectory Tracking Control of a Four-Wheel Differentially Driven Mobile Robot, Proc. IEEE International Conference on Robotics and Automation, Detroit, Michigan, May 1999.

[56]. R.E. Colyer, J.T. Economou, Soft modeling and fuzzy logic control of wheeled skid-steer electric vehicles with steering prioritization, International Journal of Approximate Reasoning, vol. 22, pp. 31-52, February 1999.

[57]. A. Bemporad, A.D. Luca, G. Oriolo, Local incremental planning for a car-like robot navigation among obstacles. Proc. IEEE International Conference of Robotic Automation, vol. 2, pp. 1205-1211, April 1996.

[58]. J. Gutiérrez, D. Apostolopoulos, J.L. Gordillo, Numerical comparison of steering geometries for robotic vehicles by modeling positioning error, Autonomous Robot, vol. 23, issue 2, pp. 147-159, August 2007.

[59]. A. Okubo, Dynamical aspects of animal grouping: Swarms, schools, flocks, and herds, Advances in Biophysics, vol. 22, pp. 1–94, 1986.

[60]. Herbert G. Tanner, Ali Jadbabaie, George J. Pappas, Stable flocking of mobile agents, part ii: Dynamic topology, In IEEE Conference on Decision and Control, pp. 2016–2021, 2003.

[61]. Herbert G. Tanner, Ali Jadbabaie, George J. Pappas, Flocking in teams of nonholonomic agents, In M. Thoma and M. Morari, editors, Cooperative Control, vol. 309 of Lecture Notes in Control and Information Sciences, pp. 458–460. Springer, 2004.

[62]. H. Yamaguchi, A cooperative hunting behavior by multiple nonholonomic mobile robots. In IEEE International Conference on Systems, Man, and Cybernetics, 1998, vol. 4, pp. 3347 –3352, 1998.

[63]. T. Koishi T. Murakami, An approach to cooperative control for formation flight of multiple autonomous helicopters, In 35th Annual Conference of IEEE Industrial Electronics, IECON '09, pp. 1456 –1461, 2009.

[64]. ZhiDong Wang, Y. Hirata, K. Kosuge, Control a rigid caging formation for cooperative object transportation by multiple mobile robots, In Proc. IEEE International Conference on Robotics and Automation, ICRA '04, 2004, vol. 2, pp. 1580 – 1585, April 2004.

[65]. J.J. Liang, P.N. Suganthan, Dynamic multi-swarm particle swarm optimizer with local search, In IEEE Congress on Evolutionary Computation, 2005, vol. 1, pp. 522 –528, 2005.

[66]. N.M. Kwok, V.T. Ngo, Q.N. Ha, PSO-Based Cooperative Control of Multiple Mobile Robots in Parameter-Tuned Formations, In IEEE International Conference on Automation Science and Engineering, CASE 2007, pp. 332 –337, 2007.

[67]. John Ehlen, Dan Henderson, Luke Scchraw, Kevin Watson, Chad Nelson, and Shafa Wala, "Design of an AWD AWS Autonomous Vehicle (unpublished)". Senior Design Project, Department of Mechanical Engineering, NDSU, 2006-2008.

[68]. Vikramjeet Singh, Aaron Vander Vorst, Aaron Zuther, Ben Hest, "AWD AWS Autonomous Robotic Vehicle (unpublished)". Senior Design Project, Department of Electrical/Mechanical Engineering, NDSU, 2007-2008.

[69]. M. J.- Kharaajoo, F. Besharati, Sliding Mode Traction Control of an Electric Vehicle with Four Separate Wheel Drives, Proc. IEEE Conference on Emerging Technologies and Factory Automation, ETFA, vol. 2, pp. 291-296, September 2003.

[70]. S.-I. Sakai, H. Sado, Y Hori, Motion Control in an Electric Vehicle with Four Independently Driven In-Wheel Motors, IEEE/ASME Transactions on Mechatronics, vol. 4, no. 1, March 1999.

[71]. R.E. Colyer, J.T. Economou, Soft modeling and fuzzy logic control of wheeled skid-steer electric vehicles with steering prioritization, Int. Journal of Approximate Reasoning, vol. 22, issues 1-2, pp. 31-52, September October 1999.

[72]. J. Lunze, Book: Feedback Control of Large-Scale Systems, Prentice Hall, New York, 1992.

[73]. Y. Li, M. Cantoni, E. Weyer, Design of a Centralized Controller for an Irrigation Channel using H_∞ Loop-shaping, Proc. UKACC Control Conference, University of Bath, U.K., September 2004.

[74]. J.S. Albus, A.J. Barbera, R.N. Nagel, Theory and Practice of Hierarchical Control, Proc. Twenty Third IEEE Computer Society International Conference, pp. 18-25, September 1981.

[75]. D.H. Shim, H.J. Kim, S. Sastry, Hierarchical Control System Synthesis for Rotorcraft Based Unmanned Aerial Vehicles, American Institute of Aeronautics & Astronautics, Proc. AIAA Conference on Guidance, Navigation and Control, August 2000.

[76]. H. Hu, M. Brady, A parallel processing architecture for sensor-based control of intelligent mobile robots, Robotics and Autonomous Systems, vol. 17, issue 4, pp. 235-257, June 1996.

[77]. L. Bakule, Decentralized Control: An overview, Annual Reviews in Control, vol. 32, pp. 87-98, May 2008.

[78]. D.D. Siljak, Decentralized Control and Computations: Status and Prospects, Annual Reviews in Control, vol. 20, pp. 131-141, 1996.

[79]. J.P. Lynch, K.H. Law, Decentralized Control Techniques for Large-Scale Civil Structural Systems, Proc. 20th International Modal Analysis Conference, Los Angeles, CA, February 2002.

[80]. S.-J. Park, K.-H. Cho, Decentralized supervisory control of discrete event systems with communication delays based on conjunctive and permissive decision structures, Automatica, vol. 43, issue 4, pp. 738-743, April 2007.

[81]. R. Donner, Emergence of Synchronization in Transportation Networks with Biologically Inspired Decentralized Control, Book: Recent Advances in Nonlinear Dynamics and Synchronization - Theory and Applications, Springer, pp. 237-275, September 2009.

[82]. V.A. Ugrinovskii, I.R. Petersen, A.V. Savkin, E.Y. Ugrinovskaya, Decentralized state-feedback stabilization and robust control of uncertain large scale systems with integrally constrained interconnections, Systems and Control Letters, vol. 40, issue 2, pp. 107-119, June 2000.

[83].E. A. Parsheva, A. M. Tsykunov, Adaptive Decentralized Control of Multivariable Objects, Automation and Remote Control, vol. 62, pp. 290-303, February 2001.

[84]. D.D. Silijak, Decentralized Control and Computations: Status and Prospects, International Federation of Automatic Control, vol. 20, pp. 131-141, 1997.

[85]. J. Haibo, C. Hongzhi, J.F. Dorsey, Q. Zhihua, Toward a globally robust decentralized control for large-scale power systems, IEEE Transactions on Control Systems Technology, vol. 5, pp. 309- 319, 1997.

[86]. W. Shepherd, L. N. Hulley, D.T.W. Liang, Book: Power Electronics and Motor Control, Cambridge University Press, 1995.

[87]. D. K. Lindner, M. Zhu, N. Vujic, D.J. Leo, Comparison of linear and Switching Drive amplifiers for Piezoelectric Actuators, American Institute of Aeronautics and Astronautics, 2002.

[88]. J. Honda, J. Adams, Application Note: Class D Audio Amplifier Basics, International Rectifier.

[89]. A. Smaili, F. Mrad, Book: Applied Mechatronics, Oxford University Press, 2008.

[90]. T. Wilmshurst, Book: Designing Embedded Systems with PIC Microcontrollers (Second Edition), 2009.

[91]. Freescale Semiconductor, Motorola Semiconductor Technical Data MC9S12D-Family, 2002.

[92]. Microchip, dsPIC33FJ128MCX02/X04 - High-Performance, 16-bit Digital Signal Controllers, 2009.

[93]. J. D. Spencer, D. E. Pippenger, Book: The Voltage regulator handbook, Texas Instruments, 1977.

[94]. R. C. Dorf, Book: Sensors, nanoscience, biomedical engineering, and instruments, CRC Press, 2006.

[95]. Application Note AN-6076: Design and Application Guide of Bootstrap Circuit for High-Voltage Gate-Driver IC, Fairchild Semiconductor Corporation, 2008.

[96]. J. C. Gracia, J. A. M-Nelson, S. Nooshabadi, A Single Capacitor Bootstrapped Power Efficient CMOS Driver, Symposium on Circuits and Systems, 2005, 48th Midwest, Covington.

[97]. S.C. Peters, K. Iagnemma, Mobile robot path tracking of aggressive maneuvers on sloped terrain, Proc. IEEE International Conference on Intelligent Robots and Systems, France, September, 2008.

[98]. E. Maalouf, M. Saad, H. Saliah, A higher level path tracking controller for a four-wheel differentially driven steered mobile robot, Robotics and Autonomous Systems, vol. 54, pp. 23-33, November 2005.

[99]. C. Gao, Y. Su, H. Ma, Agricultural Robot Path Tracking Based on Predictable Path, Proc. IEEE International Conference on Networking and Digital Society, 2010.

APPENDIX A

Circuit Diagrams

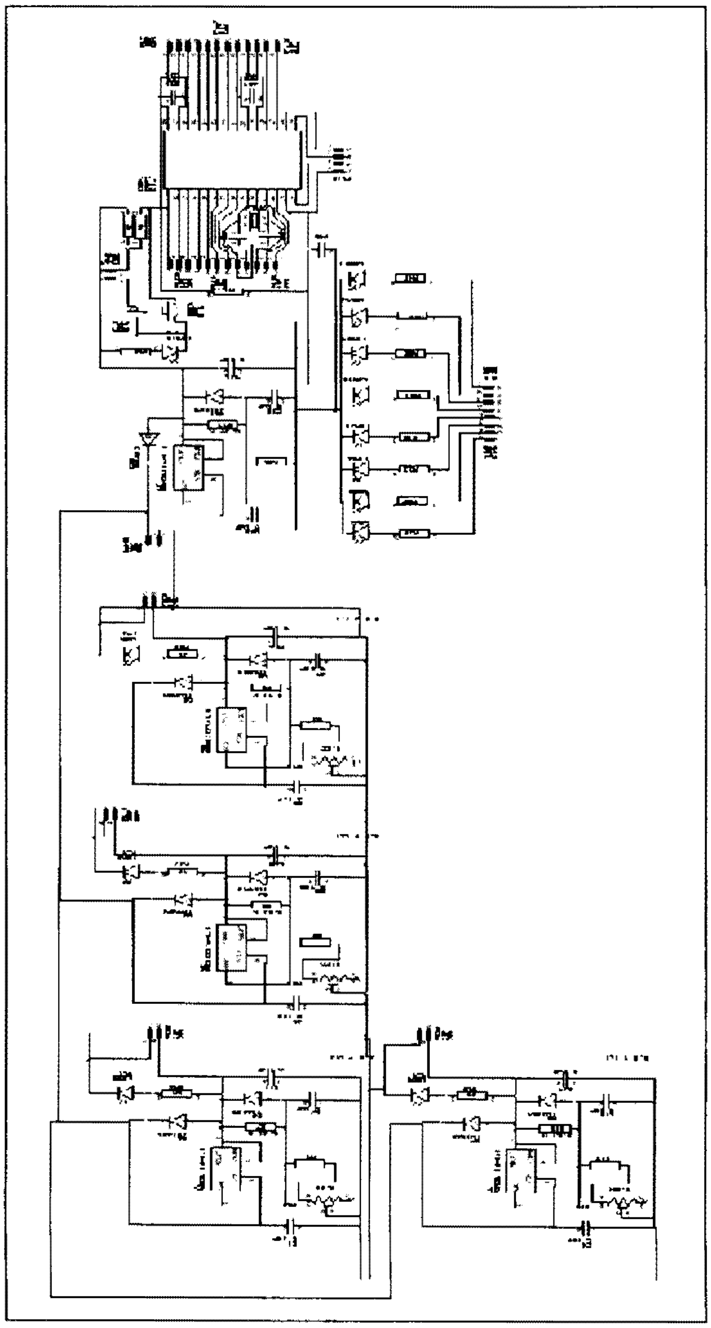


Figure 1: Schematics for the VBC

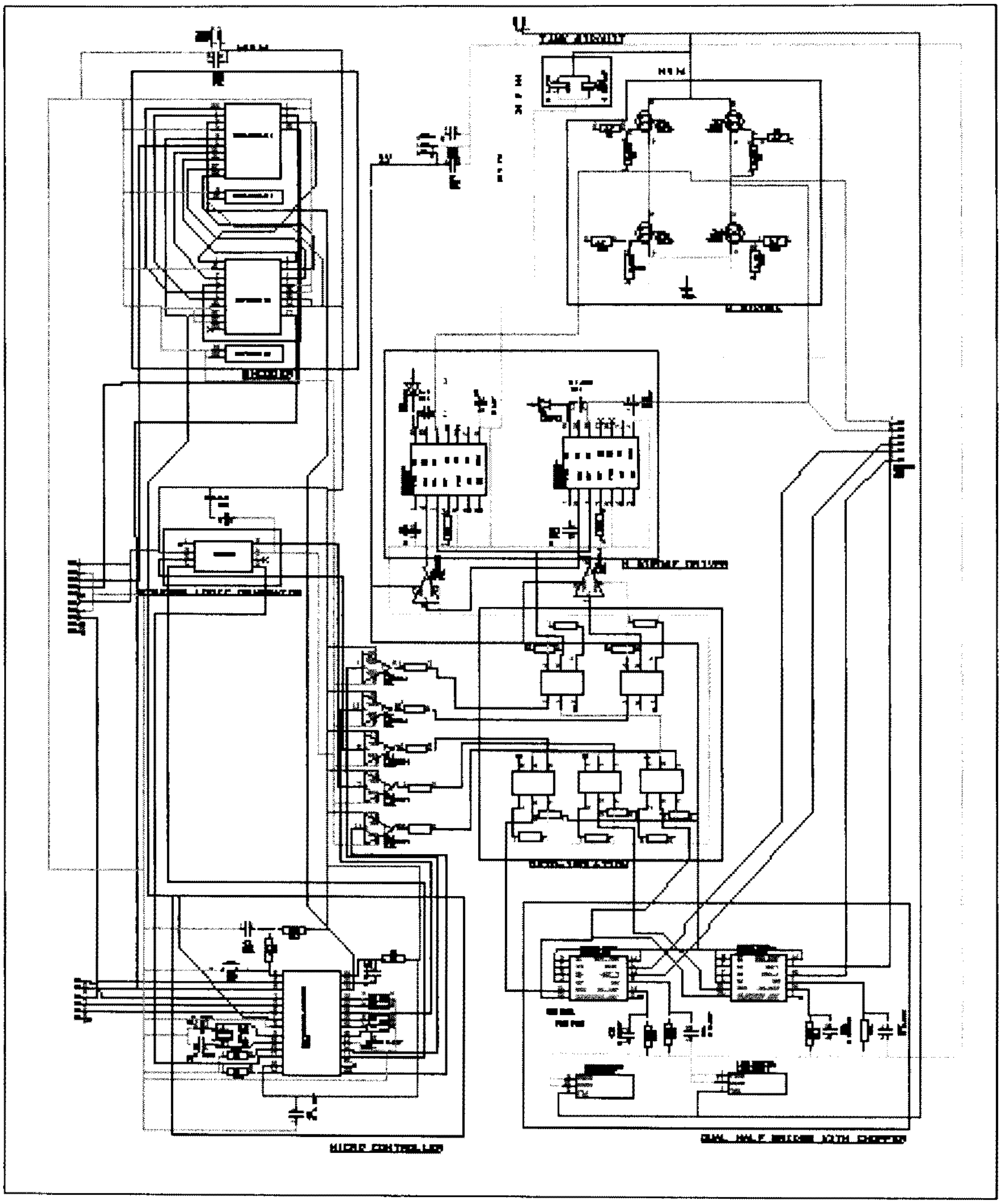


Figure 2: Schematics for the WCU.

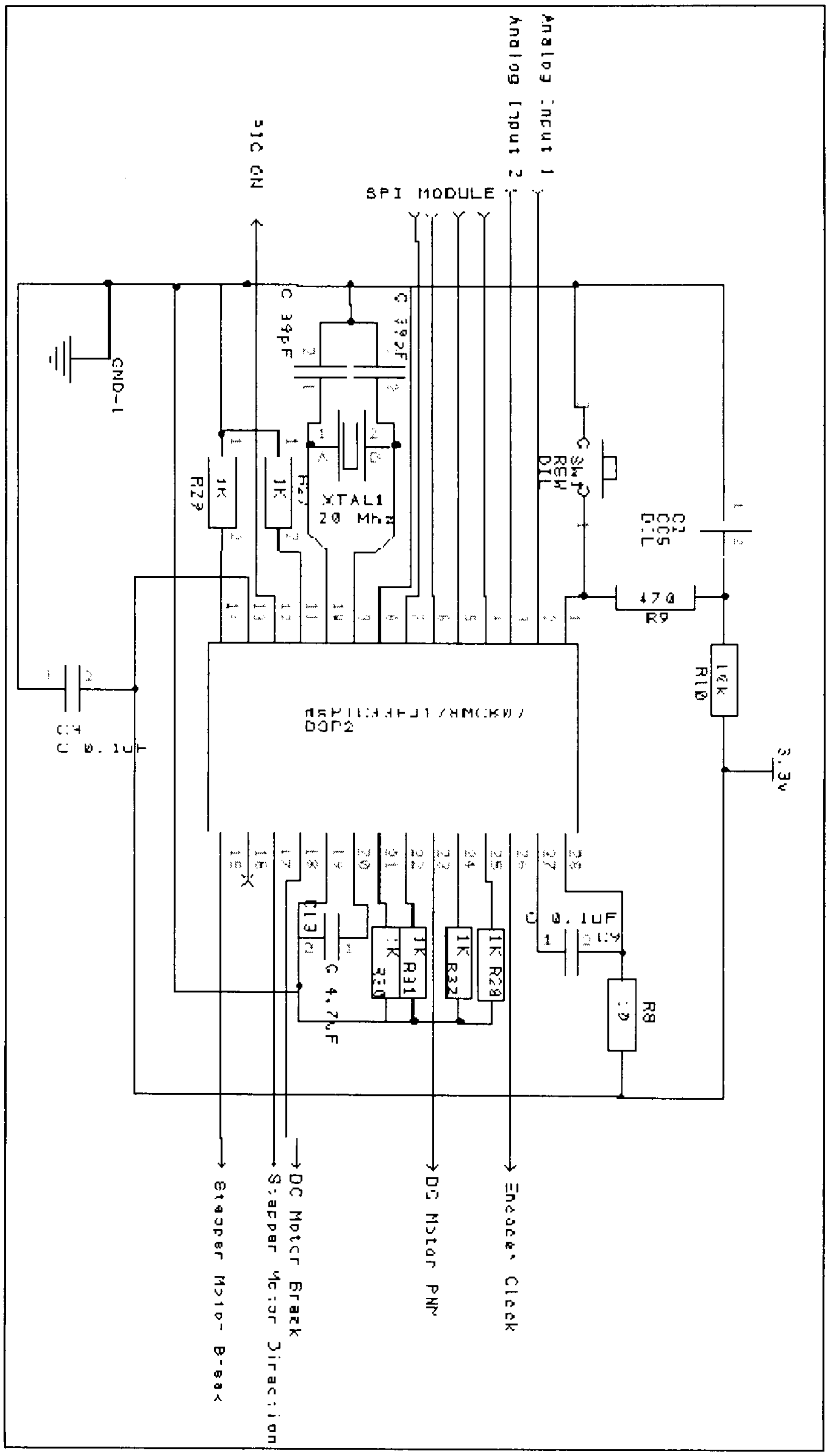


Figure 3: Schematics for the Control Section (dsPIC33FJ) of the WCU.

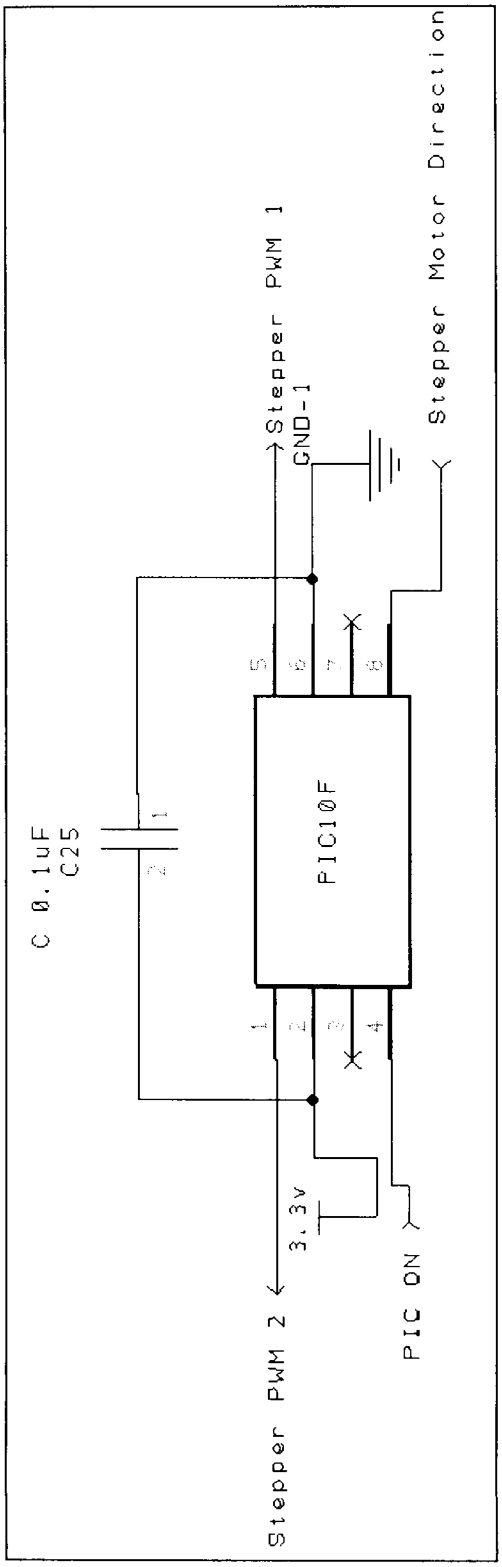


Figure 4: Schematics for the Stepper Motor Control PWM generation using PIC10F.

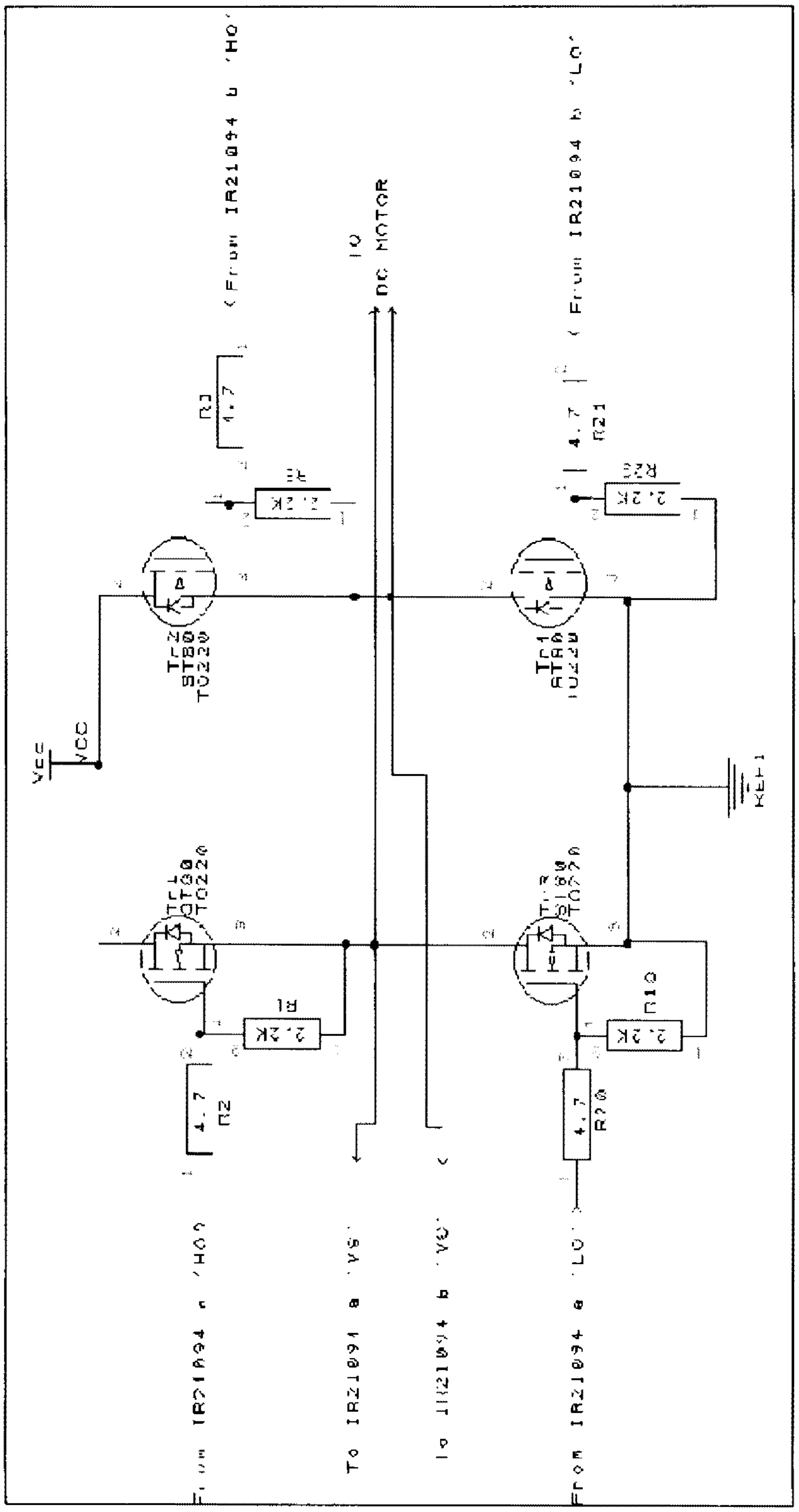


Figure 5: Schematics for the DC Hub motor H-Bridge Circuitry.

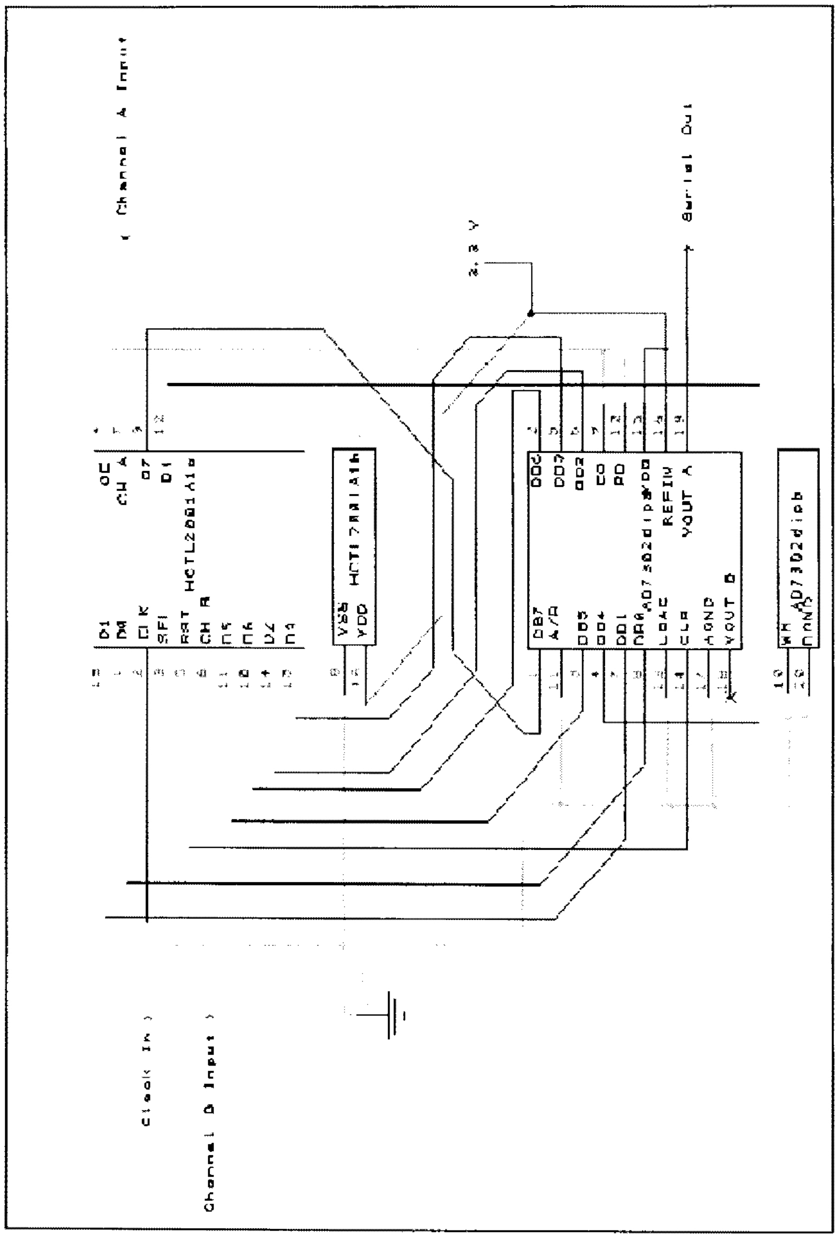


Figure 6: Schematics for speed encoder.

APPENDIX B

Board Designs

1. VBC

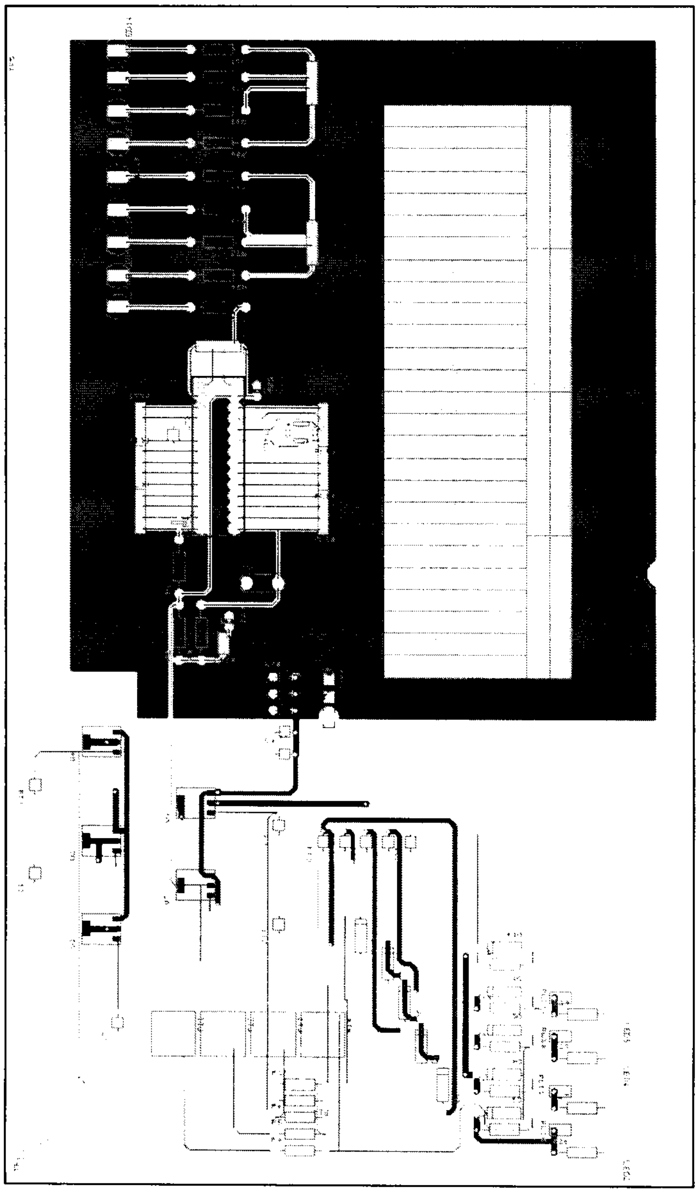


Figure 1: Top Copper.

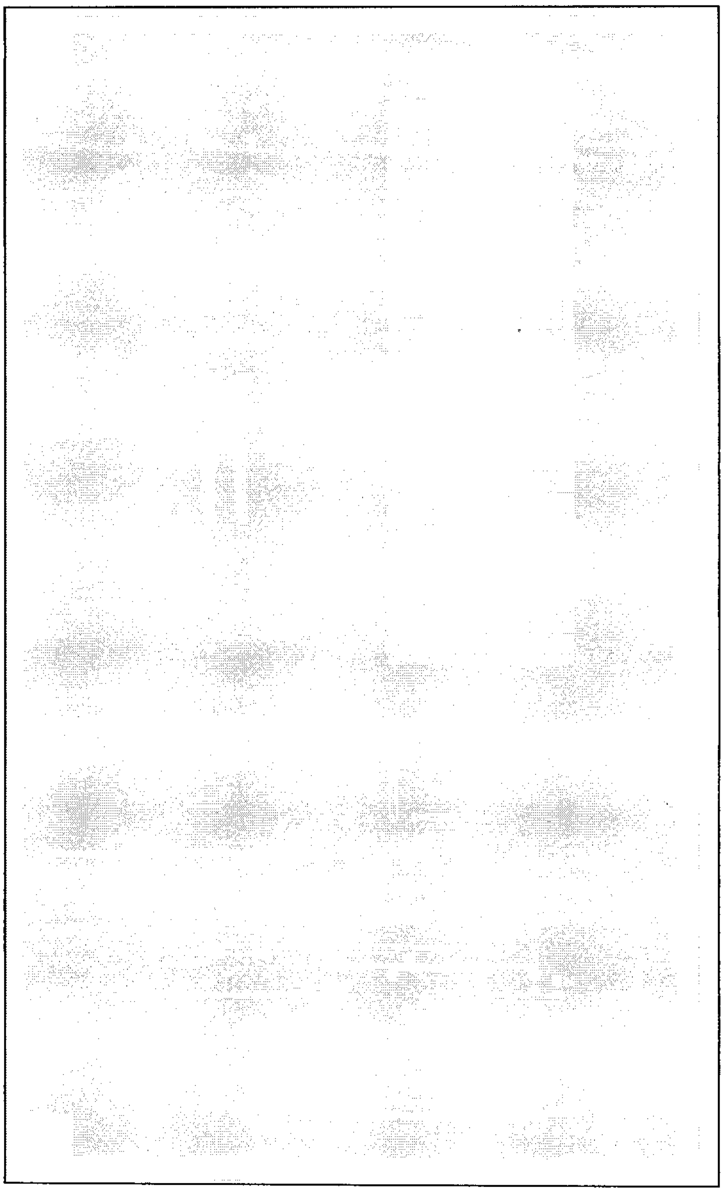


Figure 2: Bottom Copper.

2. **WCU**

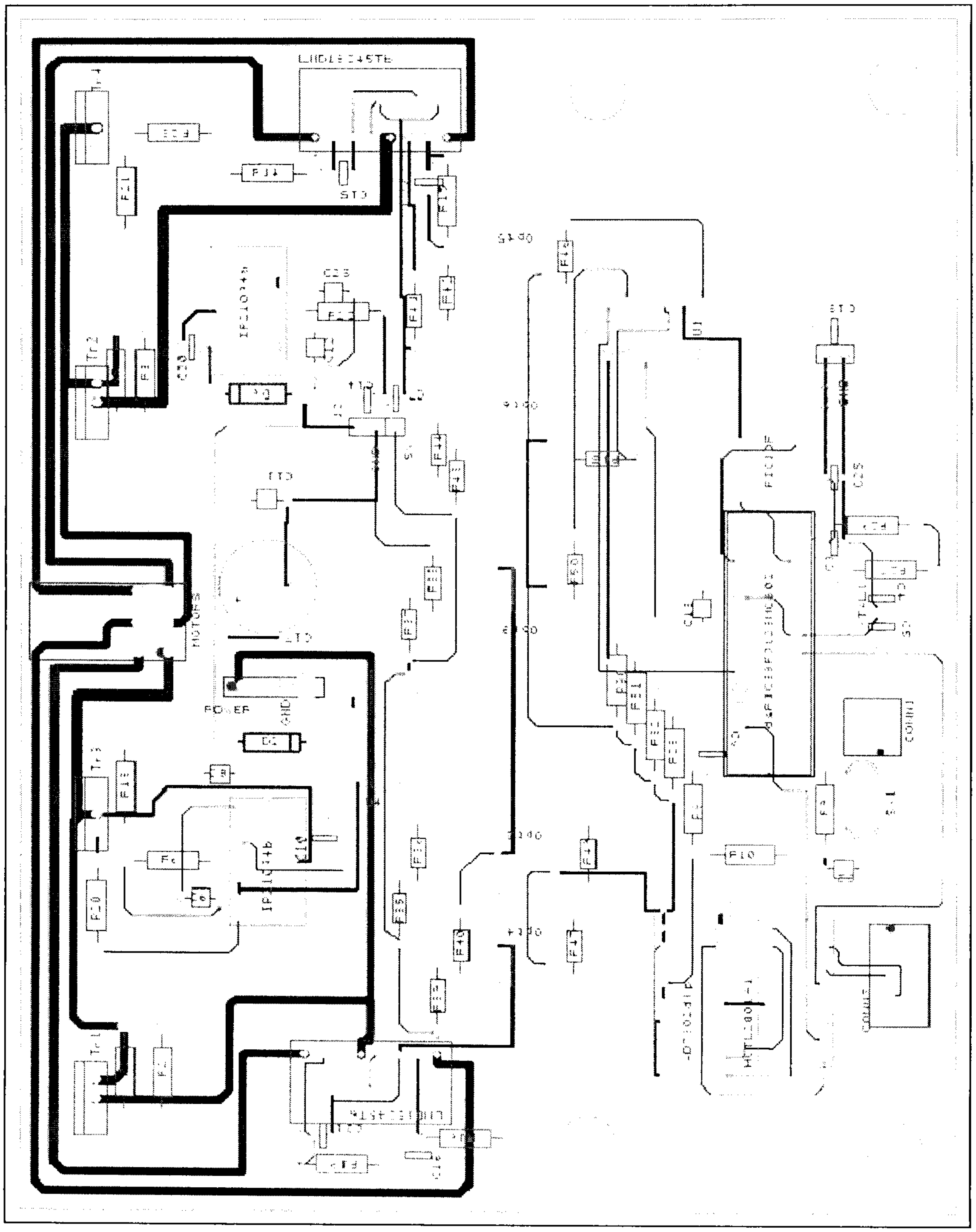


Figure 3: Top Copper.

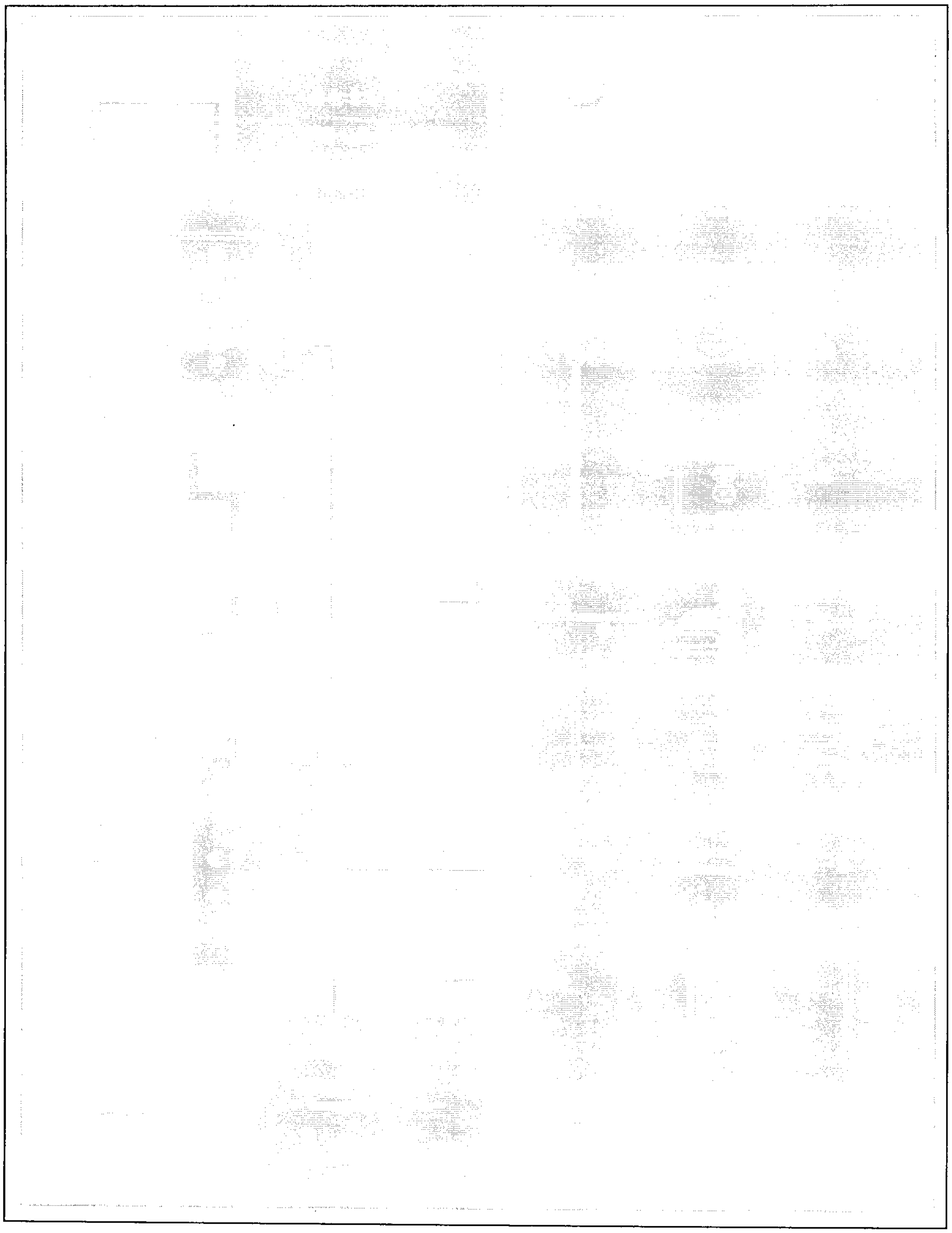


Figure 4: Bottom Copper.

3. Target Board

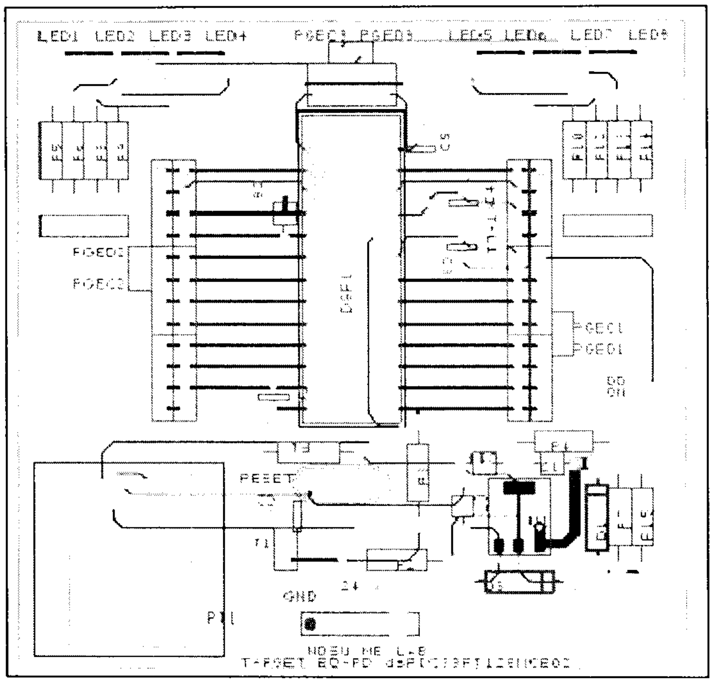


Figure 5: Top Copper.

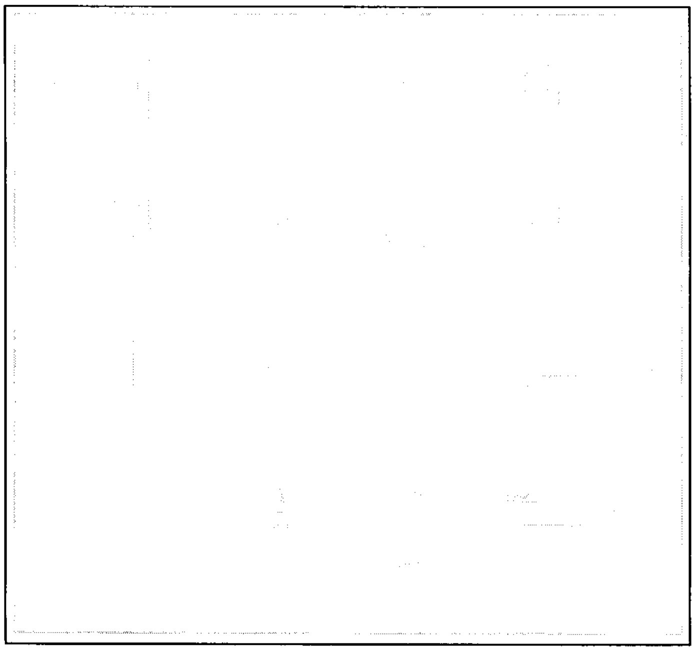


Figure 6: Bottom Copper.

4. Power Module

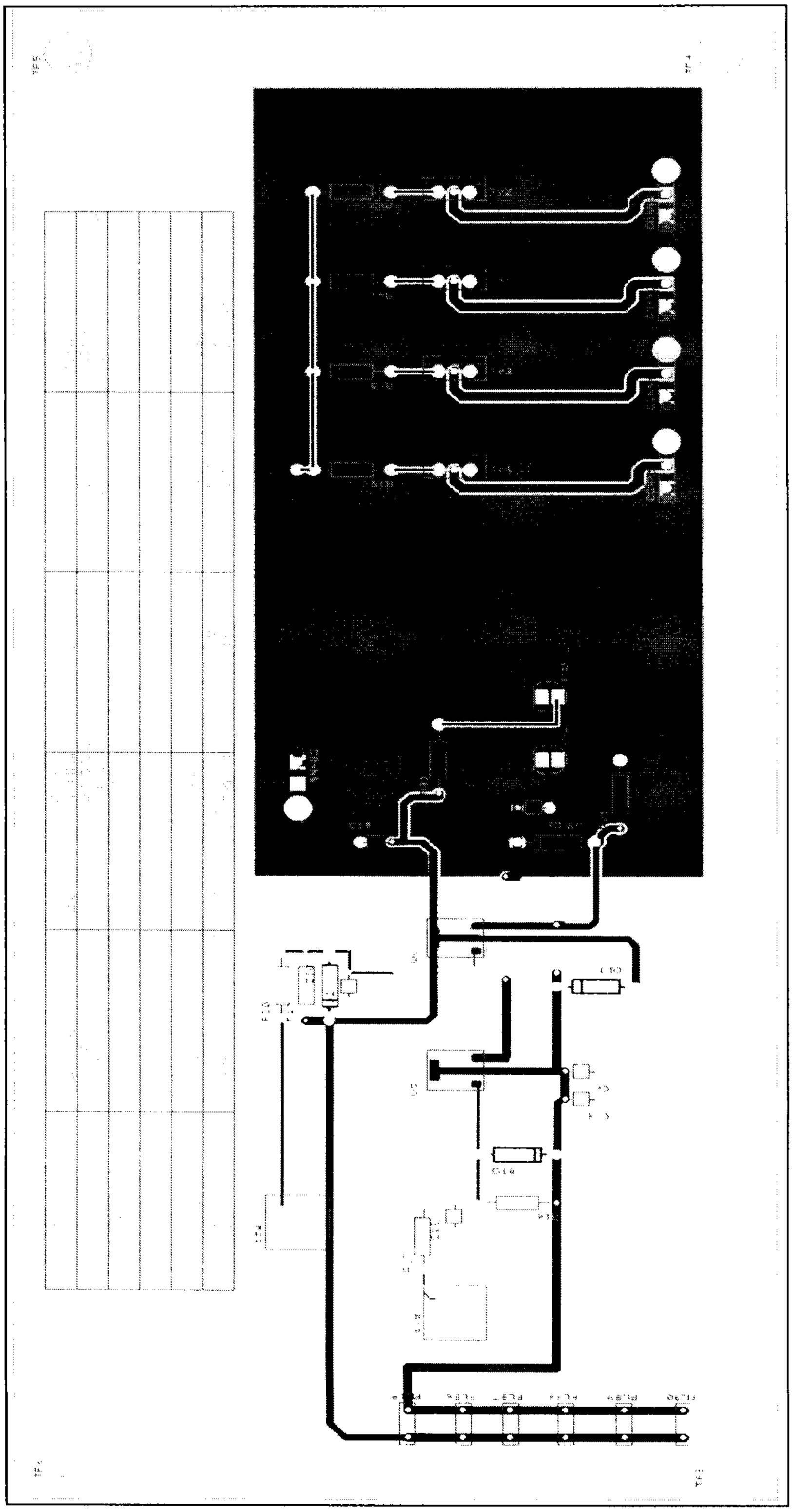


Figure 7: Top Copper.

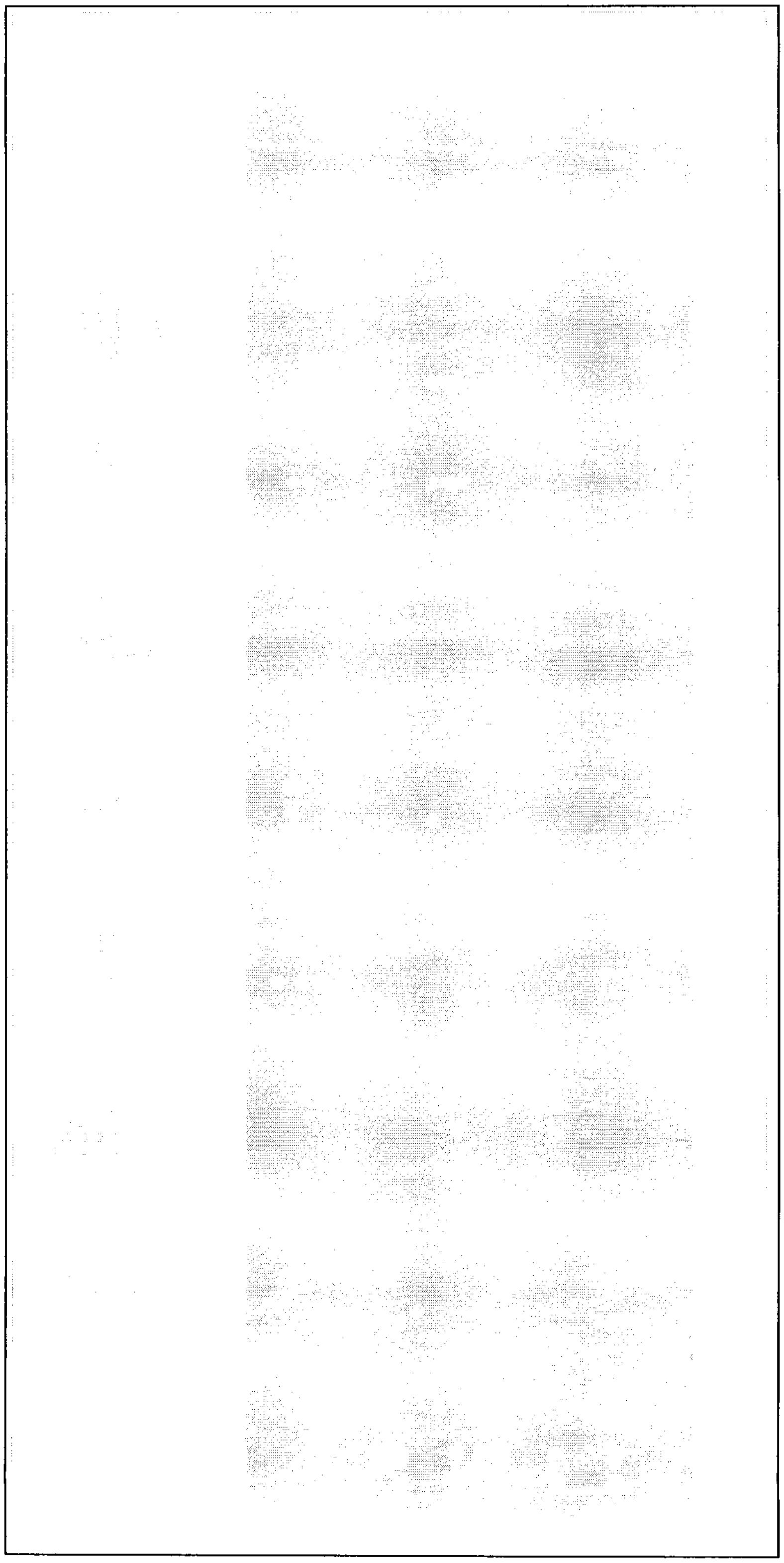


Figure 8: Bottom Copper

APPENDIX C

Firmware Codes

1. WCU

a. PWM setup

File Name: pwmDrv1.c

```
*************************************************************************

#include "p33FJ128MC802.h"
#include "pwmDrv1.h"
void initPwm1(void)
{
      P1TCONbits.PTMOD = 0b00;              // Free running mode
      P1TCONbits.PTCKPS= 0b00;              // input clock period= 1 Tcy
      P1TCONbits.PTOPS= 0b00;               // Output post scale is 1:1
      P1TCONbits.PTSIDL= 0;                 // Runs in idle mode
      P1TPER= 999;                          // Pwm frequency of 40 Khz
      PWM1CON1bits.PMOD2 = 0;         // Pwm1 pair 2 is in independent mode
      PWM1CON1 = 0;
      PWM1CON1bits.PEN2H = 1;         // Pwm1H2 pin is set active
      PWM1CON2bits.IUE = 1;                 // Immediate updates for PWM
      P1DTCON1bits.DTAPS = 0b00;            // Dead time for unit A is Tcy
      P1DTCON1bits.DTBPS = 0b00;            // Dead time for unit B is Tcy
      P1DTCON1bits.DTA = 0;                 // Dead time for unit A is '0'
      P1DTCON1bits.DTB = 20;                // Dead time for unit B is '0'

      P1DTCON2bits.DTS1A = 0;
      // Dead time for the PWM1L1 active signal is provided by unit A
      P1DTCON2bits.DTS1I = 0;
      // Dead time for the PWM1L1 inactive signal is provided by unit A
      P1DTCON2bits.DTS2A = 1;
      // Dead time for the PWM1H2 active signal is provided by unit B
      P1DTCON2bits.DTS2I = 0;
      // Dead time for the PWM1H2 inactive signal is provided by unit A
      P1OVDCONbits.POVD1L = 1;
      // Output on  PWM1L1 is controlled by the PWM generator
      P1OVDCONbits.POVD2H = 1;
      // Output on  PWM1H2 is controlled by the PWM generator
      P1DC2 = 1023;                         // Duty cycle of PWM1H2
      PWM2CON1 = 0;
      P1TCONbits.PTEN= 1;                   // PWM  module is ON
}
```

b. Analog to Digital Conversion (ADC) setup

File Name: adcDrv1.c

```
*************************************************************************

#include "p33FJ128MC802.h"
#include "adcDrv1.h"
#include "tglPin.h"

#define  SAMP_BUFF_SIZE 2  // Size of the input buffer per analog input
#define  NUM_CHS2SCAN   2  // Number of channels enabled for channel scan

void initAdc1(void)
{
      AD1CON1bits.FORM   = 0;
      // Data Output Format: Integer
      AD1CON1bits.SSRC   = 2;
      // Sample Clock Source: GP Timer starts conversion
      AD1CON1bits.ASAM   = 1;
      // ADC Sample Control: Sampling begins immediately after conversion
      AD1CON1bits.AD12B  = 0;
      // 10-bit ADC operation
      AD1CON2bits.CSCNA = 1;
      // Scan Input Selections for CH0+ during Sample A bit
      AD1CON2bits.CHPS  = 0;
      // Converts CH0
      AD1CON3bits.ADRC = 0;
      // ADC Clock is derived from Systems Clock
      AD1CON3bits.ADCS = 63;
      //  ADC  Conversion  Clock  Tad=Tcy*(ADCS+1)=  (1/40M)*64  =  1.6us
      (625Khz)
      // ADC Conversion Time for 10-bit Tc=12*Tad = 19.2us
      AD1CON2bits.SMPI    = (NUM_CHS2SCAN-1);
      // 2 ADC Channel is scanned
      //AD1CSSH/AD1CSSL: A/D Input Scan Selection Register
      AD1CSSLbits.CSS0=1;                // Enable AN0 for channel scan
      AD1CSSLbits.CSS1=1;                // Enable AN1 for channel scan
      //AD1PCFGH/AD1PCFGL: Port Configuration Register
      AD1PCFGL=0xFFFF;
      AD1PCFGLbits.PCFG0 = 0;      // AN0 as Analog Input
      AD1PCFGLbits.PCFG1 = 0;      // AN1 as Analog Input

       IFS0bits.AD1IF = 0;         // Clear the A/D interrupt flag bit
       IEC0bits.AD1IE = 1;               // Enable A/D interrupt
       AD1CON1bits.ADON = 1;       // Turn on the A/D converter
      tglPinInit();
}
/*=============================================================
Timer 3 is setup to time-out every 125 microseconds (8Khz Rate). As a
result, the module will stop sampling and trigger a conversion on every
Timer3 time-out, i.e., Ts=125us.
=============================================================*/
void initTmr3()
```

```
{
        TMR3 = 0x0000;
        PR3 = 4999;
        IFS0bits.T3IF = 0;
        IEC0bits.T3IE = 0;
        //Start Timer 3
        T3CONbits.TON = 1;
}
/*=================================================================
ADC INTERRUPT SERVICE ROUTINE
=================================================================*/
int  ain0Buff[SAMP_BUFF_SIZE];
int  ain1Buff[SAMP_BUFF_SIZE];
int  scanCounter=0;
int  sampleCounter=0;

void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void)
{
      switch (scanCounter)
      {
            case 0:
                  ain0Buff[sampleCounter]=ADC1BUF0;
                  break;
            case 1:
                  ain1Buff[sampleCounter]=ADC1BUF0;
                  break;
            default:
                  break;
      }
      scanCounter++;
      if(scanCounter==NUM_CHS2SCAN)
      {
            scanCounter=0;
            sampleCounter++;
      }
      if(sampleCounter==SAMP_BUFF_SIZE)
            sampleCounter=0;

      tglPin();                         // Toggle RA6
    IFS0bits.AD1IF = 0;                 // Clear the ADC1 Interrupt Flag
}
```

c. Remappable Pin Configuration

File Name: rpInit.c

```
***************************************************************************

#include "p33FJ128MC802.h"
#include "rpInit.h"
void initRP(void)
{
//REMAPPABLE PINS CONFIGURATION
//Unlock the registers
__builtin_write_OSCCONL(OSCCON & ~(1<<6));

//Configure SPI1 Port for SLAVE mode
AD1PCFGL=0x00FF;
//analogue pins configured as digital IO
RPINR20=0x0002;
//SCK1 Input is associated to pin RP00 and SDI1 input is associated to
pin RP02 (pin 5 of the dsPIC)
RPINR21=0x0003;
// SS1 IS ASSOCIATED WITH RP03
//Lock the registers
__builtin_write_OSCCONL(OSCCON | (1<<6));
}
```

d. Serial Peripheral Interface (SPI) setup

File Name: spiDrv1.c

****

```
#include "p33FJ128MC802.h"
void initSPI(void)
{
      //SPI1 configuration
      IFS0bits.SPI1IF=0;    //make sure the SPI interrupt flag is cleared
      IEC0bits.SPI1IE=0;    // Interrupt disabled
      SPI1CON1 = 0x051A;
      /*configure SPI1 module as Slave, CKP=0, CKE=1, SMP=1, first 4:1
and secondary prescaler are set to 2:1 therefore SCLK frequency is 5MHz,
idle state for clock is low, SCKx and SDOx controlled by the SPI module*/

      SPI1CON1bits.SSEN = 1;          // SS1 PIN INPUT ENABLED
      SPI1CON1bits.DISSDO = 1;        // DIABLING SDO1 PIN
      SPI1STATbits.SPIROV = 0;
      //make sure the overflow flag is cleared
      SPI1STAT = 0x8000;                    //enable SPI1 module
      IFS0bits.SPI1IF=0;
      //make sure the SPI interrupt flag is cleared
      IEC0bits.SPI1IE=1;                    // Interrupt enabled
}
```

e. Input/ Output Pins setup

File Name: ioInit.c

```
************************************************************************

#include "p33FJ128MC802.h"
#include "ioInit.h"
void initIO(void)
{
      // SETTING TRISx REGISTER FOR OUTPUT
      TRISAbits.TRISA4=0;
      TRISBbits.TRISB4=0;
      TRISBbits.TRISB5=0;
      TRISBbits.TRISB6=0;
      TRISBbits.TRISB7=0;
      TRISBbits.TRISB8=0;
      TRISBbits.TRISB9=0;
      TRISBbits.TRISB10=0;
      TRISBbits.TRISB11=0;
      TRISBbits.TRISB13=0;
      TRISBbits.TRISB14=0;

      //SETTING TRISx REIGISTERS FOR INPUT
      TRISBbits.TRISB0=1;  //pin RB0/RP0 is configured as input for clk
      TRISBbits.TRISB2=1;  //pin RB2/RP2 is configured as input for data
(MOSI)
      TRISBbits.TRISB1=1;  // pin RB1/RP1 is configured as an ON/OFF
      TRISBbits.TRISB3=1;      // SS1 input


      // Initializing the Port Values
      PORTAbits.RA4=0;
      PORTBbits.RB3=0;
      PORTBbits.RB4=0;
      PORTBbits.RB5=0;
      PORTBbits.RB6=0;
      PORTBbits.RB7=0;
      PORTBbits.RB8=0;
      PORTBbits.RB9=0;
      PORTBbits.RB10=0;
      PORTBbits.RB13=0;
      PORTBbits.RB14=0;

}
```

f. Module: Front Left

File Name: main_FR01.c

```
************************************************************************

#include "p33FJ128MC802.h"
#include "math.h"
#include "delay.h"
#include "adcDrv1.h"
#include "ioInit.h"
#include "pwmDrv1.h"
#include "spiDrv1.h"
#include "rpInit.h"

_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_HS);
_FWDT(FWDTEN_OFF);
_FPOR(PWMPIN_OFF & HPOL_ON & LPOL_ON)

int pot_reading=0;              // for potentiometer reading
int steering_angle=0;           // desired steering angle
int speed=0;                    // desired speed
int mode=0;                     // desired mode
int encoder=0;                  // for the speed encoder
int dir=0;                      // steering direction
int read=0;                     // information from the central controller
int a=26;                       // width of the base
int b=23;                       // length of the base
signed int var1=0;
// for the difference in the desired and the actuall steering angle

void __attribute__((interrupt, no_auto_psv)) _SPI1Interrupt(void)
// interrupt setup for the Serial communication
{
      read = SPI1BUF;
      Update_Values(read);
      IFS0bits.SPI1IF = 0;         // Clear the SPI1 Interrupt Flag
}
void Direction(unsigned int x1, unsigned int y1 )
// To determine the PWM duty for the desired speed in the desired
direction, where x1 = speed,
y1 = encoder value
{
      switch ( x1 )
      {
            case 0x0000:
                  PORTBbits.RB9 = 0;
                  P1DC2 = 0;
            break;
            case 0x0400:
                  P1DC2 = 375;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0800:
                  P1DC2 = 580;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0C00:
                  PORTBbits.RB9 = 1;
                  P1DC2 = 790;
            break;
            case 0x1000:
                  P1DC2 = 1325 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x1400:
                  P1DC2 = 1460 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x1800:
                  P1DC2 = 1660 ;
                  PORTBbits.RB9 = 1;
            break;
            default:
                  PORTBbits.RB9 = 0;
                  P1DC2 = 0;
            break;
      }
}
void Do_Steering (unsigned int x, unsigned int y)
// To do the desired steering, where x = pot_reading, y = required
steering angle
{
      PORTBbits.RB6 = 1;
      var1 = (x - y);
      if (var1 <= -5)
      {
            PORTBbits.RB6 = 0;      //PORTBbits.RB6 = Break
            PORTBbits.RB8 = 1;
            //PORTBbits.RB8 = STEERING SPEED CONTROL
            PORTAbits.RA4 = 0;      //PORTAbits.RA4 = Left/Right

      }
      else if ( var1 >= 5 )
      {
            PORTBbits.RB6 = 0;
            PORTAbits.RA4 = 1;
            PORTBbits.RB8 = 0;
      }
      else
      {
            PORTBbits.RB6 = 1;
      }
}
void Steering1( unsigned int x2)
// For the straight run , where X2= pot_reading
{
      Do_Steering(x2,512);
}
void Steering2(unsigned int x3, unsigned int y3, unsigned int z3)
```

```
// For Front and all wheel steering where, x3= steering_angle, y3=
pot_reading, z3= direction         (front or rear wheel)
{
      if(z3==1)
      {
      if ( mode == 0x4000)
      {
x3=(512-((1.5708-(atan((a/b)+(1/tan((512-x3)*0.004602)))))*217.299));
      }
            else if ( mode == 0x2000)
      {
x3=(512-((1.5708-(atan((a/(2*b))+(1/tan((512-x3)*0.004602)))))*217.299));
      }
      }
      else if(z3==0)
      {
            if (mode==0x4000)
            {
x3=(((1.5708-(atan((1/tan((x3-512)*0.004602))-(a/b))))*217.299)+512);
            }
            else if (mode==0x2000)
            {
x3=(((1.5708-(atan((1/tan((x3-512)*0.004602))-(a/(2*b)))))*217.299)+512);
            }
      }
      Do_Steering(y3,x3);
}
void Steering3( unsigned int x4, unsigned int y4)
//For parallel steering, where x4= steering_angle,  y4= pot_reading
{
      Do_Steering(y4,x4);
}
void Steering4 (unsigned int y5)
// For Zero radius turn steering where x5= steering_angle, y5=
pot_reading
{
      unsigned int x5;
      x5 = (512 - (0.72425*217.3));
      Do_Steering(y5, x5);
}
void Update_Values(unsigned int x5)
// Update the variables such as speed, mode and steering angle from the
data obtained from the data bus
{
      mode                = ( x5 & 0xE000 );
      steering_angle           = ( x5 & 0x03FF );
      speed                    = ( x5 & 0x1C00 );
}
void Wheel_Flip (unsigned int x6)
// Call the required steering function depending on the direction of
movement and the mode selected, where x6 = speed
{
      if ( x6 >= 0 && x6 <= 3072)
      {
            switch ( mode )
            {
                  case 0x0000:
                        Steering1( pot_reading );
                        // straight run steering called
                break;
                case 0x2000:
                        Steering2 ( steering_angle, pot_reading, dir );
                        // Front wheel steering called
                break;
                case 0x4000:
                        Steering2 ( steering_angle, pot_reading, dir );
                        // All wheel steering called
                break;
                case 0x8000:
                        Steering4(pot_reading);
                        // Zero radius turn steering called
                break;
                case 0xC000:
                        Steering3 ( steering_angle, pot_reading );
                        // Parallel steering called
                break;
                default:
                        Steering1( pot_reading );
                        // Default steering i.e., straight run
                break;
        }
}
else if ( x6 >= 4096 && x6 <= 6144)
{
        switch ( mode )
        {
                case 0x0000:
                        Steering1( pot_reading );
                break;
                case 0x2000:
                        Steering1( pot_reading );
                break;
                case 0x4000:
                        Steering2 ( steering_angle, pot_reading, dir );
                break;
                case 0x8000:
                        Steering4(pot_reading);
                break;
                case 0xC000:
                        Steering3 ( steering_angle, pot_reading );
                break;
                default:
                        Steering1( pot_reading );
                break;
        }
}
else if ( x6 > 3072 && x6 < 4096)    // default steering called
{
        switch ( mode )
        {
                case 0x0000:
                        Steering1( pot_reading );
                break;
                case 0x2000:
                        Steering1( pot_reading );
                  break;
                  case 0x4000:
                        Steering1( pot_reading );
                  break;
                  case 0x8000:
                        Steering1( pot_reading );
                  break;
                  case 0xC000:
                        Steering1( pot_reading );
                  break;
                  default:
                        Steering1( pot_reading );
                  break;
            }
      }
}
void Steering_Direction(int x7)
// To determine the steering direction i.e., either left or right where,
x7 = steering_angle
{
      if (steering_angle >= 0 && steering_angle <= 512)
      {
            dir = 1;
      }
      else
      {
            dir=0;
      }
}
int main (void)                                   // Main function
{
      int temp;                 // temporary variable to read the SP1BUF
/* Configure Oscillator to operate the device at 40Mhz
   Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
   Fosc= 20M*40/(2*4)=80Mhz for 8M input clock */
      PLLFBD=30;                      // M=32
      CLKDIVbits.PLLPOST=0;           // N1=2
      CLKDIVbits.PLLPRE=2;            // N2=4
// clock switch to incorporate PLL
      __builtin_write_OSCCONH(0x03); // Initiate Clock Switch to Primary
                                      // Oscillator with PLL (NOSC=0b011)
      __builtin_write_OSCCONL(0x01);       // Start clock switching
      while (OSCCONbits.COSC != 0b011); // Wait for Clock switch to occur
                                           // Wait for PLL to lock
      while(OSCCONbits.LOCK!=1) {};

// Peripheral Initialisation
      initRP();                 //Initialize Remappable Peripheral Pins
      initAdc1();               //Initialize ADC module
      initTmr3();               //Initialize TIMER 3 for ADC conversion
      initPwm1();               // Initialize PWM module
      initIO();                 // Initialize Input/output pins
      initSPI();                // Initialize SPI module
      read = 0;
      temp = SPI1BUF;
      while(1)
```

```
	{
		pot_reading		= ( ain0Buff[0]+ain0Buff[1]) / 2;
		encoder		= ( ain1Buff[0]+ain1Buff[1]) / 2;
		Direction ( speed, encoder );
		Steering_Direction(steering_angle);
		Wheel_Flip(speed);
		if(PORTBbits.RB3==1)
		{
			temp=SPI1BUF;
			SPI1STAT = 0x0000;		//disable SPI1 module
			IFS0bits.SPI1IF=0;
			//make sure the SPI interrupt flag is cleared
			IEC0bits.SPI1IE=0;		// Interrupt disabled
			SPI1STAT=0x8000;
			SPI1STATbits.SPIROV=0;
			IFS0bits.SPI1IF=0;
			//make sure the SPI interrupt flag is cleared
			IEC0bits.SPI1IE=1;		// Interrupt enabled
		}
	}
	return (0);
}
```

g. Module: Front Right

File Name: main_FR02.c

```
**************************************************************************

#include "p33FJ128MC802.h"
#include "math.h"
#include "delay.h"
#include "adcDrv1.h"
#include "ioInit.h"
#include "pwmDrv1.h"
#include "spiDrv1.h"
#include "rpInit.h"

_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_HS);
_FWDT(FWDTEN_OFF);
_FPOR(PWMPIN_OFF & HPOL_ON & LPOL_ON)

int pot_reading=0;              // for potentiometer reading
int steering_angle=0;           // desired steering angle
int speed=0;                    // desired speed
int mode=0;                     // desired mode
int encoder=0;                  // for the speed encoder
int dir=0;                      // steering direction
int read=0;                     // information from the central controller
int a=26;                       // width of the base
int b=23;                       // length of the base
signed int var1=0;
// for the difference in the desired and the actual steering angle
void __attribute__((interrupt, no_auto_psv)) _SPI1Interrupt(void)
// interrupt setup for the Serial Communication
{
      read = SPI1BUF;
      Update_Values(read);
      IFS0bits.SPI1IF = 0;          // Clear the SPI1 Interrupt Flag
}
void Direction(unsigned int x1, unsigned int y1 )
// To determine the PWM duty for the desired speed in the desired
direction, where x1 = speed,
y1 = encoder value
{
      if ( mode == 0x8000)
      {
            switch ( x1 )
            {
                  case 0x0000:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                  break;
                  case 0x1800:
                        P1DC2 = 130;
                        PORTBbits.RB9 = 1;
                break;
                case 0x1400:
                        P1DC2 = 450;
                        PORTBbits.RB9 = 1;
                break;
                case 0x1000:
                        PORTBbits.RB9 = 1;
                        P1DC2 = 720;
                break;
                case 0x0C00:
                        P1DC2 = 1260 ;
                        PORTBbits.RB9 = 1;
                break;
                case 0x0800:
                        P1DC2 = 1390 ;
                        PORTBbits.RB9 = 1;
                break;
                case 0x0400:
                        P1DC2 = 1700 ;
                        PORTBbits.RB9 = 1;
                break;
                default:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                break;
        }
}
else
{
        switch ( x1 )
        {
                case 0x0000:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                break;
                case 0x0400:
                        P1DC2 = 130;
                        PORTBbits.RB9 = 1;
                break;
                case 0x0800:
                        P1DC2 = 450;
                        PORTBbits.RB9 = 1;
                break;
                case 0x0C00:
                        PORTBbits.RB9 = 1;
                        P1DC2 = 720;
                break;
                case 0x1000:
                        P1DC2 = 1260 ;
                        PORTBbits.RB9 = 1;
                break;
                case 0x1400:
                        P1DC2 = 1390 ;
                        PORTBbits.RB9 = 1;
                break;
                case 0x1800:
                        P1DC2 = 1700 ;
                        PORTBbits.RB9 = 1;
                break;
                default:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                break;
        }
    }
}
void Do_Steering (unsigned int x, unsigned int y)
// To do the desired steering, where x = pot_reading, y = required
steering angle
{
        PORTBbits.RB6 = 1;
        var1 = (x - y);
        if (var1 <= -5)
        {
                PORTBbits.RB8 = 0;      //PORTBbits.RB6 = Break
                PORTBbits.RB6 = 0;
                //PORTBbits.RB8 = STEERING SPEED CONTROL
                PORTAbits.RA4 = 0;      //PORTBbits.RA4 = Left/Right
        }
        else if ( var1 >= 5 )
        {
                PORTBbits.RB8 = 1;
                PORTAbits.RA4 = 1;
                PORTBbits.RB6 = 0;
        }
        else
        {
                PORTBbits.RB6 = 1;
        }
}
void Steering1( unsigned int x2)
// For the straight run , where X2= pot_reading
{
        Do_Steering(x2,512);
}
void Steering2(unsigned int x3, unsigned int y3, unsigned int z3)
// For Front and all wheel steering where, x3= steering_angle, y3=
pot_reading, z3= direction ( front or rear wheel)
{
        if(z3==1)
        {
                if ( mode == 0x4000)
                {
x3 = (((1.5708-(atan((a/b) + (1/tan((x3-512)*0.004602)))))*217.299)+512);
                }
                else if ( mode == 0x2000)
                {
x3=(((1.5708-(atan((a/(2*b))+(1/tan((x3-512)*0.004602)))))*217.299)+512);
                }
        }
        else if(z3==0)
        {
                if (mode==0x4000)
```

```
            {
      x3=512-((1.5708-(atan((1/tan((512-x3)*0.004602))-(a/b))))*217.299);
            }
            else if (mode==0x2000)
            {
x3=512-((1.5708-(atan((1/tan((512-x3)*0.004602))-(a/(2*b)))))*217.299);
            }
      }
      Do_Steering(y3,x3);
}
void Steering3( unsigned int x4, unsigned int y4)
//For parallel steering, where x4= steering_angle,  y4= pot_reading
{
      Do_Steering(y4,x4);
}
void Steering4 (unsigned int y5)
//   For   Zero   radius   turn   steering   where   x5=   steering_angle,   y5=
pot_reading
{
      unsigned int x5;
      x5 = (512 + (0.72425*217.3));
      Do_Steering(y5, x5);
}
void Update_Values(unsigned int x5)
// Update the variables such as speed, mode and steering angle from the
data obtained from the data bus
{
      mode                = ( x5 & 0xE000 );
      steering_angle      = ( x5 & 0x03FF );
      speed               = (x5 & 0x1C00);
}
void Wheel_Flip (unsigned int x6)
// Call  the  required  steering  function  depending  on  the  direction  of
movement and the mode selected, where x6 = speed
{
      if ( x6 >= 0 && x6 <= 3072)
      {
            switch ( mode )
            {
                  case 0x0000:
                        Steering1( pot_reading );
                        // straight run steering called
                  break;
                  case 0x2000:
                        Steering2 ( steering_angle, pot_reading, dir );
                        // Front wheel steering called
                  break;
                  case 0x4000:
                        Steering2 ( steering_angle, pot_reading, dir );
                        // All wheel steering called
                  break;
                  case 0x8000:
                        Steering4(pot_reading);
                        // Zero radius turn steering called
                  break;
                  case 0xC000:
                    Steering3 ( steering_angle, pot_reading );
                    // Parallel steering called
                break;
                default:
                    Steering1( pot_reading );
                    // Default steering i.e., straight run
                break;
        }
}
else if ( x6 >= 4096 && x6 <= 6144)
{
        switch ( mode )
        {
                case 0x0000:
                    Steering1( pot_reading );
                break;
                case 0x2000:
                    Steering1( pot_reading );
                break;
                case 0x4000:
                    Steering2 ( steering_angle, pot_reading, dir );
                break;
                case 0x8000:
                    Steering4(pot_reading);
                break;
                case 0xC000:
                    Steering3 ( steering_angle, pot_reading );
                break;
                default:
                    Steering1( pot_reading );
                break;
        }
}
else if ( x6 > 3072 && x6 < 4096)    // default steering called
{
        switch ( mode )
        {
                case 0x0000:
                    Steering1( pot_reading );
                break;
                case 0x2000:
                    Steering1( pot_reading );
                break;
                case 0x4000:
                    Steering1( pot_reading );
                break;
                case 0x8000:
                    Steering1(pot_reading);
                break;
                case 0xC000:
                    Steering1( pot_reading );
                break;
                default:
                    Steering1( pot_reading );
                break;
        }
}
```

```
}
void Steering_Direction(int x7)
// To determine the steering direction i.e., either left or right where,
x7 = steering_angle
{
      if (steering_angle >= 0 && steering_angle <= 512)
      {
            dir =0;
      }
      else
      {
            dir=1;
      }
}
int main (void)                                // Main function
{
      int temp;               // temporary variable to read the SPIBUF

/* Configure Oscillator to operate the device at 40Mhz
   Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
   Fosc= 20M*40/(2*4)=80Mhz for 8M input clock */
      PLLFBD=30;                    // M=32
      CLKDIVbits.PLLPOST=0;         // N1=2
      CLKDIVbits.PLLPRE=2;          // N2=4
// clock switch to incorporate PLL
      __builtin_write_OSCCONH(0x03); // Initiate Clock Switch to Primary
                                    // Oscillator with PLL (NOSC=0b011)
      __builtin_write_OSCCONL(0x01);            // Start clock switching
      while (OSCCONbits.COSC != 0b011); // Wait for Clock switch to occur
                                       // Wait for PLL to lock
      while(OSCCONbits.LOCK!=1) {};

// Peripheral Initialisation
      initRP();               // Initialize Remappable Peripheral Pins
      initAdc1();             // Initialize ADC module
      initTmr3();             // Initialize TIMER 3 for ADC conversion
      initPwm1();             // Initialize PWM module
      initIO();               // Initialize Input/output pins
      initSPI();              // Initialize SPI module

      read = 0;
      temp = SPI1BUF;
      while(1)
      {
            pot_reading       = ( ain0Buff[0]+ain0Buff[1]) / 2;
            encoder           = ( ain1Buff[0]+ain1Buff[1]) / 2;
            Direction ( speed, encoder );
            Steering_Direction(steering_angle);
            Wheel_Flip(speed);
            if(PORTBbits.RB3==1)
            {
                  temp=SPI1BUF;
                  SPI1STAT = 0x0000;            //disable SPI1 module
                  IFS0bits.SPI1IF=0;
                  //make sure the SPI interrupt flag is cleared
                  IEC0bits.SPI1IE=0;            // Interrupt disabled
                  SPI1STAT=0x8000;
```

```
                SPI1STATbits.SPIROV=0;
                IFS0bits.SPI1IF=0;
                //make sure the SPI interrupt flag is cleared
                IEC0bits.SPI1IE=1;            // Interrupt enabled
            }
        }
        return (0);
}
```

h. Module: Rear Left

File Name: main_FR03.c

```
****************************************************************************

#include "p33FJ128MC802.h"
#include "math.h"
#include "delay.h"
#include "adcDrv1.h"
#include "ioInit.h"
#include "pwmDrv1.h"
#include "spiDrv1.h"
#include "rpInit.h"

_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_HS);
_FWDT(FWDTEN_OFF);
_FPOR(PWMPIN_OFF & HPOL_ON & LPOL_ON)

int pot_reading=0;              // for potentiometer reading
int steering_angle=0;           // desired steering angle
int speed=0;                    // desired speed
int mode=0;                     // desired mode
int encoder=0;                  // for the speed encoder
int dir=0;                      // steering direction
int read=0;                     // information from the central controller
int a=26;                       // width of the base
int b=23;                       // length of the base
signed int var1=0;
// for the difference in the desired and the actual steering angle
void __attribute__((interrupt, no_auto_psv)) _SPI1Interrupt(void)
// interrupt setup for the Serial communication
{
      read = SPI1BUF;
      Update_Values(read);
      IFS0bits.SPI1IF = 0;          // Clear the SPI1 Interrupt Flag
}
void Direction(unsigned int x1, unsigned int y1 )
// To determine the PWM duty for the desired speed in the desired
direction, where x1 = speed, y1 = encoder value
{
      switch ( x1 )
      {
            case 0x0000:
                  PORTBbits.RB9 = 0;
                  P1DC2 = 0;
            break;
            case 0x0400:
                  P1DC2 = 320;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0800:
                P1DC2 = 530;
                PORTBbits.RB9 = 1;
          break;
          case 0x0C00:
                PORTBbits.RB9 = 1;
                P1DC2 = 645;
          break;
          case 0x1000:
                P1DC2 = 1200 ;
                PORTBbits.RB9 = 1;
          break;
          case 0x1400:
                P1DC2 = 1420 ;
                PORTBbits.RB9 = 1;
          break;
          case 0x1800:
                P1DC2 = 1610 ;
                PORTBbits.RB9 = 1;
          break;
          default:
                PORTBbits.RB9 = 0;
                P1DC2 = 0;
          break;
    }
}
void Do_Steering (unsigned int x, unsigned int y)
// To do the desired steering, where x = pot_reading, y = required
steering angle
{
    PORTBbits.RB6 = 1;
    var1 = (x - y);
    if (var1 <= -5)
    {
          PORTBbits.RB8 = 1;          // PORTBbits.RB6 = Break
          PORTBbits.RB6 = 0; // PORTBbits.RB8 = Steering Speed Control
          PORTAbits.RA4 = 0;          // PORTAbits.RA4 = Left/Right

    }
    else if ( var1 >= 5 )
    {
          PORTBbits.RB8 = 0;
          PORTAbits.RA4 = 1;
          PORTBbits.RB6 = 0;
    }
    else
    {
          PORTBbits.RB6 = 1;
    }
}
void Steering1( unsigned int x2)
// For the straight run , where X2= pot_reading
{
    Do_Steering(x2,512);
}
void Steering2(unsigned int x3, unsigned int y3, unsigned int z3)
// For Front and all wheel steering where, x3= steering_angle, y3=
pot_reading, z3= direction ( front or rear wheel)
```

```
{
      if(z3==1)
      {
            if ( mode == 0x4000)
            {
x3 = 512-((1.5708-(atan((1/tan((x3-512)*0.004602))-(a/b))))*217.299);
            }
            else if ( mode == 0x2000)
            {
x3=((1.5708-(atan((a/(2*b)) + (1/tan((x3-512)*0.004602)))))*217.299)+512;
            }
      }
      else if(z3==0)
      {
            if (mode==0x4000)
            {
x3=512+((1.5708-(atan((a/b)+(1/tan((512-x3)*0.004602)))))*217.299);
            }
            else if (mode==0x2000)
            {
x3=512-((1.5708-(atan((1/tan((512-x3)*0.004602))-(a/(2*b)))))*217.299);
            }
      }
      Do_Steering(y3,x3);
}
void Steering3( unsigned int x4, unsigned int y4)
//For parallel steering, where x4= steering_angle,  y4= pot_reading
{
      Do_Steering(y4,x4);
}
void Steering4 (unsigned int y5)
//  For  Zero  radius  turn  steering  where  x5=  steering_angle,  y5=
pot_reading
{
      unsigned int x5;
      x5 = (512 + (0.72425*217.3));
      Do_Steering(y5, x5);
}
void Update_Values(unsigned int x5)
// Update the variables such as speed, mode and steering angle from the
data obtained from the data bus
{
      mode                = ( x5 & 0xE000 );
      steering_angle      = ( x5 & 0x03FF );
      speed               = ( x5 & 0x1C00 );
}
void Wheel_Flip (unsigned int x6)
// Call the required steering function depending on the direction of
movement and the mode selected, where x6 = speed
{
      if ( x6 >= 4096 && x6 <= 6144)
      {
            switch ( mode )
            {
                  case 0x0000:
                        Steering1( pot_reading );
                        // straight run steering called
            break;
            case 0x2000:
                  Steering2 ( steering_angle, pot_reading, dir );
                  // Front wheel steering called
            break;
            case 0x4000:
                  Steering2 ( steering_angle, pot_reading, dir );
                  // All wheel steering called
            break;
            case 0x8000:
                  Steering4(pot_reading);
                  // Zero radius turn steering called
            break;
            case 0xC000:
                  Steering3 ( steering_angle, pot_reading );
                  // Parallel steering called
            break;
            default:
                  Steering1( pot_reading );
                  // Default steering i.e., straight run
            break;
      }
}
else if ( x6 >= 0 && x6 <= 3072)
{
      switch ( mode )
      {
            case 0x0000:
                  Steering1( pot_reading );
            break;
            case 0x2000:
                  Steering1( pot_reading );
            break;
            case 0x4000:
                  Steering2 ( steering_angle, pot_reading, dir );
            break;
            case 0x8000:
                  Steering4(pot_reading);
            break;
            case 0xC000:
                  Steering3 ( steering_angle, pot_reading );
            break;
            default:
                  Steering1( pot_reading );
            break;
      }
}
else if ( x6 > 3072 && x6 < 4096)   // default steering called
{
      switch ( mode )
      {
            case 0x0000:
                  Steering1( pot_reading );
            break;
            case 0x2000:
                  Steering1( pot_reading );
            break;
```

```
                case 0x4000:
                      Steering1( pot_reading );
                break;
                case 0x8000:
                      Steering1(pot_reading);
                break;
                case 0xC000:
                      Steering1( pot_reading );
                break;
                default:
                      Steering1( pot_reading );
                break;
          }
    }
}
void Steering_Direction(int x7)
// To determine the steering direction i.e., either left or right where,
x7 = steering_angle
{
      if (steering_angle >= 0 && steering_angle <= 512)
      {
            dir =0;
      }
      else
      {
            dir=1;
      }
}
int main (void)                                       // Main function
{
      int temp;                 // temporary variable to read the SP1BUF
/* Configure Oscillator to operate the device at 40Mhz
   Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
   Fosc= 20M*40/(2*4)=80Mhz for 8M input clock */
      PLLFBD=30;                      // M=32
      CLKDIVbits.PLLPOST=0;           // N1=2
      CLKDIVbits.PLLPRE=2;            // N2=4
// clock switch to incorporate PLL
      __builtin_write_OSCCONH(0x03); // Initiate Clock Switch to Primary
                                      // Oscillator with PLL (NOSC=0b011)
      __builtin_write_OSCCONL(0x01);            // Start clock switching
      while (OSCCONbits.COSC != 0b011); // Wait for Clock switch to occur
                                              // Wait for PLL to lock
      while(OSCCONbits.LOCK!=1) {};

// Peripheral Initialisation
      initRP();                 // Initialize Remappable Peripheral Pins
      initAdc1();               // Initialize ADC module
      initTmr3();               // Initialize TIMER 3 for ADC conversion
      initPwm1();               // Initialize PWM module
      initIO();                 // Initialize Input/output pins
      initSPI();                // Initialize SPI module
      read = 0;
      temp = SPI1BUF;
      while(1)
      {
            pot_reading       = ( ain0Buff[0]+ain0Buff[1]) / 2;
```

```
        encoder            = ( ain1Buff[0]+ain1Buff[1]) / 2;
        Direction ( speed, encoder );
        Steering_Direction(steering_angle);
        Wheel_Flip(speed);
        if(PORTBbits.RB3==1)
        {
              temp=SPI1BUF;
              SPI1STAT = 0x0000;             //disable SPI1 module
              IFS0bits.SPI1IF=0;
               //make sure the SPI interrupt flag is cleared
              IEC0bits.SPI1IE=0;             // Interrupt disabled
              SPI1STAT=0x8000;
              SPI1STATbits.SPIROV=0;
              IFS0bits.SPI1IF=0;
               //make sure the SPI interrupt flag is cleared
              IEC0bits.SPI1IE=1;             // Interrupt enabled
        }
  }
  return (0);
}
```

i. Module: Rear Right

File Name: main_FR04.c

****

```
#include "p33FJ128MC802.h"
#include "math.h"
#include "delay.h"
#include "adcDrv1.h"
#include "ioInit.h"
#include "pwmDrv1.h"
#include "spiDrv1.h"
#include "rpInit.h"

_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_HS);
_FWDT(FWDTEN_OFF);
_FPOR(PWMPIN_OFF & HPOL_ON & LPOL_ON)

int pot_reading=0;              // for potentiometer reading
int steering_angle=0;           // desired steering angle
int speed=0;                    // desired speed
int mode=0;                     // desired mode
int encoder=0;                  // for the speed encoder
int dir=0;                      // steering direction
int read=0;                     // information from the central controller
int a=26;                       // width of the base
int b=23;                       // length of the base
signed int var1=0;
// for the difference in the desired and the actual steering angle
void __attribute__((interrupt, no_auto_psv)) _SPI1Interrupt(void)
// interrupt setup for the Serial communication
{
      read = SPI1BUF;
      Update_Values(read);
      IFS0bits.SPI1IF = 0;          // Clear the SPI1 Interrupt Flag
}
void Direction(unsigned int x1, unsigned int y1 )
// To determine the PWM duty for the desired speed in the desired
direction, where x1 = speed, y1 = encoder value
{
      if (mode == 0x8000)
      {
            switch ( x1 )
            {
                  case 0x0000:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                  break;
                  case 0x1800:
                        P1DC2 = 285;
                        PORTBbits.RB9 = 1;
            break;
            case 0x1400:
                  P1DC2 = 530;
                  PORTBbits.RB9 = 1;
            break;
            case 0x1000:
                  PORTBbits.RB9 = 1;
                  P1DC2 = 800;
            break;
            case 0x0C00:
                  P1DC2 = 1330 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0800:
                  P1DC2 = 1420 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0400:
                  P1DC2 = 1610 ;
                  PORTBbits.RB9 = 1;
            break;
            default:
                  PORTBbits.RB9 = 0;
                  P1DC2 = 0;
            break;
      }
}
else
{
      switch ( x1 )
      {
            case 0x0000:
                  PORTBbits.RB9 = 0;
                  P1DC2 = 0;
            break;
            case 0x0400:
                  P1DC2 = 285;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0800:
                  P1DC2 = 530;
                  PORTBbits.RB9 = 1;
            break;
            case 0x0C00:
                  PORTBbits.RB9 = 1;
                  P1DC2 = 800;
            break;
            case 0x1000:
                  P1DC2 = 1330 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x1400:
                  P1DC2 = 1420 ;
                  PORTBbits.RB9 = 1;
            break;
            case 0x1800:
                  P1DC2 = 1610 ;
```

```
                        PORTBbits.RB9 = 1;
                    break;
                    default:
                        PORTBbits.RB9 = 0;
                        P1DC2 = 0;
                    break;
                }
        }
}

void Do_Steering (unsigned int x, unsigned int y)
// To do the desired steering, where x = pot_reading, y = required
steering angle
{
        PORTBbits.RB6 = 1;
        var1 = (x - y);
        if (var1 <= -5)
        {
                PORTBbits.RB6 = 0;      //PORTBbits.RB6 = Break
                PORTBbits.RB8 = 0; //PORTBbits.RB8 = STEERING SPEED CONTROL
                PORTAbits.RA4 = 0;      //PORTAbits.RA4 = Left/Right
        }
        else if ( var1 >= 5 )
        {
                PORTBbits.RB8 = 1;
                PORTAbits.RA4 = 1;
                PORTBbits.RB6 = 0;
        }
        else
        {
                PORTBbits.RB6 = 1;
        }
}
void Steering1( unsigned int x2)
// For the straight run , where X2= pot_reading
{
        Do_Steering(x2,512);
}
void Steering2(unsigned int x3, unsigned int y3, unsigned int z3)
 // For Front and all wheel steering where, x3= steering_angle, y3=
pot_reading, z3= direction ( front or rear wheel)
{
        if(z3==1)
        {
                if ( mode == 0x4000)
                {
x3 = 512+((1.5708-(atan((1/tan((512-x3)*0.004602))-(a/b))))*217.299);
                }
                else if ( mode == 0x2000)
                {
x3=512-((1.5708-(atan((a/(2*b)) + (1/tan((512-x3)*0.004602)))))*217.299);
                }
        }
        else if(z3==0)
        {
                if (mode==0x4000)
                {
```

```
x3=512-((1.5708-(atan((a/b) + (1/tan((x3-512)*0.004602)))))*217.299);
               }
               else if (mode==0x2000)
               {
x3=((1.5708-(atan((1/tan((x3-512)*0.004602))-(a/(2*b)))))*217.299)+512;
               }
        }

        Do_Steering(y3,x3);
}
void Steering3( unsigned int x4, unsigned int y4)
//For parallel steering, where x4= steering_angle,  y4= pot_reading
{
        Do_Steering(y4,x4);
}
void Steering4 (unsigned int y5)
//  For  Zero  radius  turn  steering  where  x5=  steering_angle,  y5=
pot_reading
{
        unsigned int x5;
        x5 = (512 - (0.72425*217.3));
        Do_Steering(y5, x5);
}
void Update_Values(unsigned int x5)
// Update the variables such as speed, mode and steering angle from the
data obtained from the data bus
{
        mode                = ( x5 & 0xE000 );
        steering_angle      = ( x5 & 0x03FF );
        speed               = ( x5 & 0x1C00 );
}
void Wheel_Flip (unsigned int x6)
// Call the required steering function depending on the direction of
movement and the mode selected, where x6 = speed
{
        if ( x6 >= 4096 && x6 <= 6144)
        {
               switch ( mode )
               {
                      case 0x0000:
                             Steering1( pot_reading );
                             // straight run steering called
                      break;
                      case 0x2000:
                             Steering2 ( steering_angle, pot_reading, dir );
                             // Front wheel steering called
                      break;
                      case 0x4000:
                             Steering2 ( steering_angle, pot_reading, dir );
                             // All wheel steering called
                      break;
                      case 0x8000:
                             Steering4(pot_reading);
                             // Zero radius turn steering called
                      break;
                      case 0xC000:
                        Steering3 ( steering_angle, pot_reading );
                        // Parallel steering called
                break;
                default:
                        Steering1( pot_reading );
                        // Default steering i.e., straight run
                break;
        }
}
else if ( x6 >= 0 && x6 <= 3072)
{
        switch ( mode )
        {
                case 0x0000:
                        Steering1( pot_reading );
                break;
                case 0x2000:
                        Steering1( pot_reading );
                break;
                case 0x4000:
                        Steering2 ( steering_angle, pot_reading, dir );
                break;
                case 0x8000:
                        Steering4(pot_reading);
                break;
                case 0xC000:
                        Steering3 ( steering_angle, pot_reading );
                break;
                default:
                        Steering1( pot_reading );
                break;
        }
}
else if ( x6 > 3072 && x6 < 4096)    // default steering called
{
        switch ( mode )
        {
                case 0x0000:
                        Steering1( pot_reading );
                break;
                case 0x2000:
                        Steering1( pot_reading );
                break;
                case 0x4000:
                        Steering1( pot_reading );
                break;
                case 0x8000:
                        Steering1(pot_reading);
                break;
                case 0xC000:
                        Steering1( pot_reading );
                break;
                default:
                        Steering1( pot_reading );
                break;
        }
}
```

```
}
void Steering_Direction(int x7)
// To determine the steering direction i.e., either left or right where,
x7 = steering_angle
{
      if (steering_angle >= 0 && steering_angle <= 512)
      {
            dir = 1;
      }
      else
      {
            dir=0;
      }
}
int main (void)                                   // Main function
{
      int temp;                // temporary variable to read the SP1BUF

/* Configure Oscillator to operate the device at 40Mhz
   Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
   Fosc= 20M*40/(2*4)=80Mhz for 8M input clock */
      PLLFBD=30;                      // M=32
      CLKDIVbits.PLLPOST=0;           // N1=2
      CLKDIVbits.PLLPRE=2;            // N2=4
// clock switch to incorporate PLL
      __builtin_write_OSCCONH(0x03); // Initiate Clock Switch to Primary
                                     // Oscillator with PLL (NOSC=0b011)
      __builtin_write_OSCCONL(0x01);        // Start clock switching
      while (OSCCONbits.COSC != 0b011); // Wait for Clock switch to occur
                                            // Wait for PLL to lock
      while(OSCCONbits.LOCK!=1) {};

// Peripheral Initialization
      initRP();                // Initialize Remappable Peripheral Pins
      initAdc1();              // Initialize ADC module
      initTmr3();              // Initialize TIMER 3 for ADC conversion
      initPwm1();              // Initialize PWM module
      initIO();                // Initialize Input/output pins
      initSPI();               // Initialize SPI module

      read = 0;
      temp = SPI1BUF;
      while(1)
      {
            pot_reading        = ( ain0Buff[0]+ain0Buff[1]) / 2;
            encoder            = ( ain1Buff[0]+ain1Buff[1]) / 2;
            Direction ( speed, encoder );
            Steering_Direction(steering_angle);
            Wheel_Flip(speed);
            if(PORTBbits.RB3==1)
            {
                  temp=SPI1BUF;
                  SPI1STAT = 0x0000;            //disable SPI1 module
                  IFS0bits.SPI1IF=0;
                   //make sure the SPI interrupt flag is cleared
                  IEC0bits.SPI1IE=0;            // Interrupt disabled
```

```
                SPI1STAT=0x8000;
                SPI1STATbits.SPIROV=0;
                IFS0bits.SPI1IF=0;
                 //make sure the SPI interrupt flag is cleared
                IEC0bits.SPI1IE=1;              // Interrupt enabled
        }
    }
    return (0);
}
```

j. PWM setup for Stepper Motor Control in PIC10

File name: PWM.asm

```
*************************************************************************

list      p=10F206             ; list directive to define processor
#include <p10F206.inc>         ; processor specific variable definitions
__CONFIG   _MCLRE_OFF & _CP_OFF & _WDT_OFF
/*** VARIABLE DEFINITIONS****/
temp0        EQU 0x10
temp1        EQU 0x11
temp2        EQU 0x12
#DEFINE ON        GPIO,1
#DEFINE DIR GPIO,3
#DEFINE PWM1      GPIO,0
#DEFINE PWM2      GPIO,2

ORG     0x1FF              ; processor reset vector
; Internal RC calibration value is placed at location 0x1FF by Microchip
; as a movlw k, where the k is a literal value.
ORG     0x000              ; coding begins here
MOVWF   OSCCAL             ; update register with factory cal value
GOTO  start
bigdelay     MOVLW 0x05
             MOVWF temp1
bigdelay_10 MOVLW 0xFF
             MOVWF temp0
bigdelay_11 DECFSZ      temp0,1
             GOTO  bigdelay_11
             DECFSZ      temp1,1
             GOTO  bigdelay_10
             RETLW  0
start
             BCF         CMCON0,3           ; Comparator OFF
             MOVLW 0x0A                     ; Set GPIO 0,2 as an output
             TRIS        GPIO
             MOVLW 0x00
             OPTION
             BSF         ON
             BSF         DIR
             BCF   PWM1
             BCF         PWM2
Repeat
             BTFSS       ON
             GOTO  Repeat
             BTFSS DIR
             GOTO  LOOP1
             BTFSS       ON
             GOTO  Repeat
             GOTO  LOOP2
LOOP1
             MOVLW 0x04
             MOVWF temp2
```

```
LOOP11
            BSF    PWM1
            BSF          PWM2
            CALL  bigdelay
            BSF          PWM1
            BCF          PWM2
            CALL  bigdelay
            BCF          PWM1
            BCF          PWM2
            CALL  bigdelay
            BCF          PWM1
            BSF          PWM2
            CALL  bigdelay
            BTFSS ON
            GOTO  LOOP3
            BTFSS DIR
            GOTO  LOOP11
            GOTO  Repeat
LOOP2
            MOVLW 0x04
            MOVWF temp2
LOOP22
            BSF    PWM2
            BSF          PWM1
            CALL  bigdelay
            BSF          PWM2
            BCF          PWM1
            CALL  bigdelay
            BCF          PWM2
            BCF          PWM1
            CALL  bigdelay
            BCF          PWM2
            BSF          PWM1
            CALL  bigdelay
            BTFSS ON
            GOTO  LOOP4
            BTFSC DIR
            GOTO  LOOP22
            GOTO  Repeat
LOOP3
            DECFSZ       temp2,1
            GOTO  LOOP11
            GOTO  Repeat
LOOP4
            DECFSZ       temp2,1
            GOTO  LOOP22
            GOTO  Repeat
            RETLW        0
            END                        ; directive 'end of program'
```

2. VBC

File Name: Central_testrun.c

```
**************************************************************************

#include "p33FJ128MC802.h"
_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_HS);
_FWDT(FWDTEN_OFF);
_FPOR(PWMPIN_OFF & HPOL_ON & LPOL_ON)

int send=0;                     // to send the data via SPI
int READ;                       // To read from the SP1BUF
int cycle=0;
// variable to determine the number of steering turns (might be
repetitive), to be used
int i=0;                        // counter variable
int j=0;                        // counter variable
int k=0;                        // counter variable
int selection;                  // to select the mode of steering
int PORTB1;                     // to read the PortB
int speed=0;                    // speed
int steering=512;               // steering
int mode=0;                     // mode
void Update_data( int x, int y, int z)
// To update the data on the data bus, where x= speed, y = steering
angle, z = mode
{
      send = x+y+z;
}
void FWS_U (int x1, int y1)
{
      mode = x1;
      speed = y1;
      for(i=8000;i>=0;i--)                        // delay function
      {
            for(j=5000;j>0;j--);
      }
      SPI1BUF           = 0;                      // make wheels straight
      PORTBbits.RB3     = 0;                      // starts transmission
      while (SPI1STATbits.SPITBF);
      i=4400;
      while (1)
      {
            for (;i>0;i--)
            {
                  for (j=20;j>0;j--)                    // inner module
                  {
                        while (!SPI1STATbits.SPIRBF);
                        PORTBbits.RB3=1;
                        READ = SPI1BUF;
                        SPI1STAT= 0x0000;
                        for(k=500;k>0;k--);
                        SPI1STATbits.SPIROV      = 0;
                        //make sure the overflow flag is cleared
                        SPI1STAT = 0x8000;       //enable SPI1 module
                        switch(cycle)
                        {
                              case 0:
                                    Update_data(speed,0,0);
                                    // First motion statement
                              break;
                              case 1:
                                    Update_data(speed,400,mode);
                                    // second motion statement
                              break;
                              case 2:
                                    Update_data(speed,0,0);
                                    // third motion statement
                              break;
                              case 3:
                                    Update_data(0,0,0);
                                    // final stop
                              break;
                              case 4:
                                    while(1);
                              break;
                        }
                        SPI1BUF             = send;
                        PORTBbits.RB3       = 0;  // starts transmission
                        while (SPI1STATbits.SPITBF);
                  }
            }
            cycle=cycle+1;
            switch (cycle)
            {
                  case 1:
                        i=4800;
                  break;
                  case 2:
                        i=4400;
                  break;
                  case 3:
                        i=100;
                  break;
                  default:
                        i=1;
                        j=1;
                  break;
            }
      }
}
void FWS_Zigzag (int x2, int y2)
{
      mode = x2;
      speed = y2;
      for(i=8000;i>=0;i--)                          // delay function
      {
            for(j=5000;j>0;j--);
```

```
}
SPI1BUF             = 0;                    // make wheels straight
PORTBbits.RB3       = 0;                    // starts transmission
while (SPI1STATbits.SPITBF);
i=480;
while (1)
{
      for (;i>0;i--)
      {
            for (j=20;j>0;j--)              //inner module
            {
                  while (!SPI1STATbits.SPIRBF);
                  PORTBbits.RB3=1;
                  READ = SPI1BUF;
                  SPI1STAT= 0x0000;
                  for(k=500;k>0;k--);
                  SPI1STATbits.SPIROV     = 0;
                  //make sure the overflow flag is cleared
                  SPI1STAT = 0x8000;      //enable SPI1 module
                  switch(cycle)
                  {
                        case 0:
                              Update_data(speed,0,0);
                              // First motion statement
                        break;
                        case 1:
                              Update_data(speed,600,mode);
                              // second motion statement
                        break;
                        case 2:
                              Update_data(speed,0,0);
                              // third motion statement
                        break;
                        case 3:
                              Update_data(speed,424,mode);
                              // fourth motion statement
                        break;
                        case 4:
                              Update_data(speed,0,0);
                              // fifth motion statement
                        break;
                        case 5:
                              Update_data(speed,600,mode);
                              // sixth motion statement
                        break;
                        case 6:
                              Update_data(speed,424,mode);
                              // seventh motion statement
                        break;
                        case 7:
                              Update_data(0,0,0);
                              // final stop
                        break;
                        case 8:
                              while(1);
                        break;
                  }
```

```
                        SPI1BUF =  send;
                        PORTBbits.RB3=0;        // starts transmission
                        while (SPI1STATbits.SPITBF);
                    }
                }
                cycle=cycle+1;
                switch (cycle)
                {
                    case 1:
                        i=1900;
                    break;
                    case 2:
                        i=480;
                    break;
                    case 3:
                        i=2700;
                    break;
                    case 4:
                        i=480;
                    break;
                    case 5:
                        i=2700;
                    break;
                    case 6:
                        i=760;
                    break;
                    case 7:
                        i=100;
                    break;
                    default:
                        i=1;
                        j=1;
                    break;
                }
        }
}
void FWS_S (int x3, int y3)
{
        mode = x3;
        speed = y3;
        for(i=8000;i>=0;i--)                                    // delay function
        {
                for(j=5000;j>0;j--);
        }
        SPI1BUF           = 0;                      // make wheels straight
        PORTBbits.RB3     = 0;                      // starts transmission
        while (SPI1STATbits.SPITBF);
        i=1750;
        while (1)
        {
                for (;i>0;i--)
                {
                        for (j=20;j>0;j--)
                        {
                                while (!SPI1STATbits.SPIRBF);
                                PORTBbits.RB3=1;
                                READ = SPI1BUF;
```

```
            SPI1STAT= 0x0000;
            for(k=500;k>0;k--);
            SPI1STATbits.SPIROV     = 0;
            //make sure the overflow flag is cleared
            SPI1STAT = 0x8000;      //enable SPI1 module
            switch(cycle)
            {
                  case 0:
                        Update_data(speed,0,0);
                        // First motion statement
                  break;
                  case 1:
                        Update_data(speed,600,mode);
                        // second motion statement
                  break;
                  case 2:
                        Update_data(speed,0,0);
                        // third motion statement
                  break;
                  case 3:
                        Update_data(speed,424,mode);
                        // fourth motion statement
                  break;
                  case 4:
                        Update_data(speed,0,0);
                        // fifth motion statement
                  break;
                  case 5:
                        Update_data(0,0,0);
                        // final stop
                  break;
                  case 6:
                        while(1);
                  break;
            }
            SPI1BUF =  send;
            PORTBbits.RB3=0;        // starts transmission
            while (SPI1STATbits.SPITBF);
      }
}
cycle=cycle+1;
switch (cycle)
{
      case 1:
            i=1450;
      break;
      case 2:
            i=1330;
      break;
      case 3:
            i=1450;
      break;
      case 4:
            i=860;
      break;
      case 5:
            i=100;
```

```
                break;
                default:
                      i=1;
                      j=1;
                break;
          }
      }
}
void AWS_U (int x4, int y4)
{
      mode = x4;
      speed = y4;
      for(i=8000;i>=0;i--)                         // delay function
      {
            for(j=5000;j>0;j--);
      }
      SPI1BUF           = 0;                       // make wheels straight
      PORTBbits.RB3     = 0;                       // starts transmission
      while (SPI1STATbits.SPITBF);
      i=4400;
      while (1)
      {
            for (;i>0;i--)
            {
                  for (j=20;j>0;j--)
                  {
                        while (!SPI1STATbits.SPIRBF);
                        PORTBbits.RB3=1;
                        READ = SPI1BUF;
                        SPI1STAT= 0x0000;

                        for(k=500;k>0;k--);

                        SPI1STATbits.SPIROV     = 0;
                        //make sure the overflow flag is cleared
                        SPI1STAT = 0x8000;    `  //enable SPI1 module
                        switch(cycle)
                        {
                              case 0:
                                    Update_data(speed,0,0);
                                    // First motion statement
                              break;
                              case 1:
                                    Update_data(speed,400,mode);
                                    // second motion statement
                              break;
                              case 2:
                                    Update_data(speed,0,0);
                                    // third motion statement
                              break;
                              case 3:
                                    Update_data(0,0,0);
                                    // final stop
                              break;
                              case 4:
                                    while(1);
                              break;
                        }
                        SPI1BUF           = send;
                        PORTBbits.RB3     = 0;  // starts transmission
                        while (SPI1STATbits.SPITBF);
                }
            }
            cycle=cycle+1;
            switch (cycle)
            {
                case 1:
                        i=2700;
                break;
                case 2:
                        i=4400;
                break;
                case 3:
                        i=100;
                break;
                default:
                        i=1;
                        j=1;
                break;
            }
        }
}
void AWS_Zigzag(int x5, int y5)
{
        mode = x5;
        speed = y5;
        for(i=8000;i>=0;i--)                        // delay function
        {
            for(j=5000;j>0;j--);
        }
        SPI1BUF           = 0;                      // make wheels straight
        PORTBbits.RB3     = 0;                      // starts transmission
        while (SPI1STATbits.SPITBF);
        i=480;
        while (1)
        {
            for (;i>0;i--)
            {
                for (j=20;j>0;j--)
                {
                        while (!SPI1STATbits.SPIRBF);
                        PORTBbits.RB3=1;
                        READ = SPI1BUF;
                        SPI1STAT= 0x0000;
                        for(k=500;k>0;k--);
                        SPI1STATbits.SPIROV     = 0;
                        //make sure the overflow flag is cleared
                        SPI1STAT = 0x8000;      //enable SPI1 module
                        switch(cycle)
                        {
                            case 0:
                                    Update_data(speed,0,0);
                                    // First motion statement
                            break;
                        case 1:
                              Update_data(speed,600,mode);
                              // second motion statement
                        break;
                        case 2:
                              Update_data(speed,0,0);
                              // third motion statement
                        break;
                        case 3:
                              Update_data(speed,424,mode);
                              // fourth motion statement
                        break;
                        case 4:
                              Update_data(speed,0,0);
                              // fifth motion statement
                        break;
                        case 5:
                              Update_data(speed,600,mode);
                              // sixth motion statement
                        break;
                        case 6:
                              Update_data(speed,424,mode);
                              // seventh motion statement
                        break;
                        case 7:
                              Update_data(speed,600,mode);
                              // eigth motion statement
                        break;
                        case 8:
                              Update_data(speed,0,0);
                              // ninth motion statement
                        break;
                        case 9:
                              Update_data(0,0,0);
                              // final stop
                        break;
                        case 10:
                              while(1);
                        break;
                  }
                  SPI1BUF             = send;
                  PORTBbits.RB3       = 0;  // starts transmission
                  while (SPI1STATbits.SPITBF);
            }
      }
      cycle=cycle+1;
      switch (cycle)
      {
            case 1:
                  i=1200;
            break;
            case 2:
                  i=480;
            break;
            case 3:
                  i=1200;
            break;
```

```
                        case 4:
                             i=480;
                        break;
                        case 5:
                             i=1200;
                        break;
                        case 6:
                             i=1200;
                        break;
                        case 7:
                             i=1200;
                        break;
                        case 8:
                             i=480;
                        break;
                        case 9:
                             i=100;
                        break;
                        default:
                             i=1;
                             j=1;
                        break;
                  }
            }
      }
void Parallel (int x6, int y6)
{
      mode = x6;
      speed = y6;
      for(i=8000;i>=0;i--)                          // delay function
      {
            for(j=5000;j>0;j--);
      }
      SPI1BUF             = 0;                      // make wheels straight
      PORTBbits.RB3       = 0;                      // starts transmission
      while (SPI1STATbits.SPITBF);
      i=1750;
      while (1)
      {
            for (;i>0;i--)
            {
                  for (j=20;j>0;j--)
                  {
                        while (!SPI1STATbits.SPIRBF);
                        PORTBbits.RB3=1;
                        READ = SPI1BUF;
                        SPI1STAT= 0x0000;
                        for(k=500;k>0;k--);
                        SPI1STATbits.SPIROV      = 0;
                        //make sure the overflow flag is cleared
                        SPI1STAT = 0x8000;       //enable SPI1 module
                        switch(cycle)
                        {
                              case 0:
                                    Update_data(speed,0,0);
                                    // First motion statement
                              break;
```

```
                                        case 1:
                                              Update_data(speed,600,mode);
                                              // second motion statement
                                        break;
                                        case 2:
                                              Update_data(speed,0,0);
                                              // third motion statement
                                        break;
                                        case 3:
                                              Update_data(speed,424,mode);
                                              // fourth motion statement
                                        break;
                                        case 4:
                                              Update_data(speed,0,0);
                                              // fifth motion statement
                                        break;
                                        case 5:
                                              Update_data(0,0,0);
                                              // final stop
                                        break;
                                        case 6:
                                              while(1);
                                        break;
                                  }
                                  SPI1BUF           = send;
                                  PORTBbits.RB3     = 0;  // starts transmission
                                  while (SPI1STATbits.SPITBF);
                            }
                      }
                      cycle=cycle+1;
                      switch (cycle)
                      {
                            case 1:
                                  i=1450;
                            break;
                            case 2:
                                  i=1330;
                            break;
                            case 3:
                                  i=1450;
                            break;
                            case 4:
                                  i=860;
                            break;
                            case 5:
                                  i=100;
                            break;
                            default:
                                  i=1;
                                  j=1;
                            break;
                      }
                }
          }
          void ZRT (int x7, int y7)
          {
                mode = x7;
```

```
speed = y7;
for(i=8000;i>=0;i--)                          // delay function
{
      for(j=5000;j>0;j--);
}
SPI1BUF            = 0;                       // make wheels straight
PORTBbits.RB3      = 0;                       // starts transmission
while (SPI1STATbits.SPITBF);
i=3300;
while (1)
{
      for (;i>0;i--)
      {
            for (j=20;j>0;j--)
            {
                  while (!SPI1STATbits.SPIRBF);
                  PORTBbits.RB3=1;
                  READ = SPI1BUF;
                  SPI1STAT= 0x0000;
                  for(k=500;k>0;k--);
                  SPI1STATbits.SPIROV     = 0;
                  //make sure the overflow flag is cleared
                  SPI1STAT = 0x8000;      //enable SPI1 module
                  switch(cycle)
                  {
                        case 0:
                              Update_data(speed,0,0);
                              // First motion statement
                        break;
                        case 1:
                              Update_data(0,0,0);
                              // second motion statement
                        break;
                        case 2:
                              Update_data(speed,0,mode);
                              // third motion statement
                        break;
                        case 3:
                              Update_data(0,0,0);
                              // fourth motion statement
                        break;
                        case 4:
                              Update_data(speed,0,0);
                              // fifth motion statement
                        break;
                        case 5:
                              Update_data(0,0,0);
                              // final stop
                        break;
                        case 6:
                              while(1);
                        break;
                  }
                  SPI1BUF             = send;
                  PORTBbits.RB3       = 0;  // starts transmission
                  while (SPI1STATbits.SPITBF);
            }
            }
            cycle=cycle+1;
            switch (cycle)
            {
                  case 1:
                        i=100;
                  break;
                  case 2:
                        i=1900;
                  break;
                  case 3:
                        i=500;
                  break;
                  case 4:
                        i=3300;
                  break;
                  case 5:
                        i=100;
                  break;
                  default:
                        i=1;
                        j=1;
                  break;
            }
      }
}
int main (void)                                 // Main function
{
/* Configure Oscillator to operate the device at 40Mhz
   Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
   Fosc= 20M*40/(2*4)=80Mhz for 8M input clock */
      PLLFBD=30;                        // M=32
      CLKDIVbits.PLLPOST=0;             // N1=2
      CLKDIVbits.PLLPRE=2;              // N2=4
// clock switch to incorporate PLL
      __builtin_write_OSCCONH(0x03);// Initiate Clock Switch to Primary
                                        // Oscillator with PLL (NOSC=0b011)
      __builtin_write_OSCCONL(0x01);          // Start clock switching
      while (OSCCONbits.COSC != 0b011); // Wait for Clock switch to occur
                                              //   Wait   for   PLL   to   lock
      while(OSCCONbits.LOCK!=1) {};
// REMAPPABLE PINS CONFIGURATION
//Unlock the registers
      __builtin_write_OSCCONL(0x20);
//Configure SPI1 Port for MASTER mode
      RPINR20       = 0x001F;     // SDI1 input is associated to Vss.
      RPOR0         = 0x0008;
        //remappable pin RP00 (pin 3 of the dsPIC) is associated to SCK
      RPOR1 = 0x0907;     //RP02 and RP03 are output
//Lock the registers   OSCCON | (1<<6);
      __builtin_write_OSCCONL(0x40);
// Configuring input/output ports
      TRISBbits.TRISB0 = 0; //pin RB0/RP0 is configured as output for clk
      TRISBbits.TRISB2 = 0;
      //pin RB2/RP2 is configured as output for data (MOSI)
      TRISBbits.TRISB1 = 0;
      // pin RB1/RP1 is configured as an output (ON/OFF)
```

```
        TRISBbits.TRISB3 = 0;    // SS1 output
        TRISBbits.TRISB12 = 1;   // Motion select bit 0 (Reverse/Forward)
        TRISBbits.TRISB13 = 1;   // Mode select bit 0
        TRISBbits.TRISB14 = 1;   // Mode select bit 1
        TRISBbits.TRISB15 = 1;   // Mode select bit 2
        PORTBbits.RB3=1;
        PORTBbits.RB1=0;
//SPI1 configuration
        IFS0bits.SPI1IF = 0; //make sure the SPI1 interrupt flag is cleared
        IEC0bits.SPI1IE = 0;          // Interrupt disabled

        SPI1CON1          = 0x053A;
        /*configure SPI1 module as Master, CKP=1, CKE=0, SMP=1, first 4:1
        and secondary prescaler are set to 2:1 therefore SCLK frequency is
        5MHz, idle state for clock is low, SCKx and SDOx controlled by the
        SPI module*/

        SPI1STATbits.SPIROV= 0;  //make sure the overflow flag is cleared
        SPI1STAT          = 0x8000;        //enable SPI1 module
        READ = SPI1BUF;
        PORTB1=PORTB;
        selection = (PORTB1 & 0xF000);
        switch ( selection )          // this is for selection of mode
        {
              case 0x0000:
                    FWS_U(0x2000, 0xC000);
                    // U-Turn in Front Wheel Steering
              break;
              case 0x2000:
                    FWS_Zigzag(0x2000, 0xC000);
                    // Zigzag (Continuous) turn in Front Wheel Steering
              break;
              case 0x4000:
                    FWS_S (0x2000, 0xC000);
                    // Zigzag Turn in Front Wheel Steering
              break;
              case 0x8000:
                    AWS_U(0x4000, 0xC000);
                    // U-Turn in All Wheel Steering
              break;
              case 0x1000:
                    AWS_Zigzag( 0x4000, 0xC000);
                    // Zigzag (Continuous) turn in All Wheel Steering
              break;
              case 0x3000:
                    Parallel (0xC000, 0xC000);
                    // Zigzag Turn in Parallel Steering
              break;
              case 0x5000:
                    ZRT (0x0000, 0xC000);
                    // U-Turn in Zero Radius Turn Steering
              break;
        }
        return (0);
}
```