SELF BALANCING ROBOT

ROSYIMAH BINTI ABD AMIN

ELECTRICAL & ELECTRONICS ENGINEERING
UNIVERSITI TEKNOLOGI PETRONAS
JUNE 2010

SELF BALANCING ROBOT

By:

ROSYIMAH BINTI ABD AMIN

Dissertation submitted in partial fulfillment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

June 2010

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

SELF BALANCING ROBOT

by

Rosyimah Binti Abd Amin

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL AND ELECTRONICS ENGINEERING)

Approved by, [illegible]

(Assoc. Prof. Dr. E. Irraivan)

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

June 2010

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

Rosyim

ROSYIMAH BINTI ABD AMIN

ABSTRACT

The aim of this project is to build a Self-Balancing Robot. The robot built comprises three round platforms arranged vertically with two wheels attached at the bottom of each side. The robot is equipped with self-balancing mechanism. The project started with thorough literature review and research and ended with the presentation of a moving two-wheels robot. A programmed microcontroller chip, AtMega16, connected to its development circuit with input component, Memsic2125 accelerometer, and output components, Futaba servo motors, are attached unto the robot as the main elements for the self-balancing mechanism. Hence, the robot should be able to stand upright and balance itself, move from one point to the other as pre-programmed and carry object. The self-balancing robot built possibly counter many problems facing by people nowadays from all background as it is able to carry object efficiently, consumes small space and conquer the not flat floor surfaces. Besides as to could able to serve the community a higher of transportation technology yet manages to take part in contributing to saving the earth mission.

ACKNOWLEDGEMENTS

Alhamdulillah, first and foremost I am expressing my appreciation and praise to Allah for his guidance and blessings throughout my entire final year, eventually dissertation was finally completed and finished successfully on time as the final tasks for my final year project.

I wish to thank all the people who have contributed directly or indirectly in accomplishment of this report. I would like to sincerely thank my supervisor AP Dr E. Irraivan for his supervision and his support in accomplishing my final year project.

My appreciation and gratitude is extended to Mr Illani, the Electrical & Electronic Engineering Department Technician for his help with PCB (Printed Circuit Board) and fabrication of the robot.

My appreciation would be incomplete without giving tribute to Universiti Teknologi PETRONAS (UTP), especially Electrical & Electronics Engineering Department who has equipped me with essential skills for self-learning. Its well-rounded graduate philosophy has proven to be useful in the industry.

I have learned so many things during my final year. Thanks all for being a part of this wonderful experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLE

CHAPTER 1
INTRODUCTION

1.1 Background of Study

Technology develops and enhances in every possible ways in order to fulfill needs and demands raised by human race. One of the daily concerns raised by people from all backgrounds is that, is there any easier ways to carrying things from one point to other point regardless of the sizes and distances efficiently, and this includes transportation mean. Left massive object transportation alone as it is firmly taken care, lets focus on something that can be utilized in smaller area or somewhere that is indoor, specifically. People from all background are lacking of an object carrier that build in smaller size and can be utilized in smaller area yet efficiently transfer object from one point to other point despite the condition of the floor.

Hence, the aim of this project is to build a Self-Balancing Robot. Basically, this project is to produce a two-wheeled self-balancing robot that is equipped with a successful balancing mechanism. The robot comprises three platforms arranged vertically with two wheels attached at the bottom of each sides, Memsic2125 accelerometer as the input component, servo motors as the output components and AtMega16 microcontroller chip as well as the circuitry and the other hardware which it is needed for the robot to work successfully. The robot should able to stand upright and balance itself when either static or moving. Memsic2125 feeds in information on the stability of the robot in terms of tilting angle to the AtMega16. This input data will be process through algorithm and instructions set pre-programmed in the AtMega16.

From this, output data will be prepared and sent to servo motors for them to operate as desired. The servo motors will rotate either clockwise or anti-clockwise with certain speed accordingly as an attempt to keep the robot at balance. This project involves with a precise weight distributed fabricated structure and very effective microcontroller chip, input and output components in ensuring the stability of the robot.

1.2 Problem Statement

As compared to four-wheels trolley, having a platform with only two wheels that is able to balance itself and move around is advantageous as it possibly maneuvers better, move on either flat or sloppy surfaces, and to be further enhances, apply for carrying limitless tasks, hence benefit the community. Yet, the main challenges to this possibility is how to have a two wheels robot that is able to stand upright and balance itself, furthermore carry object from one point to other point efficiently. This is where the significant of this project comes from which is to build a self-balancing robot.

1.3 Objectives

The main objective for this project is to build a self-balancing robot. The robot is comprised platforms with two wheels and equipped with successful balancing mechanism. The self-balancing robot should be able to;

1. Stand upright and balance itself.
2. Move from one point to other point as pre-programmed.
3. Carry an object from one point to the other.

1.4 Scope of Study

This project is divided into five major stages which are literature review, designing, fabrication, assembling and programming. The project started with literature review. In this stage, information about from physics laws and theories related for balancing a structure to real life applications are gathered in developing understanding required in building the self-balancing robot for this project. Then the project continued to designing stage. In this stage, based on the knowledge gathered and understanding developed, detailed designs on physical of the robot, circuitry as well the programming needed in order for the robot to work successfully are drawn. Based on the designs, the each element of the robot is the fabricated. The project is then carried to assembling stage. In this stage, all the hardware are fabricated then together with electronic components and its circuitry constructed for the project are assembled for the self-balancing to take form as designed. Testing is then took place in ensure the working and reliability of the mechanical part of the robot. Then, the final stage of this project which is programming took turn. In this stage, the programming codes are listed and programmed unto the AtMega16 microcontroller chip. The testing is then took place to ensure that the self-balancing robot built meets the objectives of this project. In any part error occurred, it is corrected instantaneously.

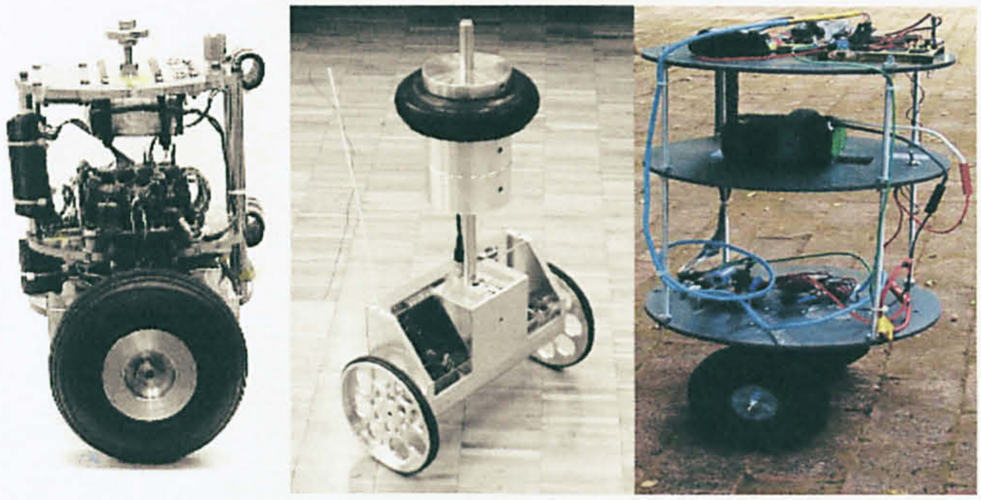
CHAPTER 2
LITERATURE REVIEW

2.1 Overview on Self-Balancing Robot

Over the last decade, the research on self-balancing robot has gained interest of many robotics enthusiasts as well gained momentum in a number of robotics laboratory around the world. This situation is due to the inherent unbalance dynamics of the system. Such robots are built with ability to self-balance on its two wheels. This ability added better maneuverability into its features and allows effective navigation on various terrains. A two-wheeled self-balancing robot that is able to balance itself and carry object from one point to the other more efficiently especially when compared to the widely used four-wheeled trolley as its trail is limited to flat. These capabilities have the potential to conquer the challenges in industry and society.

As a matter of fact, the application of self-balancing robot is beyond the boundary. With its ultimate capability being added with more additional features and its physical structure are specified into its purpose, and then it can serve the community and encounter the problems of the many raised. For example, a motorized wheelchair utilizing this technology would give the operator greater maneuverability and thus access to place most able-bodied people take for granted [3]. Small carts built utilizing this technology allows humans to travel short distance in a small area or factories as opposed to using cars or buggies which is more polluting [3]. Besides that, self-balancing robot can be applied in small business or domestic areas, such as an object carrier which specifically be an orders carrier around a restaurant, for example, or an object carrier around a house. Adding suitable camera to it, it may be carry surveillance task efficiently [16]. Indeed the application of self-balancing robot is limitless.

Many individuals have actually built a self-balancing robot and named their robot as they accordingly. Figure 1 shows a few of the self-balancing robots built by David P. Anderson, Felix Grasser et all and Ted Larson, named as nBot, Joe and Bender accordingly. Some of these self-balancing robots have participated in several of robotics competition held in many places and have intrigued the other parties to get involved with this massive phenomena.



(a) (b) (c)

Figure 1: a) nBot, b) Joe c) Bender

Moreover, some parties have taken the idea of self-balancing robot to a higher level and have actually produced a product that is now available in market for all people to get benefited from it. Segway H.T, shown in figure 2, is one of these marketed products. Note that, Segway H.T is a human transporter which is designed by Dean Kamen and is produced by Segway Incorporation. This human transporter is available in almost all countries and they come in various models suited for specific purposes. In addition to this good news, this human transporter, Segway H.T, is environmental friendly as it will not pollute the environment. Hence, people nowadays can get one level ahead in terms of transportation technology yet contribute to saving the earth mission.

(a) (b)

Figure 2: Segway H.T model; a) i2 and b) x2 adventure

2.2 Theories applied for Balancing Mechanism

Despite the diversity in possible application of self-balancing robot, still the ultimate puzzle needed to be solve here is how to have a two-wheeled robot to balance itself? The balancing mechanism which it is required to work as desired has to be ascertained or else it will not work successfully. Understanding in related physic laws and theories is prior in deriving the balancing mechanism. Also, understanding and get familiar with the main electronic components which are essential in building a self-balancing robot is as well a must.

2.2.1 Inverted Pendulum

Inverted pendulum states that an unbalance structure may be stabilized by oscillating the support rapidly up and down [1]. In experiment, inverted pendulum theory is always modeled with a cart and a vertical stick that attached either above or below the cart. For this self-balancing robot concern, focus the modeling to the one with stick attached at the above part of the cart as shown in figure 3. This vertical stick falls at either side due to instability. The cart has to move to either side (left or right) as an attempt to get the vertical stick at balance, upright. The same condition in applied in balancing a two-wheeled robot. The wheels of the robot have to continuously move to back and forth to achieve stability.

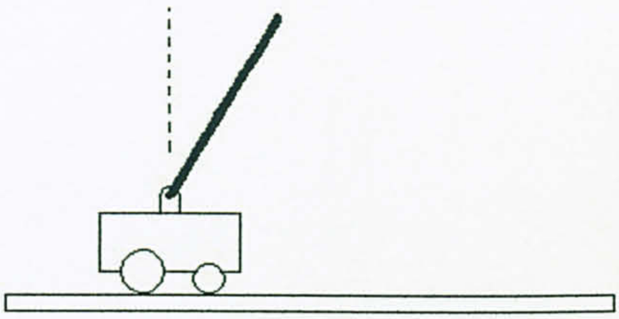


Figure 3: Inverted Pendulum Modeling

2.2.2 Rotational Inertia (Balancing a stick on one hand)

Another theory that is related in deriving the balancing mechanism for self-balancing robot is rotational inertia. Rotational inertia states that for a body to be at equilibrium (balance), there must be no net force acting on it [17]. For clear understanding, consider a modeling for rotational inertia which is balancing a stick on one hand. Referring to figure 4, a stick falls due to the gravity and there is one hand where it is placed on as to support it. As the stick is falling (losing its vertical straight orientation) due to the gravity force which as well is indicated as unbalance, hand has to move horizontally straight, to the left and to the right, as to apply force, of the same magnitude but opposite in direction, the stick to keep it from falling. The direction of the hand movement as to in accordance to the direction of which side the stick is falling to. Besides that, the speed of the hand movement has to proportional to the degree of falling. Thus, in order to keep the stick from unbalance condition, these two forces (force exerted by hand and force due to the gravity) has to apply on the stick to keep at the same time.

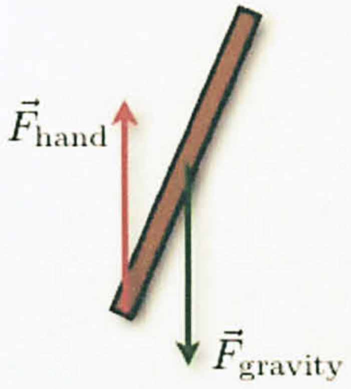


Figure 4: Forces acting on a vertical stick that is balanced on one hand

2.3 Main Electronic Components essential in building Self-Balancing Robot

Following are the main electronic components, input components, output component and the vital one, microcontroller, generally used in building self-balancing robot. Other electronic components may as well be added into the robot description as for specific purposes or perhaps for more advance self-balancing robot. Yet, what matters the most are the components that allows the robot to self-balance successfully.

2.3.1 Input Components for balancing mechanism

The most widely used input components by many self-balancing robot inventors are gyroscope and accelerometer. Fundamentally, note that gyroscope used to measure angular momentum of a body whereas for accelerometer used to measure the gravitational acceleration of a body. Some used both of the input components for better control of stability of the robot. Others used only one of the input components yet still manage to get their robots to self-balance.

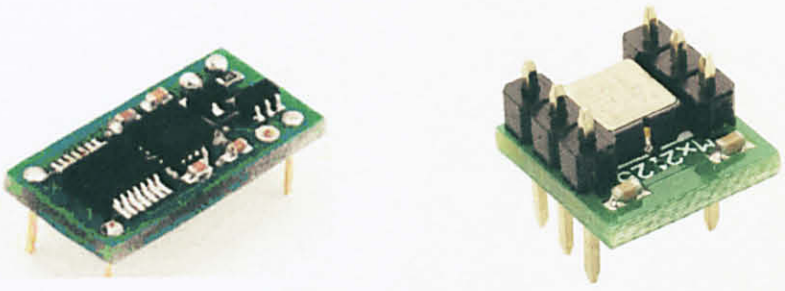
<u>Gyroscope</u>

Basically, gyroscope used to measure the position of a body. It is widely used in a system called Inertial Navigation System (INS), as in to control the navigation of an aerospace [5]. Gyroscope measures the angular momentum acting on an airplane, for example, to provide precise information on the airplane's position to the INS hence for INS to control the navigation of that airplane [5]. Usually, one airplane is equipped with a lot of gyroscopes, considering its size and to get precise information on its position.

However, this inertial system is widely used in aerospace applications but not quite in robotics applications. This situation is due to the high-quality aerospace inertial system is comparatively too expensive for most robotics application. Despite that, due to the needs of the automotive industry, a low-cost solid-state inertial system is increasingly being made commercially available. Although a considerable improvement on past systems, they clearly provide substantially less accurate position information than equivalent aerospace systems. An approach is developed to undertake this problem. This approach is to incorporate in the systems a piori information about the error characteristics of the inertial sensors and to use it directly in a Kalman Filter to estimate position before supplementing the gyroscope with absolute sensing mechanism. [5] In fact, some robot inventors have implemented gyroscope in robotics inventions for the same purpose, to measure the precise information on the robot's position hence for better control of the robot movement.

Accelerometer

The function of accelerometer is to measure the acceleration of which a body is experiencing due to the gravity. This acceleration is divided into two types which are static; constant force of gravity pulling down, and dynamic; caused by moving or vibrating [14]. Note that, by measuring the amount of static acceleration due to gravity, the angle that the device is tilted at with respect to the earth can be determined and by sensing the amount of dynamic acceleration, the way the device is moving can be analyzed [17]. There are two types of accelerometer, categorized according to its features; 2-axis accelerometer and 3-axis accelerometer, as shown in figure 5. 2-axis accelerometer measures acceleration with reference only to X-axis and Y-axis whereas 3-axis accelerometer measures acceleration with reference to X-axis, Y-axis and Z-axis.



(a) (b)

Figure 5: a) 3 axis accelerometer board and b) 2-axis accelerometer

There are several different ways to make an accelerometer. Some accelerometers made use of piezoelectric effect. This type of accelerometer contains microscopic crystal structure that gets stressed by the accelerative forces which cause a voltage to be generated. Another way to do it by sensing changes in capacitance. The other type of accelerometer made use of hot air bubbles mechanism. Internally, this type of accelerometer contains a small heater. This heater warms a “bubble” of air within the device. When gravitational forces act on this bubble it moves. This movement is detected by very sensitive thermopiles (temperature sensors) and the onboard electronics convert the bubble position [relative to g-force] into pulse outputs for the X-axis and Y-axis. There are more methods those including the use of piezoresistive effect, hot air bubbles and also light. [17][14]

In application, accelerometer is used to understand a body's surrounding better, whether it is driving uphill or it is going to fall over when it takes another step and etc. A good algorithm can interpret data provided by an accelerometer and turn them into comprehendible version such as answers to the above questions. An accelerometer can analyze problems in a car engine using vibration testing or actually use one to make a musical instrument. [17]

In the computing world, IBM and Apple have recently started using accelerometers in their laptops to protect hard drives from damages. If one accidentally drop the laptop, the accelerometer detects the sudden freefall, and switches the hard drive off so the heads do not crash on the platters. In a similar fashion, high accelerometers are the industry standard way of detecting car crashes and deploying airbags at just the right time. [17]

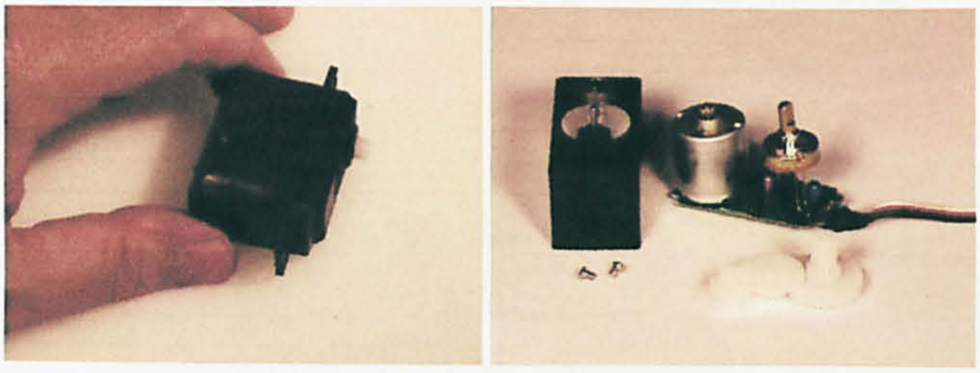
2.3.2 Output Component

Usually, self-balancing robot is built with two wheels as to have it to move any direction as well as a part of balancing mechanism. Hence, the output component for common self-balancing robot is the motors that control the wheels. Different types of motor may be used depending on the description of the robot. However, for robotics projects the most common used motor is servo motors.

Servo Motor

Generally, servo motor is a motor used for motion control in robots, hard disc, etc. It is a small device that has an output shaft. This shaft can be positioned to specific angular positions by sending the servo a coded signal. As long as the coded signal exists on the input line, the servo will maintain the angular position of the shaft. As the coded signal changes, the angular position of the shaft changes. In practice, servos are used in radio controlled airplanes to position control surfaces like the elevators and rudders. They are also used in radio controlled cars, puppets, and of course, robots. [20]

Servos are extremely useful in robotics. The motors are small, as shown in figure 6, they have built in control circuitry, and are extremely powerful for their size. A standard servo such as the Futaba S-148 has 42 oz/inches of torque, which is pretty strong for its size. It also draws power proportional to the mechanical load. A lightly loaded servo, therefore, does not consume much energy. The guts of a servo motor are shown in Figure 8; control circuitry, the motor, a set of gears, and the case, with 3 wires that connect to the outside world. Note that, one is for power (+5volts), ground, and the white wire is the control wire. [20]



(a) (b)

Figure 6: a) A Futaba S-148 Servo and b) a servo disassembled

The servo motor has some control circuits and a potentiometer (a variable resistor or well known as pot) that is connected to the output shaft. Referring to Figure 8, the pot can be seen on the right side of the circuit board. This pot allows the control circuitry to monitor the current angle of the servo motor. If the shaft is at the correct angle, then the motor shuts off. If the circuit finds that the angle is not correct, it will turn the motor the correct direction until the angle is correct. The output shaft of the servo is capable of travelling somewhere around 180 degrees. Usually, it is somewhere in the 210 degree range, but it varies by manufacturer. A normal servo is used to control an angular motion of between 0 and 180 degrees. A normal servo is mechanically not capable of turning any farther due to a mechanical stop built on to the main output gear, or else modification can be made to the servo motor to have it to rotate continuously. [20]

The amount of power applied to the motor is proportional to the distance it needs to travel. So, if the shaft needs to turn a large distance, the motor will run at full speed. If it needs to turn only a small amount, the motor will run at a slower speed. This is called proportional control. [20]

The angle of the rotation of servo motor is determined by the duration of a pulse that is applied to the control wire. This is called Pulse Coded Modulation. The servo expects to see a pulse every 20 milliseconds (.02 seconds). The length of the pulse will determine how far the motor turns. A 1.5 millisecond pulse, for example, will make the motor turn to the 90 degree position (often called the neutral position). If the pulse is shorter than 1.5 ms, then the motor will turn the shaft to closer to 0 degrees. If the pulse is longer than 1.5ms, the shaft turns closer to 180 degrees. [20]

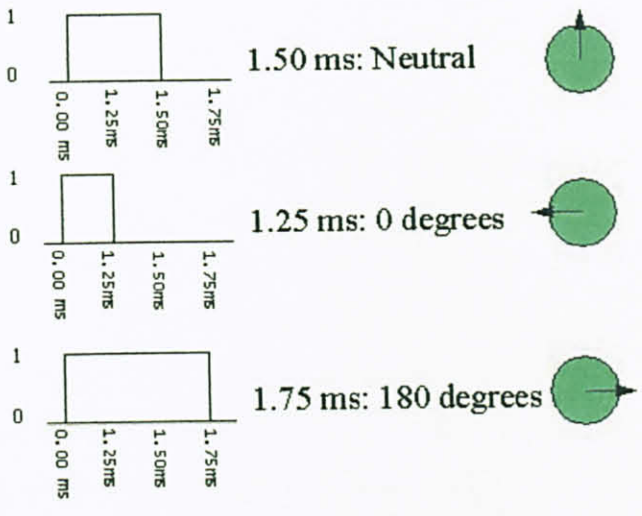


Figure 7: The pulse dictates the angle of the output shaft

With reference to figure 7, the duration of the pulse dictates the angle of the output shaft (shown as the green circle with the arrow). Note that the times here are illustrative and the actual timings depend on the motor manufacturer. The principle, however, is the same.

2.3.3 Microcontroller

A microcontroller chip is the vital electronic components for any robotics project. This is due to the fact a robot system is a computerized system, hence needs microcontroller for it to work with the connection to input and output components. Even in other applications, there has to be a controller to control a whole system.

A microcontroller (also known as microcomputer, MCU or µC) is a small computer on a single integrated circuit consisting internally of a relatively simple CPU, clock, timers, I/O ports and memory [22]. Microcontroller is designed for small dedicated applications.

Usually, microcontroller chips need to work incorporated with its appropriate development circuit for it two operate properly. This development circuit may differ one and another depending on the manufacturer and the series of that microcontroller chip. This development circuit may comprise power circuit, reset circuit, external clock circuit, I/O connections and better with in-circuit programming circuit.

As well for the programming language, again this is depending on the manufacturer and series of that microcontroller chip. Most microcontroller chip uses assembly programming language; C, others may use high-level programming language. This language is for writing the programming code which would make the robot work as desired. The programming codes may then be compiled, assembled and then programmed into the chip. Another device is necessary for programming the microcontroller chip, which is a programmer.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, and toys. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems. [22]

CHAPTER 3
METHODOLOGY

3.1 Project Activities

Project activities undertaken for this project towards completion are divided into five major stages which are literature review, designing, fabrication, assembling and programming. The detailed project activities undertaken are as shown in figure 8.

This project has been divided into five major tasks namely literature review, designing, fabrication, assembling and programming, generally. Literature review took place at start allowed gathering as much as knowledge and understanding in any laws, concepts or principals in getting this project at success. On the basic of knowledge and understanding gained from literature review, design for the self-balancing robot is drawn. This design is including the physical structure of the robot, circuitry as well as the flowchart for the programming the microcontroller chip purpose. Based on the designs, each element of the robot is fabricated as well. Further, as all the hardware for the project is obtained, they are all assembled. Testing is done in making sure the mechanical and electrical work as desired. Further is programming the microcontroller chip. The microcontroller chip used in this project is AtMega16, Atmel manufactured. Thus, familiarity and basic knowledge in programming the AtMega16 is prior to successfully program the chip hence fulfill the objectives for this project.

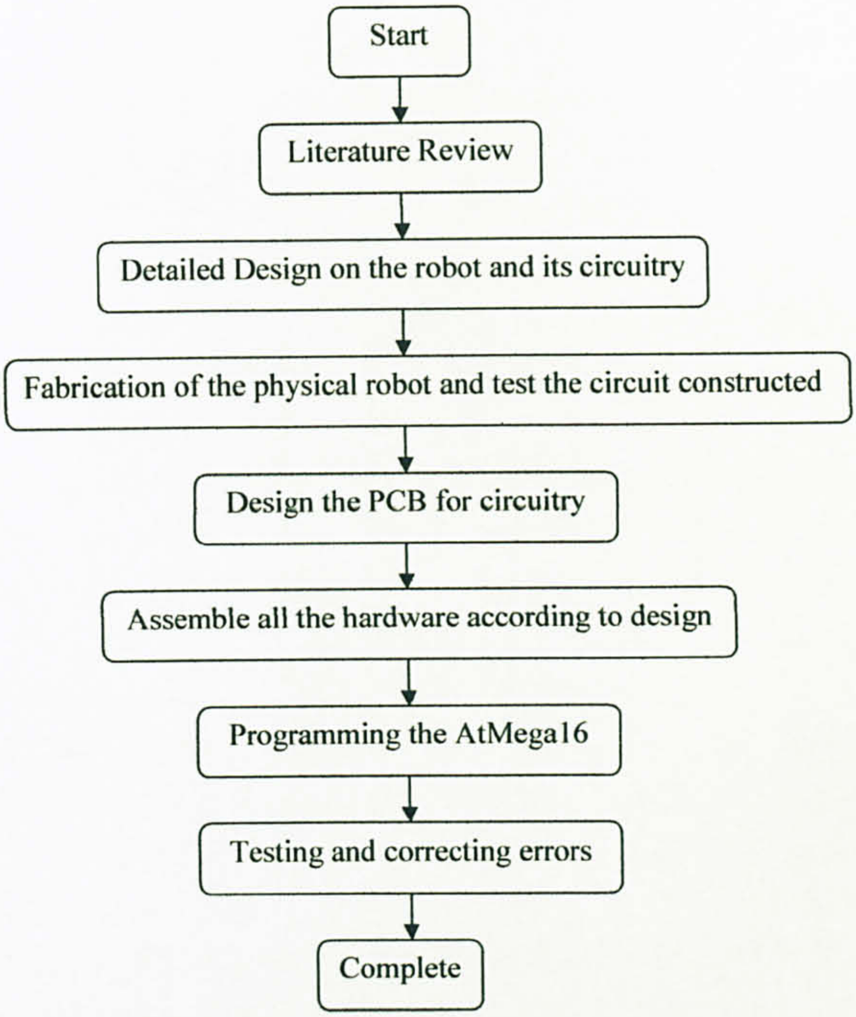

Start
Literature Review
Detailed Design on the robot and its circuitry
Fabrication of the physical robot and test the circuit constructed
Design the PCB for circuitry
Assemble all the hardware according to design
Programming the AtMega16
Testing and correcting errors
Complete

Figure 8: Flowchart for project activities

3.1.1 Gantt Charts

Gantt Charts are as to ensure that the project activities undertaken are on track with period specified, one year. Hence, to ensure that the project carried would be completed within the given period.

3.2 Literature Review Methodology

Thorough research is essential for this project. Therefore, reading materials from various sources are searched and gathered in developing understandings which are necessary for this project. The main source for the research is Internet. Encyclopedias, uploaded researches, patent-searches and uploaded projects on self-balancing robot from various sites are downloaded to obtain as much information hence derive clear understandings as possible. Other than internet, published materials also have been taken in. Several published books are referred to. Moreover, a few individuals are as well referred to eliminate confusion and misunderstanding.

3.3 Design of Self-Balancing Robot

In reference to knowledge gained in literature review stage, balancing mechanism needed for the robot to balance itself is determined. Hence the design for physical of the robot and design for the circuitry which the robot required to work properly are drawn.

3.3.1 Balancing Mechanism for Self-Balancing Robot

Inverted pendulum theory states that an unbalance structure may be stabilized by oscillating the support rapidly up and down. Rotational inertia theory states that for a body to be at equilibrium (balance), there must be no net force acting on it. A mechanism to balancing a vertical stick on one indicates that to have the vertical stick at balance, move the stick at the direction where the stick is falling to, also the speed of movement is proportional to the degree of falling. From these statements, the balancing mechanism for self-balancing robot is determined.

The input component for this project is Memsic2125 accelerometer. The Memsic2125 works as a sensor that senses the tilting angle of the robot. This tilting angle is obtained from measurement of the acceleration of the robot as it falls due to gravity. Note that, as the robot falls due to gravity it actually tilts at either side. The acceleration sensed by the Memsic2125 is interpreted as the tilting angle of the robot as it is tilting, as illustrated in figure 9.

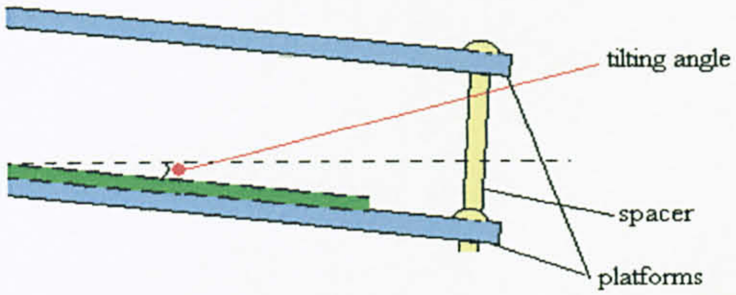
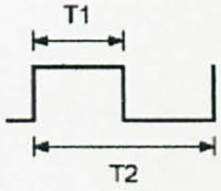


Figure 9: Tilting angle as the robot tilted

Note that, the output of Memsic2125 is pulses form of duty cycles. The pulse outputs from the Memsic2125 are set to a 50% duty cycle at 0 g. The duty cycle changes in proportion to acceleration. The higher the acceleration that is sensed, the more the duty cycle is. Hence, through calculation (refer to figure 10) the tilting angle can be derived and be used in preparing the output instruction for servo motor.



$$\mathbf{A(g) = ((T1\ /\ T2) - 0.5)\ /\ 12.5\%}$$

Figure 10: Memsic2125 Pulse Output

The output components used for this project is servo motor. Note that, one servo motor is used for each wheels, hence two servo motors are used in this project. The main function of servo motor in this project is to control the rotation as well the speed of the wheels in according to the instruction executed by the AtMega16. The direction of rotation of the wheels has to be ascertained in achieving the stability of the robot. In a case where the robot is tilting at front, the servo motor have to direct the wheel to move forward, drive the wheels in the direction that the upper part of the robot is falling, and vice versa, as illustrated in figure 11. As for the speed of the wheels, this is depending on the degree of the tilting angle. The larger the degree of the tilting angle, the faster the wheels should move. This condition is for recovering the robot from instability. This condition is applying the rotational inertia law. As the robot is tilted (take it is tilted at front) a force downward is acted at front due to gravity. Because of this condition, in order to have a structure at equilibrium (balance; no net force) a force of the same magnitude but different direction is required. Let the robot to move forward as applying inertia to counter the robot from continues to fall. This law is applied throughout the system as a mechanism that to keep the robot balance.

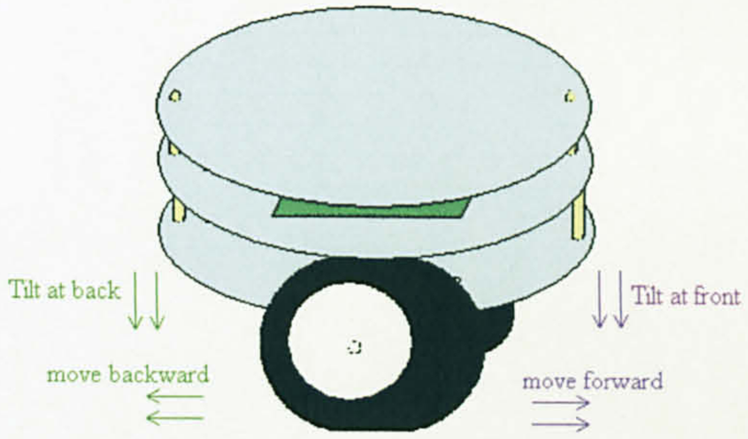
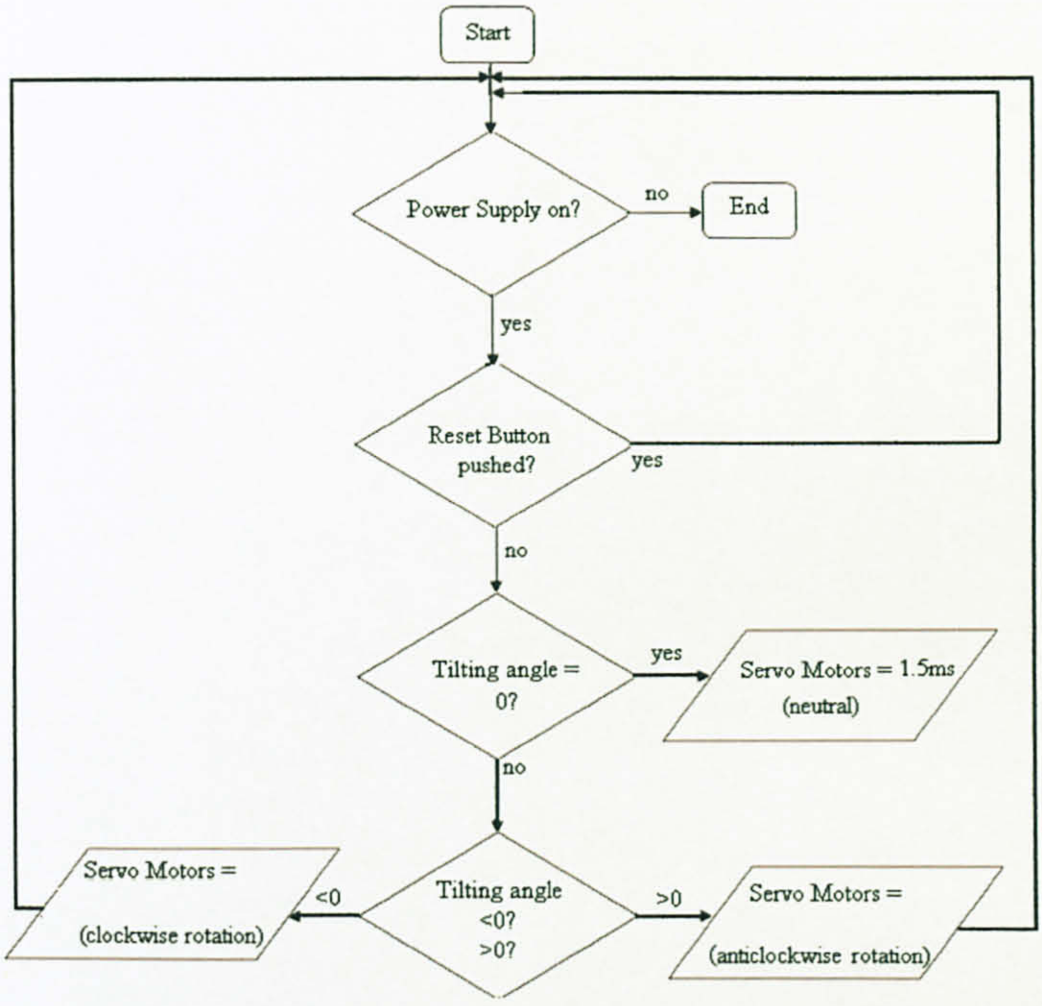


Figure 11: Operation of the wheels of the robot

Based on the above mechanism determined for balancing the robot, a flowchart is derived in order to program the AtMega16 orderly, as well to ease the programming stage. Figure 12 shows the flowchart used in programming the AtMega16.



Start
Power Supply on?
no
End
yes
Reset Button pushed?
yes
no
Tilting angle = 0?
yes
Servo Motors = 1.5ms (neutral)
no
Tilting angle <0? >0?
<0
>0
Servo Motors = (clockwise rotation)
Servo Motors = (anticlockwise rotation)

Figure 12: Flowchart for programming AtMega16

Referring to the figure 12, note that, the AtMega16 flow of operation begins at start and finishes at end. Once the flow is started, it ensures whether the power supply is turned on or off. In a case, where the power supply is not turned on, the flow goes to end and quit the whole flow. In another case where the power supply is turned on, it continues to reading reset button.

Once it reads reset button, it detects whether the reset button is pressed or not. In a case where the reset button is pressed, the flow goes back to start. In another case where the reset button is not pressed, the flow continues to the getting input data from Memsic2125.

As stated, Memsic2125 provides information on tilting angle of the robot, whether the robot is tilting at front or at back. This tilting angle is indicated by the position of the robot. For this project, take whichever side as front and the other one as back and determined the arrangement of the servo motors. In a case where the wheels are required to move forward, pre-ascertained this condition hence set the rotation of the wheels, either clockwise or anti-clockwise. Note that, Memsic2125 senses the acceleration of which the robot experienced at X-axis and Y-axis, as indicated in figure 13. Hence, set the positive X-axis as front and negative X-axis as back as illustrated in figure 14. Furthermore, in a case where the robot is tilted at front, the tilting angle being sensed is negative as illustrated in figure 15. In another case where the robot is tilted at back, the tilting angle being sensed is positive as illustrated in figure 16. These pre-determined conditions would simplify the programming codes as well as reduce confusion.

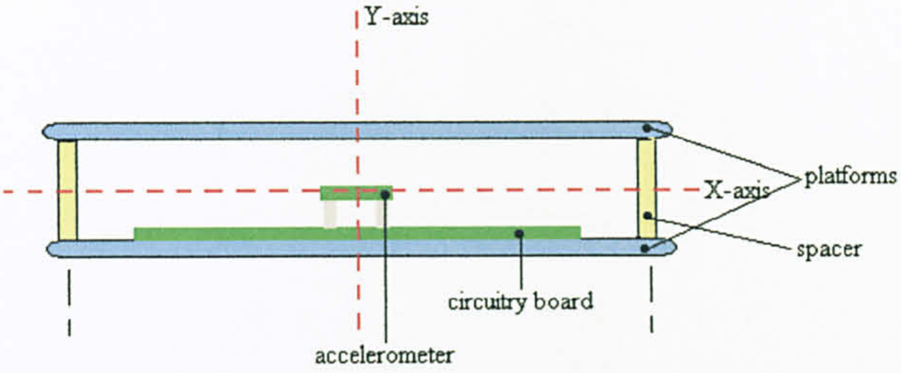


Figure 13: X-axis and Y-axis of accelerometer

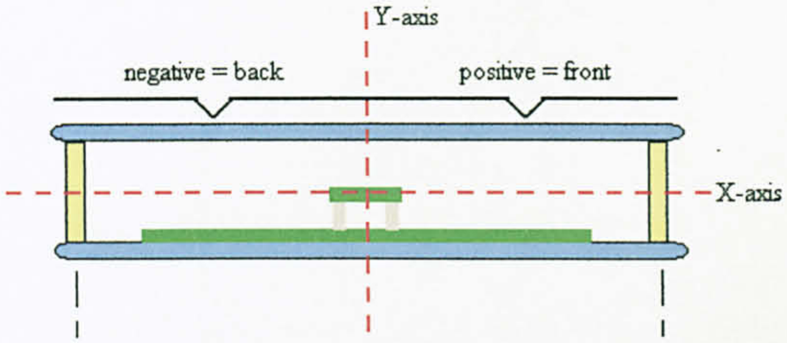


Figure 14: Front and back side of the robot

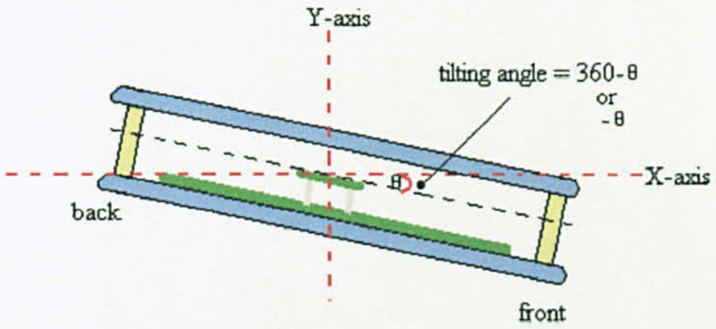


Figure 15: Robot is tilted at front

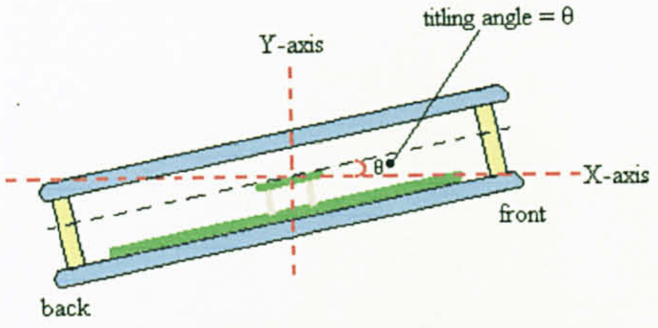


Figure 16: Robot is tilted at back

As for the operation, in a case where the robot is tilted at the front, AtMega16 would receive input data of negative tilting angle from Memsic2125 hence instructed the servo motors to move forward, rotate in clockwise direction, as an attempt to recover the robot from tilting, as shown in figure 17. In another case where the robot is tilted at the back, AtMega16 would receive input data of positive tilting angle from Memsic2125 hence instructed the servo motors to move backward, rotate in anti-clockwise direction, as an attempt to recover the robot from tilting as shown in figure 18.

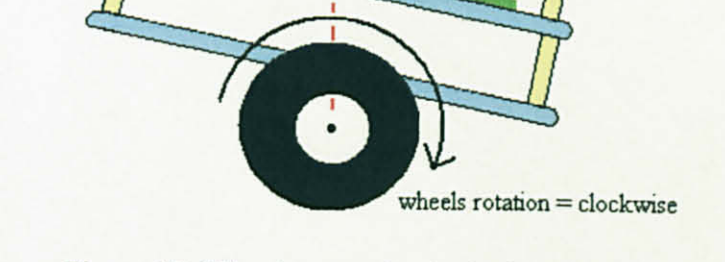


Figure 17: Wheels rotate in clockwise direction

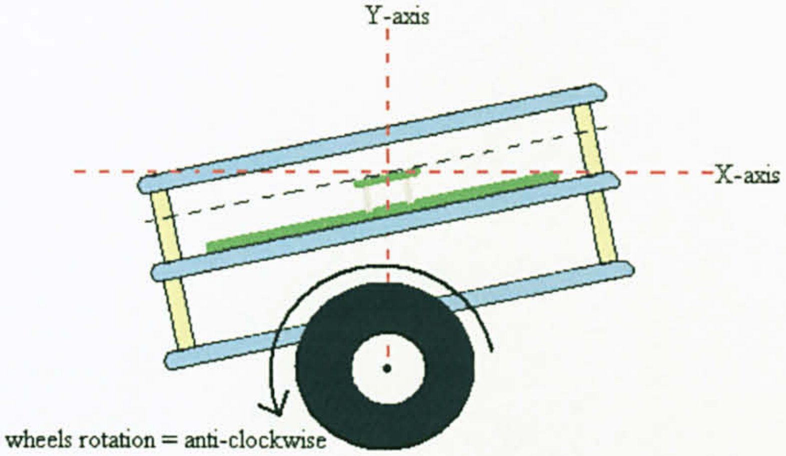


Figure 18: Wheels rotate in anti-clockwise direction

According to the flowchart in figure 12, this flow is looping back to the beginning of the flow, start, where the AtMega16 is then repeat the same operation, until the power supply is turned off which will send the flow to end. This flow allows the Memsic2125 and servo motors to continuously send and receive data to and from AtMega16 respectively. Memsic2125 is always checking on the stability of the robot, whether it is tilting at the front or at the back, as well as the degree of the tilting angle. The AtMega16 is always running the input data through calculation and logics algorithm and prepare the output data. Servo motors are always receiving output data from the AtMega16 to always move either forward or backward accordingly. This condition allows the robot to self-balance itself so long the power is supplied to the circuitry.

3.4 Design of Self-Balancing Robot

3.4.1 Physical Design

The design of the physical of the robot has in reference to the principal determined for balancing the robot. As well as meeting the third objective, the physical of the robot must able to carry object by mean of equip with a feature for object to place on.

As for this matter, the structure is chose to be able to stand upright (vertically). This is copying the phenomena of balancing a vertical stick on one hand. For this project, the structure design comprises three round platforms arranged vertically, two wheels at the bottom of each side, with circuitry, servo motors and batteries attached to the structure as illustrated in figure 19.

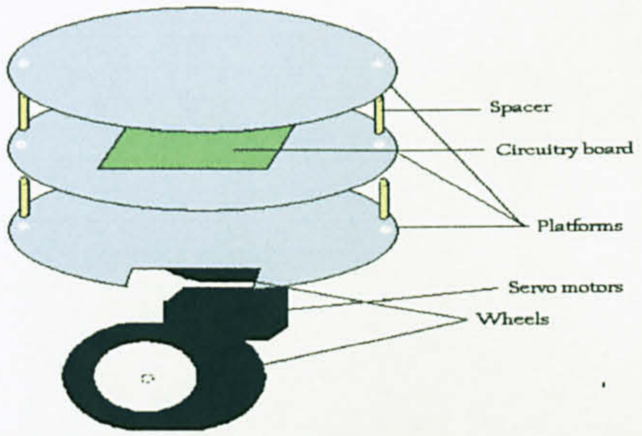


Figure 19: Illustration for physical structure

The base area for this structure is chose to be reasonably wide. Theoretically, the wider the base area is the more stable the structure would be. Even so, the structure for this project is not required to be perfectly stable without the balancing mechanism. The structure needed to wobble or tilt at significant for the Memsic2125 to sense it easily, or else the balancing mechanism may not work as expected. At the same time, the structure should not be in the state of not very stable or else it would difficult for it to balance. Hence, the base is determined as 150mm in diameter in accordance to the dimension of the platforms, as illustrated in figure 20. Note that, the size of the wheels, which is 50mm in diameter, is taken into consideration in determining the base area. Also note that, the material of the platforms is 2.5mm (thick) Perspex. The purpose of this choice is to keep the total weight of the structure at a range that the servo motors can operate at, which is 8 kg at maximum.

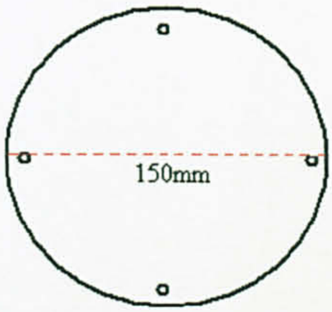


Figure 20: Dimension of the platform

As stated, three round platforms are used in this project. The second and the third platforms from below (as arranged) are kept at round shape. The first platform from below, which is the base platform, is at round shape with two rectangle shapes are taken out at each side, as shown in figure 21. Those rectangles are meant for the wheels as a purpose to keep the base area of the structure at round shape with 150mm in diameter.

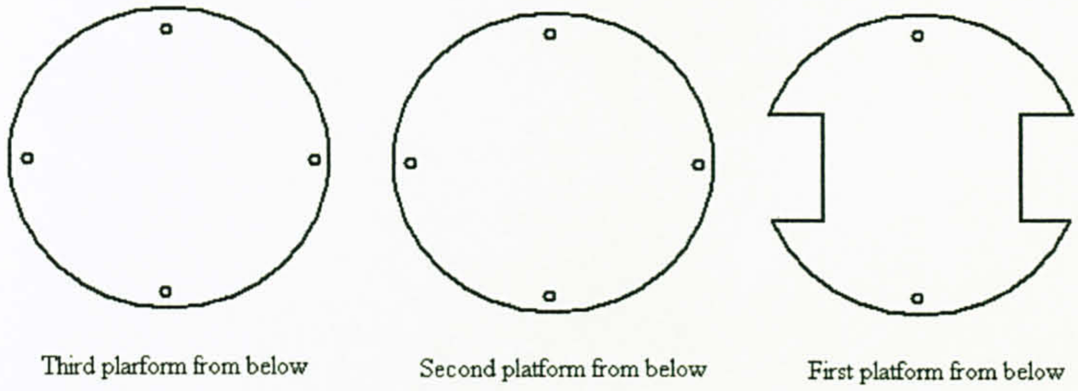

Third plarform from below
Second platform from below
First platform from below

Figure 21: Platforms for structure

The dimension of the wheels for this project is chose to be reasonably large also depends on its availability in market. Hence the dimension of the wheels is 50mm at diameter and 25mm at wide. The wheels are Tamiya manufactured.

3.4.2 Circuitry Design

Note that, different microcontroller chip requires different development circuit for it to work properly. For this project, since the microcontroller chip used is AtMega16, the development circuit is as shown in figure 22. This development circuit comprises power circuit, reset circuit, clock circuit, input and output connections, and, as for in-circuit programming purpose, serial programming circuit.

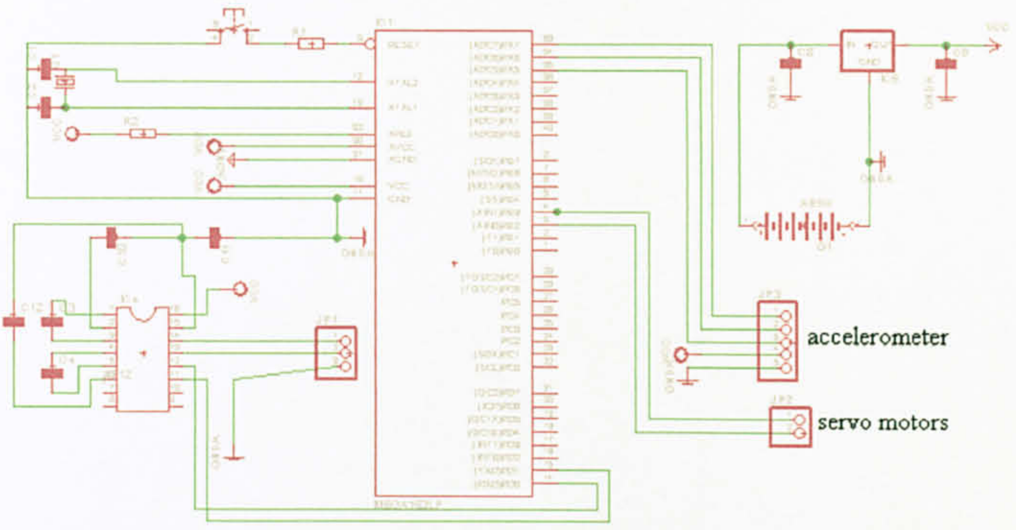
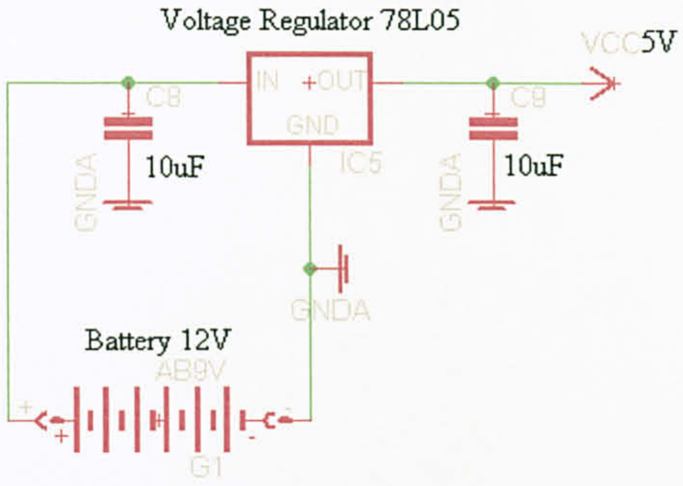
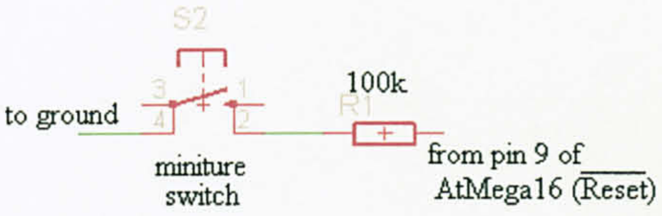


Figure 22: Schematic for circuitry

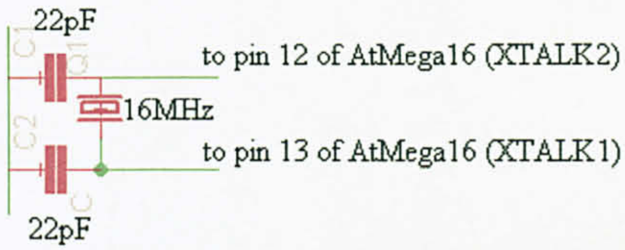
The purpose of power circuit, as shown in figure 23 (a), is to regulate the battery's voltage from 12V to 5V as appropriate for this circuit. The purpose of reset circuit, as shown in figure 23 (b), is to reset the operation of AtMega16. The purpose of the clock circuit, as shown in figure 23 (c), is to provide clock cycle for the AtMega16, which also meant to have the AtMega16 to operate faster. Note that, connecting to high frequency of crystal clock may not assure that the microcontroller chip to work as desired, in a worse case may damage the chip. Therefore, a proper frequency of crystal clock has to be ascertained depending on the type of microcontroller chip. Accelerometer pinouts are for the Memsic2125 connections, as shown in figure 23 (d), and servo motors pinouts are for servo motors connections, as shown in figure 23 (e). The purpose to have in-circuit programming circuit, as shown in figure 23 (f), is to eliminate the need to detach and reattach AtMega16 for programming. This as well would eliminate the possibility of damaging the pins of AtMega16 also ease the process.


Voltage Regulator 78L05
IN
+OUT
GND
IC5
VCC5V
C8
10uF
GNDA
C8
10uF
GNDA
GNDA
Battery 12V
AB9V
G1

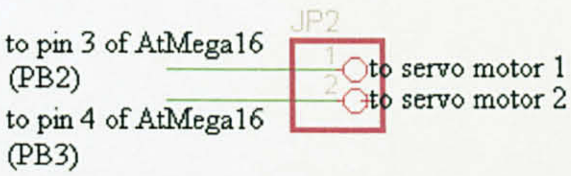
(a)


S2
to ground
3
4
1
2
miniture switch
100k
R1
from pin 9 of AtMega16 (Reset)

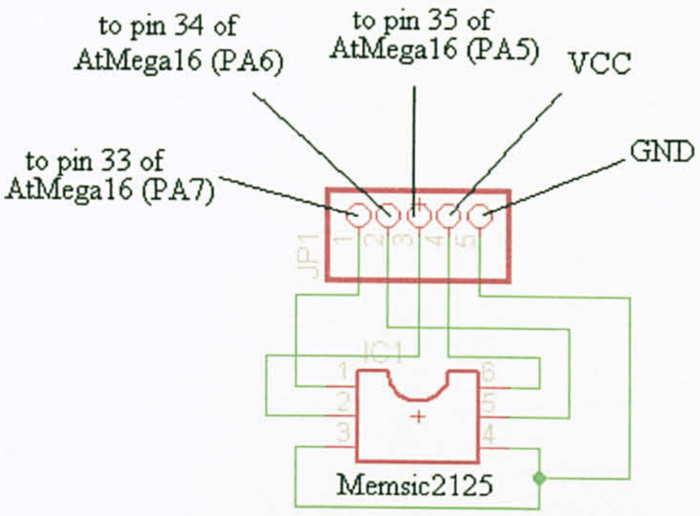
(b)


22pF
C1
Q1
to pin 12 of AtMega16 (XTALK2)
16MHz
C2
to pin 13 of AtMega16 (XTALK1)
22pF

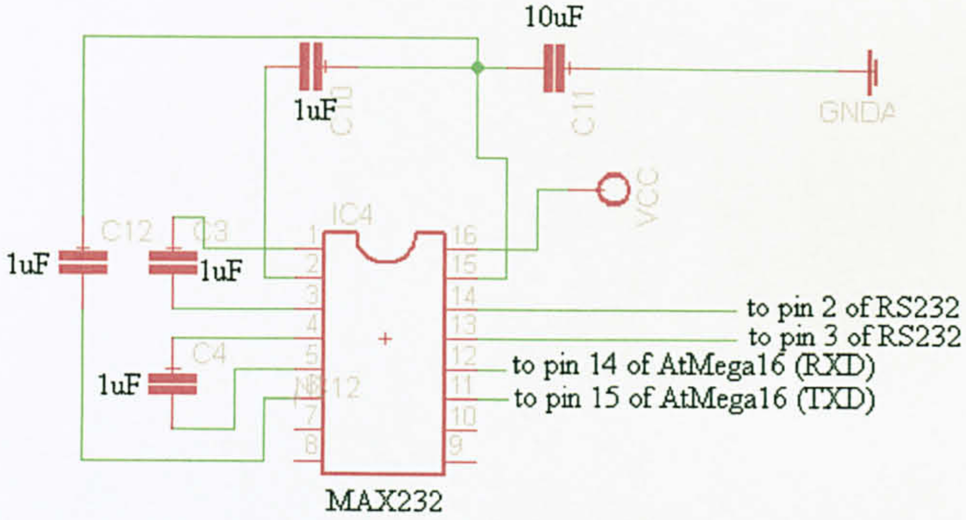
(c)


to pin 3 of AtMega16 (PB2)
to pin 4 of AtMega16 (PB3)
JP2
1
2
to servo motor 1
to servo motor 2

(d)


to pin 34 of AtMega16 (PA6)
to pin 35 of AtMega16 (PA5)
VCC
GND
to pin 33 of AtMega16 (PA7)
JP1
1
2
3
4
5
IC1
Memsic2125

(e)



(f)

Figure 23: Main circuit for AtMega16 Development Circuit: a) power circuit, b) reset circuit, c) clock circuit, d) connection for servo motors, e) connection for Memsic2125 and f) in-circuit programming circuit

The circuit design is then transferred to PCB (printed circuit board) design. The design for the PCB is as shown in figure 24. Note that, the software used for designing the PCB is EAGLE Layout Editor 5.2.1. This design is then fabricated and all the components are soldered unto it.

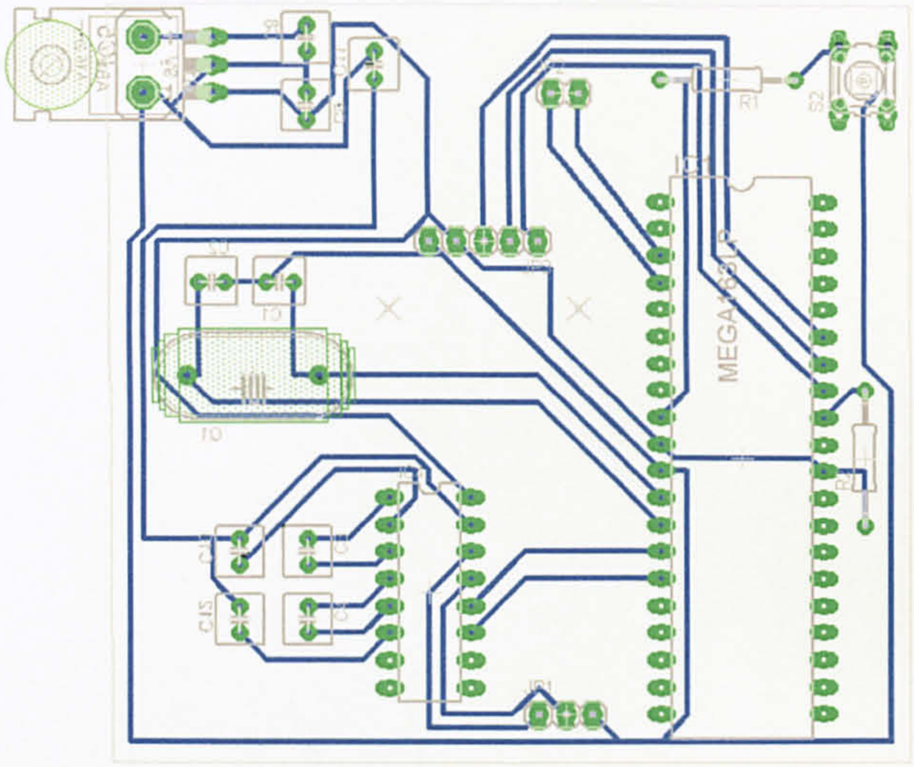


Figure 24: PCB layout for circuitry

3.5 Tools Required

3.5.1 Hardware

Hardware required is as listed in table 1 below, together with their quantity and costs.

Table 1: Cost Items

No.	Item	Quantity	Total Price (RM)
1	Atmel Microcontroller, ATMEGA16	1	35.00
2	MAX232	1	4.00
3	RC Servo Motor	2	120.00
4	Memsic 2125 Dual-axis Accelerometer	1	260.00
5	Sports Tire Set, 70111	1 set	31.00
6	Perspects	3 pieces	30.00
7	Brass M-F Threaded Hex Spacer 40mm	8	*Taken from EE store*
8	Capacitor 0.1uF	4	*Taken from EE store*
9	Capacitor 22pF	2	*Taken from EE store*
10	Capacitor 10uF	3	*Taken from EE store*
11	Capacitor 1uF	4	*Taken from EE store*
12	Resistor 100kΩ	1	*Taken from EE store*
13	Push Button	1	*Taken from EE store*
14	Crystal Clock 16MHz	1	*Taken from EE store*
15	Single Core Wire		*Taken from EE store*
16	Voltage Regulator 74L05	1	*Taken from EE store*
17	Battery 12V	1	9.90
18	Battery 6V	2	17.00
TOTAL			**506.90**

3.5.2 Software

For programming the Atmega16 Microcontroller Chip is by using WinAVR incorporated with AVR Studio 4. The language used for programming coding is C language. The software used to program the Atmega16 Microcontroller Chip is unfamiliar, yet through thorough literature review, the software is explored. The language used is familiar hence eased the writing the programming codes. Note that, all the software used is free-software.

CHAPTER 4
RESULT AND DISCUSSION

4.1 Mechanical and Electrical parts of Self-Balancing Robot assembled

Hence, the self-balancing robot is built. Figure 25 shows the balancing robot for this project and figure 26 shows the top, side and bottom view of the robot.

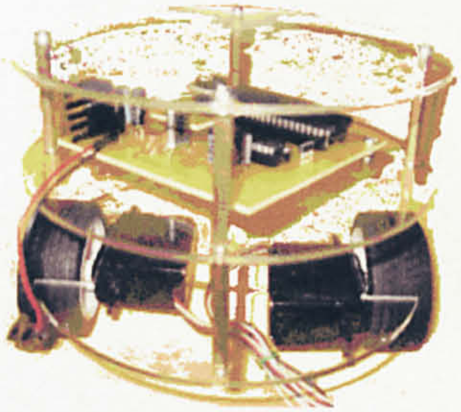
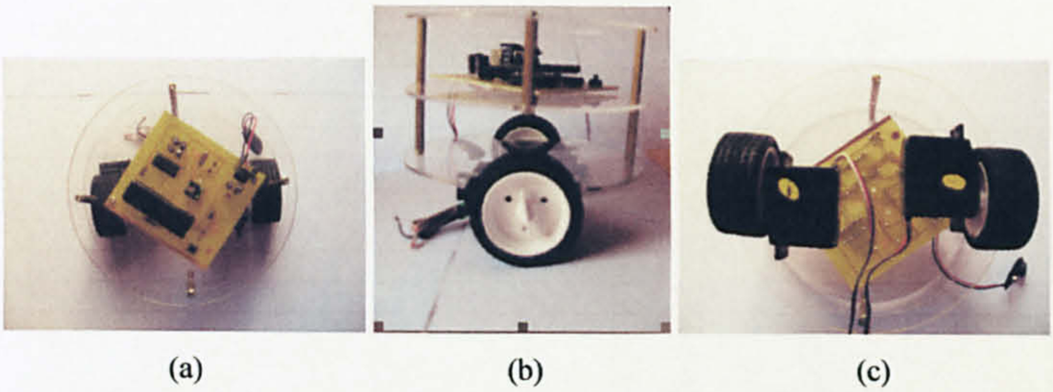


Figure 25: Self-Balancing Robot



(a) (b) (c)

Figure 26: self-balancing robot: a) top view, b) side view and c) bottom view

Following are the disassembled self-balancing robot. Note that, figure 27 (a) is the top platform of the robot which as functions as carrier platform where the object to be carried to be placed on. Next, figure 27 (b) is the second platform from below which also is the circuitry platform. This is where the main board of circuitry is screwed on. Then, figure 27 (c) is the first platform from below which also is the base platform of the robot. This is where the servo motors and wheels are attached at. Lastly, figure 27 (d) is the 33mm spacers, nuts and screws used to attach all the three platforms all together and arranged them vertically.

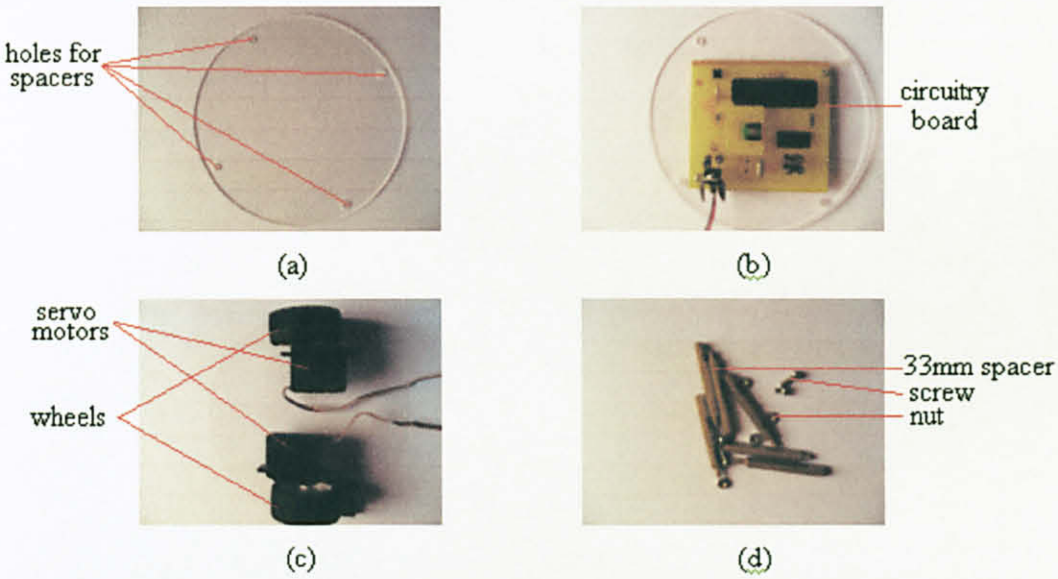


Figure 27: Platforms: a) top platform, b) circuitry platform, c) base platform and d) 33mm spacers, nuts and screws

Figure 28 shows the circuitry platform together with the circuitry board and its electronic components. All these components are carefully soldered unto it using soldering iron and soldering lead not to damage the PCB board. The components soldered are as designed. Testing is then carried out to ensure that the circuit is working properly and no components are damaged during the soldering process. This testing process mainly involved with ensuring all the connection is correct, current supplied at the precise amperes and no other contaminants to the circuit.

The soldering process is quite difficult due to unfamiliarity. However, the circuit is managed be assembled as designed as shown.

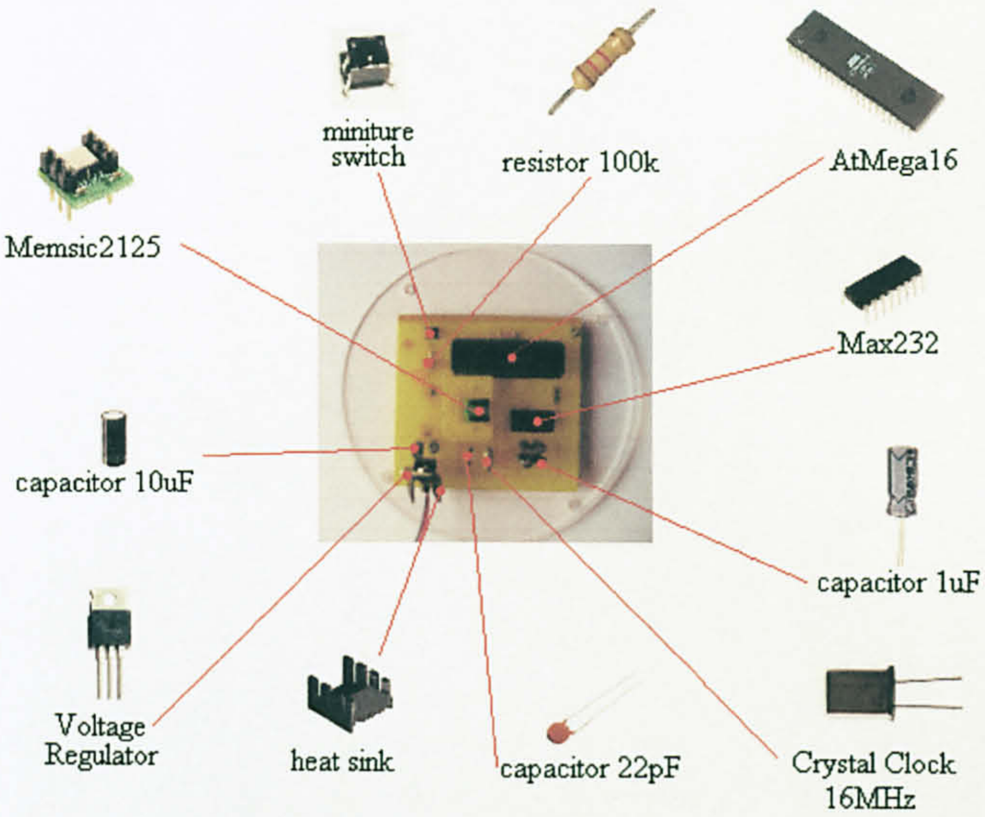


Figure 28: Circuitry platform with circuitry board and its electronic components

Figure 29 shows the base platform of the robot together with servo motors and wheels. The servo motors used is bought from Bizchip and the wheels used are Tamiya Sport tires. After the servo motor is connected to wheel, testing is carried out to ensure that the wheels have no problem in rotating.

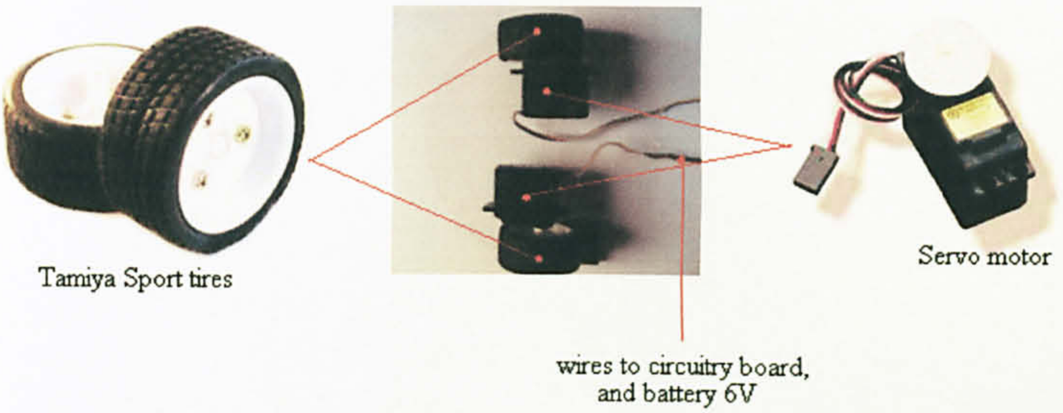


Figure 29: Base platform with servo motor and wheels

Note that, in order to have the servo motors to rotate continuously, modification to the shaft has to be done. During this process, the modification has to be done carefully or else the servo motor would not work properly. The potentiometer of the servo needed to be adjusted so that the rotation is not limited to 180 degrees. The first attempt was a failure since the servo motor is damaged and cannot be used. Hence, another servo motor needed to replace the damaged servo motors. Luckily, the second attempt is successful.

Also note that, one 6V battery is supplied voltage to each servo motor. Hence, there are two 6V battery voltage are used for this robot.

CHAPTER 5
CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

Self-balancing robot is a developing idea towards providing the society an object carrier that is able to balance itself and carry object from one point to the other efficiently. This object carrier will better replace the current technology of object carrier as it can move on either flat or sloppy floor without losing its balance. In addition, object carrier consumes smaller space and suitable for indoor purpose. With further enhances, self-balancing robot could carry more tasks, not just an object carrier.

In general, this project is divided into five major stages. Those major stages are literature review, designing, fabrication, assembling and programming. In the end, the outcome of the project should fulfill all the objectives as stated.

There several problems encountered towards completing the project. Most of problems took place during the fabrication and programming stages. Some hardware got broken and damaged and needed replacement. Luckily the hardware is easy to obtain. Besides that, a few of software which are required to use for fabrication and programming, which are EAGLE Layout Editor 5.2.1 and software to write program for AtMega16, are unfamiliar and there are less resource for direct learning. Therefore, internet research on how to use the software is done also together with referring to other individual for understanding. Limited resource has delayed the progress of the project significantly. However, all these problems are overcome.

5.2 Recommendations

There is no finish limit for self-balancing robot as it is can always be further enhances. Utilizing other stability sensor would improve the balancing system of the robot. For example, utilize gyroscope to provide information on the position of the body. Besides that, add more suitable sensors for better application. For example, add IR sensor for better maneuverability. Other than that, adding more other electronic components to add more function to it. For example, add on suitable camera to add on surveillance function. Moreover, specify its features to suit carrying specific task. For example, equip the physical features with suitable platform for carrying specific object. Furthermore, modify the whole design and enhance the balancing mechanism then it can definitely solve the many problems arising. The great example for this is Segway H.T. As a matter of fact, the enhancement of self-balancing robot could solve the many problems arising by human race.

REFERENCES

[1] Andrew K. Stimac, 1999, *Standup and Stabilization of the Inverted Pendulum*, Final Year Thesis, Department of Mechanical Engineering.

[2] Eugene Dao and Stephen Wong, *Balance Bot (B2)*, ECE4760 Final Project, Cornell University, New York.

[3] Rich Chi Ooi, 2003, *Balancing a Two-Wheeled Autonomous Robot*, Final Year Thesis, The Univerisity of Western Australia, Australia.

[4] Alessio Salerno et all, *The Embodiment Design of a Two-Wheeled Self-Balancing Robot*, Department of Mechanical Engineering & Centre for Intelligent Machine McGill University, Canada.

[5] Billur Barshan et all, 1994, *Evaluation of a Solid-State Gyroscope for Robotics Applications*, IEEE Transactions on Instrumentation and Measurement, vol. 44, no.1.

[6] Felix Grasser et all, *JOE: A Mobile, Inverted Pendulum*, Laboratory of Industrial Electronics Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland.

[7] Kamarudin B. Shehabuddeen and Fakhruldin B. Mohd Hashim, *Development of a Vertial Self-balancing Experimental Autonomous Underwater Vehicle*, Universiti Teknologi PETRONAS, Malaysia.

[8] Jwu-Sheng Hu et all 2009, "Self-balancing Control and Manipulation of a Glove Puppet Robot on a Two-wheel Mobile Platform", *International Conference on Intelligent Robots and Systems*, St. Louis, USA.

[9] Richard Kennaway, 2004, *Inverted Pendulum* <http://www.livingcontrolsystems.com/demos/inverted_pendulum/inverted_pendulum.pdf>.

[10] Johnny Lam, *Control of an Inverted Pendulum*, <http://www.ece.ucsb.edu/~roy/student_projects/Johnny_Lam_report_238.pdf>.

[11] Introduction to Servo Motor Programming, Chapter 11.

[12] Owen Bishop, 2007, *Robot Builder's Cookbook, Build And Design Your Own Robots*, Burlington, MA, Elsivier Ltd.

[13] *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash*, ATMEL, 2009.

[14] Memsic2125 Accelerometer Demo Kit (#28017), Acceleration, Tilt, and Rotation Measurement, Parallax, 2004.

[15] Bizchip Servo Motor Datasheet, <http://wwwbizchip-components.com/datasheet/2405/bizchip_servo_datasheet.pdf>.

[16] David P. Anderson, 2010, <http://www.geology.smu.edu/~dpa-www/robo/nbot/>,

[17] Dimension Engineering, <http://www.dimensionengineering.com/accelerometers.htm>.

[18] Rhett Allain, 2009, <http://scienceblogs.com/dotphysics/2009/05/balancing-sticks-choose- a-longer-stick.php>.

[19] Segway Inc, <http://www.segaway.com/>.

[20] Seattle Robotics Society, <http://www.seattlerobotics.org/guide/servos.html>.

[21] Ted Larson, 2010, <http://www.tedlarson.com/robots/balancingbot.htm>.

[22] Wikipedia.org , <http://en.wikipedia.org/wiki/Microcontroller>

APPENDIX A

GANTT CHART – FYP I

ACTIVITIES	WEEK															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Briefing from Coordinator: Meet Supervisor & finalise topic	20-31 JUL															
Familiarization of topic chose			3-7 AUG													
Submission of Preliminary Report				14 AUG												
Thorough study and Literature Review					17-21 AUG											
Mid-Term						24-28 AUG										
Continued Literature Review							31AUG – 4 SEP									
Submission of Progress Report								11 SEPT								
Design the physical robot and its circuitry									14 SEPT – 16 OCT							
Submission of Draft Report														19 OCT		
Submission of Interim Report															26 OCT	
Oral Presentation																2 – 6 SEPT

GANTT CHART – FYP II

ACTIVITIES	WEEK																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Fabrication and gathering hardware for robot	25 JAN – 12 FEB	▨	▨	▨													
Assembling (mechanical) and testing			1 – 26 FEB	▨	▨												
Submission of Progress Report 1				19 FEB													
Testing and, design and fabrication of PCB					22 FEB – 24 APR	▨	▨	▨	▨	▨	▨	▨	▨				
Mid Term								15- 19 MAR									
Submission of Progress Report 2									26 MAR								
Assembling (electrical) and testing, programming, and correcting errors									22MAR – 24APR	▨	▨	▨	▨				
Submission of Draft Report														28 APR			
Submission of Final Report and Technical Report															5 MAY		
Oral Presentation																7 – 11 JUN	
Submission of Final Report																	25 JUN

tures

- h-performance, Low-power AVR® 8-bit Microcontroller
- vanced RISC Architecture
  - 131 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- h Endurance Non-volatile Memory segments
  - 16K Bytes of In-System Self-programmable Flash program memory
  - 512 Bytes EEPROM
  - 1K Byte Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C[1]
  - Optional Boot Code Section with Independent Lock Bits
  - In-System Programming by On-chip Boot Program
  - True Read-While-Write Operation
  - Programming Lock for Software Security
- G (IEEE std. 1149.1 Compliant) Interface
  - Boundary-scan Capabilities According to the JTAG Standard
  - Extensive On-chip Debug Support
  - Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- ipheral Features
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Four PWM Channels
  - 8-channel, 10-bit ADC
    - 8 Single-ended Channels
    - 7 Differential Channels in TQFP Package Only
    - 2 Differential Channels with Programmable Gain at 1x, 10x, or 200x
  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- cial Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated RC Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby
- nd Packages
  - 32 Programmable I/O Lines
  - 40-pin PDIP, 44-lead TQFP, and 44-pad QFN/MLF
- erating Voltages
  - 2.7 - 5.5V for ATmega16L
  - 4.5 - 5.5V for ATmega16
- ed Grades
  - 0 - 8 MHz for ATmega16L
  - 0 - 16 MHz for ATmega16
- wer Consumption @ 1 MHz, 3V, and 25 C for ATmega16L
  - Active: 1.1 mA
  - Idle Mode: 0.35 mA
  - Power-down Mode: < 1 µA

8-bit AVR® Microcontroller with 16K Bytes In-System Programmable Flash

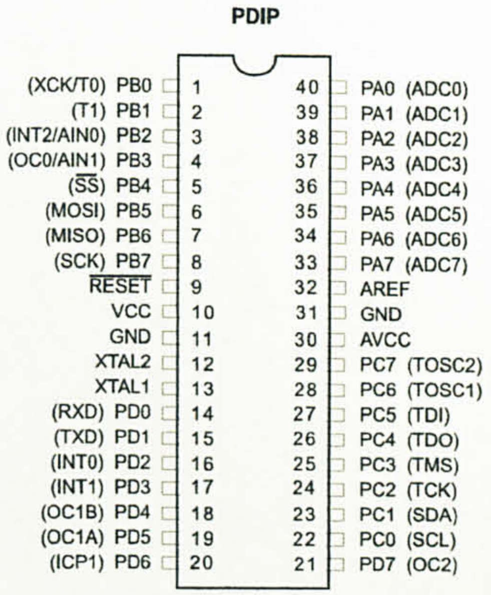
ATmega16
ATmega16L

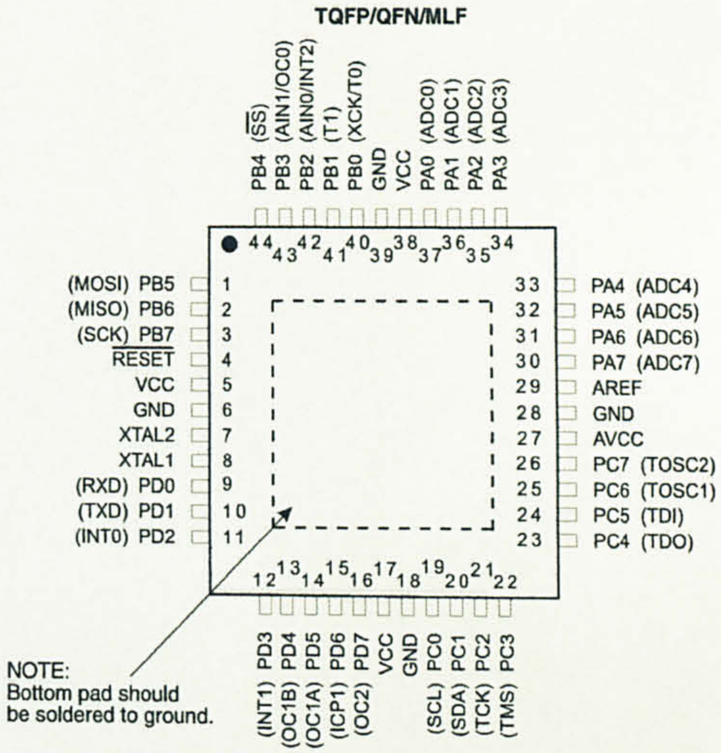
Note: Not recommended for new designs.

Rev. 2466S–AVR–05/09

n
onfigurations

Figure 1. Pinout ATmega16





sclaimer

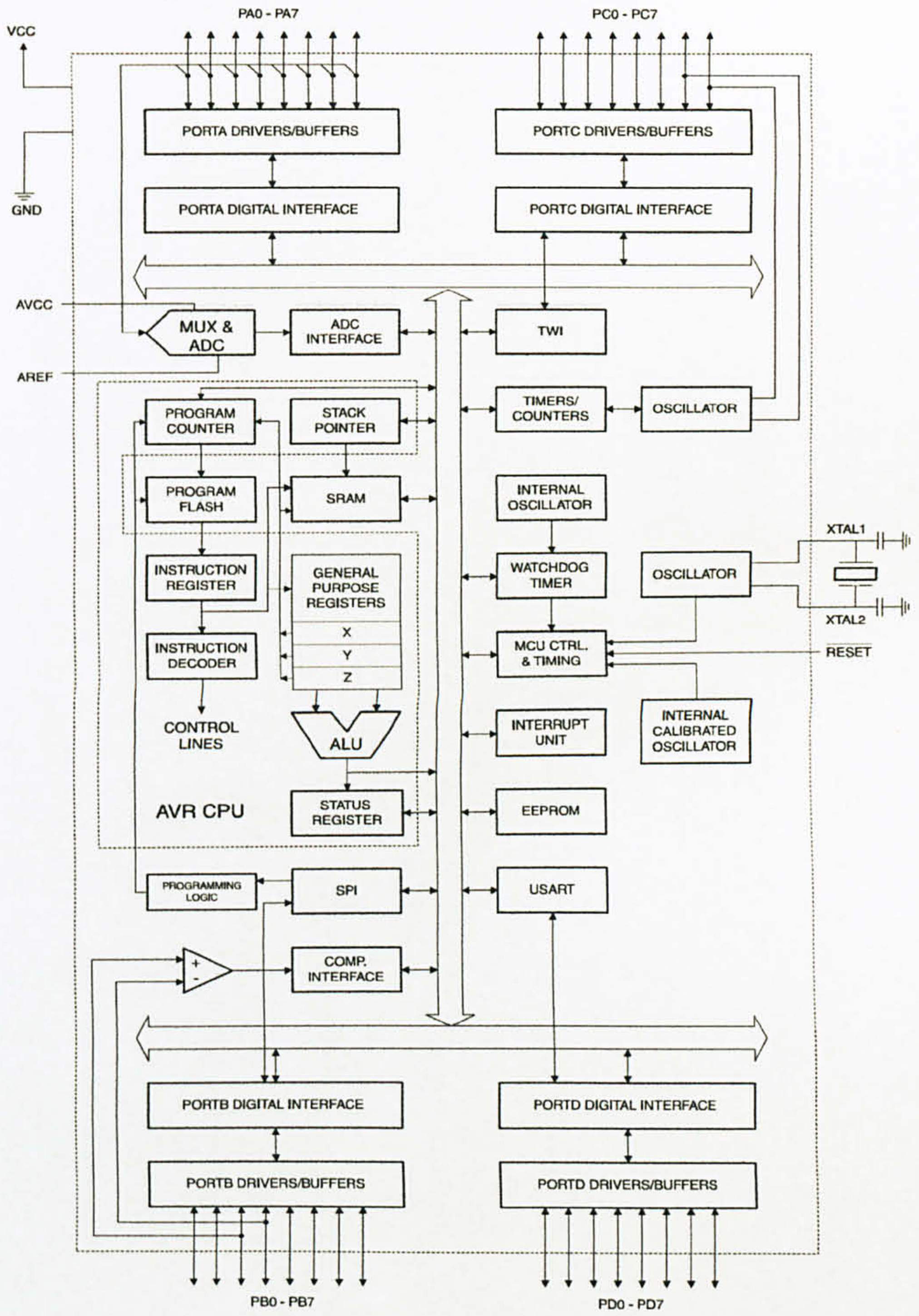
Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Min and Max values will be available after the device is characterized.

erview

The ATmega16 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega16 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

ck Diagram

Figure 2. Block Diagram



rt B (PB7..PB0)

Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port B also serves the functions of various special features of the ATmega16 as listed on page 58.

rt C (PC7..PC0)

Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resistors on pins PC5(TDI), PC3(TMS) and PC2(TCK) will be activated even if a reset occurs.

Port C also serves the functions of the JTAG interface and other special features of the ATmega16 as listed on page 61.

rt D (PD7..PD0)

Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port D also serves the functions of various special features of the ATmega16 as listed on page 63.

$\overline{\text{ESET}}$

Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 38. Shorter pulses are not guaranteed to generate a reset.

AL1

Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

AL2

Output from the inverting Oscillator amplifier.

CC

AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to V_{CC}, even if the ADC is not used. If the ADC is used, it should be connected to V_{CC} through a low-pass filter.

EF

AREF is the analog reference pin for the A/D Converter.

PARALLAX INC

599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office: (916) 624-8333
Fax: (916) 624-8003

General: info@parallax.com
Technical: support@parallax.com
Web Site: www.parallax.com
Educational: www.stampsinclass.com

Memsic 2125 Accelerometer Demo Kit (#28017)
Acceleration, Tilt, and Rotation Measurement

Introduction

The Memsic 2125 is a low cost, dual-axis thermal accelerometer capable of measuring dynamic acceleration (vibration) and static acceleration (gravity) with a range of ±2 g. For integration into existing applications, the Memsic 2125 is electrically compatible with other popular accelerometers.

What kind of things can be done with the Memsic 2125 accelerometer? While there are many possibilities, here's a small list of ideas that can be realized with a Memsic 2125 and the Parallax BASIC Stamp® microcontroller:

- Dual-axis tilt sensing for autonomous robotics applications (BOE-Bot, Toddler, SumoBot)
- Single-axis rotational position sensing
- Movement/Lack-of-movement sensing for alarm systems

Packing List

Verify that your Memsic 2125 Demo Kit is complete in accordance with the list below:

- Parallax Memsic 2125 Demo PCB (uses Memsic MXD2125GL)
- Documentation

Note: Demonstration software files may be downloaded from www.parallax.com.

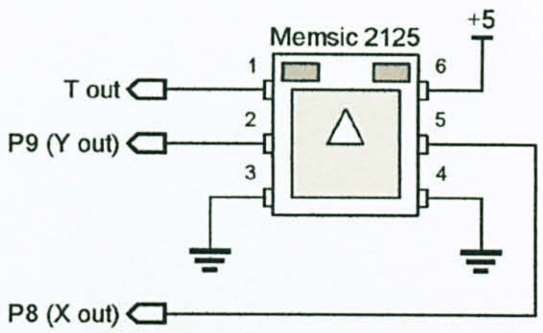
Features

- Measure 0 to ±2 g on either axis; less than 1 mg resolution
- Fully temperature compensated over 0° to 70° C range
- Simple, pulse output of g-force for X and Y axis – direct connection to BASIC Stamp
- Analog output of temperature (TOut pin)
- Low current operation: less than 4 mA at 5 vdc

Connections

Connecting the Memsic 2125 to the BASIC Stamp is a straightforward operation, requiring just two IO pins. If single-axis tilt of less than 60 degrees is your requirement, only one output from the Memsic 2125 need be connected. See Figure 1 for connection details.

Figure 1. Essential Memsic 2125 Connections

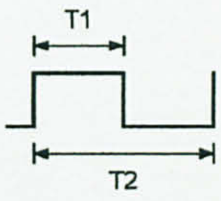


How It Works

Internally, the Memsic 2125 contains a small heater. This heater warms a "bubble" of air within the device. When gravitational forces act on this bubble it moves. This movement is detected by very sensitive thermopiles (temperature sensors) and the onboard electronics convert the bubble position [relative to g-forces] into pulse outputs for the X and Y axis.

The pulse outputs from the Memsic 2125 are set to a 50% duty cycle at 0 g. The duty cycle changes in proportion to acceleration and can be directly measured by the BASIC Stamp. Figure 2 shows the duty cycle output from the Memsic 2125 and the formula for calculating g force.

Figure 2. Memsic 2125 Pulse Output



$$A(g) = ((T1 / T2) - 0.5) / 12.5\%$$

The T2 duration is calibrated to 10 milliseconds at 25° C (room temperature). Knowing this, we can convert the formula to the following BASIC Stamp routine:

```
Read_X_Force:
  PULSIN Xin, HiPulse, xRaw
  xRaw = xRaw */ Scale
  xGForce = ((xRaw / 10) - 500) * 8
  RETURN
```

The T1 duration (Memsic output) is captured by PULSIN in the variable *xRaw*. Since each BASIC Stamp module has its own speed and will return a different raw value for the pulse, the factor called *Scale* (set by the compiler based on the BASIC Stamp module installed) is used to convert the raw output to microseconds. This will allow the program to operate properly with any BASIC Stamp 2-series module. At this point the standard equation provided by Memsic can be applied, adjusting the values to account for the pulse-width in microseconds. Fortunately, one divided by divided by 0.125 (12.5%) is eight, hence the final multiplication. The result is a signed value representing g-force in milli-g's (1/1000th g).

Experiments

Experiment 1: Dual-Axis Tilt Measurement

This experiment reads both axis values and displays the results in the DEBUG window. Calculations for g-force measurement and conversion to tilt were taken directly from Memsic documentation. Since the BASIC Stamp does not have an Arcsine function, it must be derived. Code for Arccosine and Arcsine are provided courtesy Tracy Allen, Ph.D.

```
' =========================================================================
'
'   File...... MEMSIC2125-Dual.BS2
'   Purpose... Memsic 2125 Accelerometer Dual-Axis Demo
'   Author.... (C) 2003-2004 Parallax, Inc -- All Rights Reserved
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 07 SEP 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Read the pulse outputs from a Memsic 2125 accelerometer and converts to
' G-force and tilt angle.
'
' g = ((t1 / 10 ms) - 0.5) / 12.5%
'
' Tilt = ARCSIN(g)
'
' Refer to Memsic documentation (AN-00MX-007.PDF) for details on g-to-tilt
' conversion and considerations.
'
' www.memsic.com


' -----[ Revision History ]------------------------------------------------


' -----[ I/O Definitions ]-------------------------------------------------

Xin             PIN     8                       ' X input from Memsic 2125
Yin             PIN     9                       ' Y input from Memsic 2125


' -----[ Constants ]-------------------------------------------------------

' Set scale factor for PULSIN

#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON     $200                    ' 2.0 us per unit
  #CASE BS2SX
    Scale       CON     $0CC                    ' 0.8 us per unit
  #CASE BS2P
```

```
    Scale         CON      $0C0                          ' 0.75 us per unit
  #CASE BS2PE
    Scale         CON      $1E1                          ' 1.88 us per unit
#ENDSELECT

HiPulse           CON      1                             ' measure high-going pulse
LoPulse           CON      0

DegSym            CON      176                           ' degrees symbol


' -----[ Variables ]-------------------------------------------------------

xRaw              VAR      Word                          ' pulse from Memsic 2125
xmG               VAR      Word                          ' g force (1000ths)
xTilt             VAR      Word                          ' tilt angle

yRaw              VAR      Word
ymG               VAR      Word
yTilt             VAR      Word

disp              VAR      Byte                          ' displacement (0.0 - 0.99)
angle             VAR      Byte                          ' tilt angle


' -----[ EEPROM Data ]-----------------------------------------------------


' -----[ Initialization ]--------------------------------------------------

Setup:
  PAUSE 250                                              ' let DEBUG window open
  DEBUG "Memsic 2125 Accelerometer", CR,
        "-------------------------"


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Read_Tilt                                      ' reads G-force and Tilt

    ' display results

    DEBUG CRSRXY, 0, 3
    DEBUG "X Input...  ",
          DEC (xRaw / 1000), ".", DEC3 xRaw, " ms",
          CLREOL, CR,
          "G Force... ", (xmG.BIT15 * 13 + " "),
          DEC (ABS xmG / 1000), ".", DEC3 (ABS xmG), " g",
          CLREOL, CR,
          "X Tilt.... ", (xTilt.BIT15 * 13 + " "),
          DEC ABS xTilt, DegSym, CLREOL

    DEBUG CRSRXY, 0, 7
    DEBUG "Y Input...  ",
          DEC (yRaw / 1000), ".", DEC3 yRaw, " ms",
          CLREOL, CR,
          "G Force... ", (ymG.BIT15 * 13 + " "),
          DEC (ABS ymG / 1000), ".", DEC3 (ABS ymG), " g",
```

XX

```
        CLREOL, CR,
        "Y Tilt.... ", (yTilt.BIT15 * 13 + " "),
        DEC ABS yTilt, DegSym, CLREOL

    PAUSE 200                                   ' update about 5x/second
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

Read_G_Force:
  PULSIN Xin, HiPulse, xRaw                     ' read pulse output
  xRaw = xRaw */ Scale                          ' convert to uSecs
  xmG = ((xRaw / 10) - 500) * 8                 ' calc 1/1000 g
  PULSIN Yin, HiPulse, yRaw
  yRaw = yRaw */ Scale
  ymG = ((yRaw / 10) - 500) * 8
  RETURN


Read_Tilt:
  GOSUB Read_G_Force
  disp = ABS xmG / 10 MAX 99                    ' x displacement
  GOSUB Arcsine
  xTilt = angle * (-2 * xmG.BIT15 + 1)          ' fix sign
  disp = ABS ymG / 10 MAX 99                    ' y displacement
  GOSUB Arcsine
  yTilt = angle * (-2 * ymG.BIT15 + 1)          ' fix sign
  RETURN


' Trig routines courtesy Tracy Allen, PhD. (www.emesystems.com)

Arccosine:
  disp = disp */ 983 / 3                        ' normalize input to 127
  angle = 63 - (disp / 2)                       ' approximate angle
  DO                                            ' find angle
    IF (COS angle <= disp) THEN EXIT
    angle = angle + 1
  LOOP
  angle = angle */ 360                          ' convert brads to degrees
  RETURN


Arcsine:
  GOSUB Arccosine
  angle = 90 - angle
  RETURN
```

Experiment 2: Rotational Position Sensing

If the Memsic 2125 is tilted up on its edge (X axis), the X and Y outputs can be combined to measure rotational position. Output from this program is in Brads (binary radians, 0 to 255, the BASIC Stamp's unit of angular measurement) and degrees (0 to 359).

For this code to work, the Memsic 2125 PCB must be positioned such that the sensor is perpendicular to the ground.

```
' =========================================================================
'
'   File...... MEMSIC2125-Rotation.BS2
'   Purpose... Memsic 2125 Accelerometer Rotational Angle Measurement
'   Author.... (C) 2003-2004 Parallax, Inc -- All Rights Reserved
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 07 SEP 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Read the pulse outputs from a Memsic 2125 accelerometer and combine to
' calculation rotational position.
'
' Refer to Memsic documentation (AN-00MX-007.PDF) for details on angle
' conversion and considerations.
'
' www.memsic.com


' -----[ I/O Definitions ]-------------------------------------------------

Xin             PIN     8                       ' X input from Memsic 2125
Yin             PIN     9                       ' Y input from Memsic 2125


' -----[ Constants ]-------------------------------------------------------

' Set scale factor for PULSIN

#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON     $200                    ' 2.0 us per unit
  #CASE BS2SX
    Scale       CON     $0CC                    ' 0.8 us per unit
  #CASE BS2P
    Scale       CON     $0C0                    ' 0.75 us per unit
  #CASE BS2PE
    Scale       CON     $1E1                    ' 1.88 us per unit
#ENDSELECT

HiPulse         CON     1                       ' measure high-going pulse
LoPulse         CON     0
```

```
DegSym          CON     176                         ' degrees symbol


' -----[ Variables ]-------------------------------------------------------

pulse           VAR     Word                        ' pulse input
xmG             VAR     Word                        ' g force (1000ths)
ymG             VAR     Word
brads           VAR     Word                        ' binary radians
degrees         VAR     Word


' -----[ Initialization ]--------------------------------------------------

Setup:
  DEBUG "Memsic 2125 Rotation", CR,
        "--------------------"


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Read_G_Force                              ' read X and Y

    brads = (xmG / 8) ATN (ymG / 8)                 ' calculate angle
    degrees = brads */ 360                          ' convert to degrees

    DEBUG CRSRXY, 0, 3
    DEBUG "Axis   A(g)", CR,
          "X      ", (xmG.BIT15 * 13 + " "),
          DEC (ABS xmG / 1000), ".", DEC3 (ABS xmG), " g", CR,
          "Y      ", (ymG.BIT15 * 13 + " "),
          DEC (ABS ymG / 1000), ".", DEC3 (ABS ymG), " g", CR, CR,
          "Tilt = ", DEC3 brads, " Brads", CR,
          "       ", DEC3 degrees, " Degrees"

    PAUSE 200                                       ' update about 5x/second
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

Read_G_Force:
  PULSIN Xin, HiPulse, pulse                        ' read pulse output
  pulse = pulse */ Scale                            ' convert to uSecs
  xmG = ((pulse / 10) - 500) * 8                    ' calc 1/1000 g
  PULSIN Yin, HiPulse, pulse
  pulse = pulse */ Scale
  ymG = ((pulse / 10) - 500) * 8
  RETURN
```

Experiment 3: Motion Detector

This experiment uses the Memsic 2125 as a movement or vibration detector. The program starts by reading the initial state of the sensor and storing these readings as calibration values. By doing this, the starting position of the sensor is nullified. The main loop of the program reads the sensor and compares the current outputs to the calibration values. If the output from either axis is greater than its calibration value the motion timer is incremented. If both fall below the thresholds motion timer is cleared. If the motion timer exceeds its threshold, the alarm will be turned on and will stay on until the BASIC Stamp is reset.

You can adjust the sensitivity (to motion/vibration) of the program by changing the **XLimit** and **YLimit** constants, as well as the **SampleDelay** constant (should be 100 ms or greater). The **AlarmLevel** constant determines how long motion/vibration must be present before triggering the alarm.

```
' =========================================================================
'
'   File...... MEMSIC2125-Motion.BS2
'   Purpose... Detects continuous motion for given period
'   Author.... Parallax (based on code by A. Chaturvedi of Memsic)
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 15 JAN 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Monitors X and Y inputs from Memsic 2125 and will trigger alarm if
' continuous motion is detected beyond the threshold period.


' -----[ I/O Definitions ]-------------------------------------------------

Xin             PIN     8                       ' X pulse input
Yin             PIN     9                       ' Y pulse input
ResetLED        PIN     10                      ' reset LED
AlarmLED        PIN     11                      ' alarm LED


' -----[ Constants ]-------------------------------------------------------

HiPulse         CON     1                       ' measure high-going pulse
LoPulse         CON     0

SampleDelay     CON     500                     ' 0.5 sec
AlarmLevel      CON     5                       ' 5 x SampleDelay

XLimit          CON     5                       ' x motion max
YLimit          CON     5                       ' y motion max


' -----[ Variables ]-------------------------------------------------------
```

```
xCal            VAR     Word                    ' x calibration value
yCal            VAR     Word                    ' y calibration value
xMove           VAR     Word                    ' x sample
yMove           VAR     Word                    ' y sample
xDiff           VAR     Word                    ' x axis difference
yDiff           VAR     Word                    ' y axis difference

moTimer         VAR     Word                    ' motion timer


' -----[ Initialization ]--------------------------------------------------

Initialize:
  LOW AlarmLED                                  ' alarm off
  moTimer = 0                                   ' clear motion timer

Read_Cal_Values:
  PULSIN Xin, HiPulse, xCal                     ' read calibration values
  PULSIN Yin, HiPulse, yCal
  xCal = xCal / 10                              ' filter for noise & temp
  yCal = yCal / 10

  HIGH ResetLED                                 ' show reset complete
  PAUSE 1000
  LOW ResetLED


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Get_Data                              ' read inputs
    xDiff = ABS (xMove - xCal)                  ' check for motion
    yDiff = ABS (yMove - yCal)

    IF (xDiff > XLimit) OR (yDiff > YLimit) THEN
      moTimer = moTimer + 1                     ' update motion timer
      IF (moTimer > AlarmLevel) THEN Alarm_On
    ELSE
      moTimer = 0                               ' clear motion timer
    ENDIF
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

' Sample and filter inputs

Get_Data:
  PULSIN Xin, HiPulse, xMove                    ' take first reading
  PULSIN Yin, HiPulse, yMove
  xMove = xMove / 10                            ' filter for noise & temp
  yMove = yMove / 10
  PAUSE SampleDelay
  RETURN


' Blink Alarm LED
' -- will run until BASIC Stamp is reset
```

```
Alarm_On:
  DO
    TOGGLE AlarmLED                                       ' blink alarm LED
    PAUSE 250
  LOOP                                                    ' loop until reset
```

Application Idea

Using the tilt code from Experiment 1, you can create a 3D joystick by mounting the Memsic 2125 and a pushbutton in a small, spherical enclosure (like a tennis ball). With just three pins you can measure tilt of each axis and the status of the switch. This would make an interesting, intelligent "leash" for a Parallax BOE-Bot.

Using TOut

Since the Memsic 2125 is a thermal device, the temperature is available from the TOut pin and can be measured using an external analog to digital converter (i.e., LTC1298).

Details:

- Output calibrated to 1.25 volts @ 25.0° C
- Output change: 5 millivolts per degree C

11.8 Pulse Widths – The Servo's Heartbeat

Each servo has a built-in processor that responds to electrical pulses sent to it. The BX-24 creates an electrical pulse by sending voltage to one of its pins for a very specific amount of time. The microcontroller cannot control how *much* voltage is sent – it simply turns the voltage on or off. When the voltage is on, the BX-24 outputs +5V. When the voltage is off, 0V is output. The BX-24 can turn the output voltage on and off rapidly, thereby creating **pulses** of *high* and *low* voltages.

The duration of these pulses is known as the **pulse width**. The longer the voltage is applied, the larger the pulse width. The pulse width is measured in seconds, but we often use milliseconds (ms) to describe them because the pulse duration can be very short. Recall from your introductory science classes that 1s is equivalent to 1000ms, therefore 1ms = 0.001s. For example, Figure 11.24 shows a pulse width of 5ms. **In our code, however, we always enter pulse widths in units of seconds**, so we would enter 0.005s in this case.

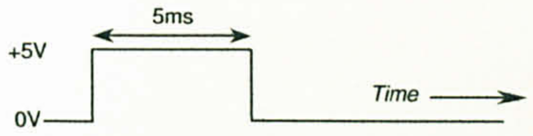


Figure 11.24. A **high** pulse whose pulse width (duration) is 5ms (0.005s).

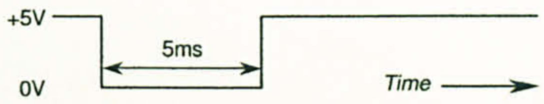


Figure 11.25. A **low** pulse whose pulse width is also 5ms in length.

The BX-24 can send either a *high* pulse (as shown in Figure 11.24) or a *low* pulse (as shown in Figure 11.25). Notice in high pulse mode, the voltage is normally 0V and pulses high when commanded. Conversely, in low pulse mode, the output is normally +5V and goes low when commanded.

With Do-Loops and For-To-Next loops, the BX-24 can produce a series of pulses at a regular rate. This repetitive series of pulses is known as a ***pulse train*** and can be produced by inserting a fixed time delay between the pulses. For example, Figure 11.26 shows a train of pulses, each with a 1ms pulse width, separated by a 5ms delay. The time between successive pulses is known as the ***period***, which, in this case, is 6ms. It is easy to see why this output is often called a *square wave*.

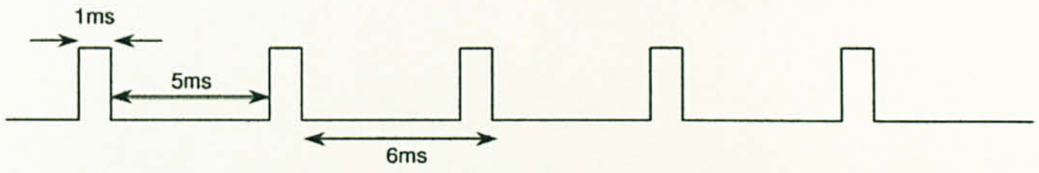


Figure 11.26. A train of 1ms pulses. Since there is a 5ms delay between pulses, the pulse period is 6ms.

Figure 11.27 shows the actual BX-24 output voltage of a train of 1ms (0.001s) pulses with a 20ms (0.02s) delay between pulses. In reality, the pulses are not perfect square waves, as the above figures would lead us to believe.

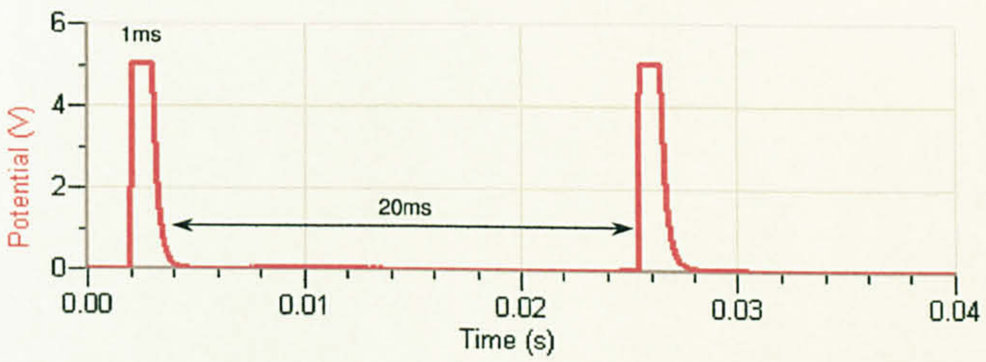


Figure 11.27. A train of 0.001s-pulses with a 0.020s delay between pulses as measured by an oscilloscope.

11.9 Servo Operation

So, how does the BX-24 control the servo? You have probably guessed that it does so with electrical pulses. The pulses are sent from the BX-24 to a control board within the servo itself. The servo's control board interprets the pulses and rotates its shaft either clockwise or counterclockwise based on the pulse widths. Servos will not respond to just any pulse width; rather, they are limited to a well-defined range of pulse widths. Most servos have a minimum pulse width limit around 0.0010s (1.0ms) and a maximum limit around 0.0020s (2.0ms), although the actual minimum and maximum pulse widths will vary slightly between the various servo brands. Sending a pulse whose pulse width is outside this range may **damage** the servo's control board. Repeatedly sending these bad pulses will almost certainly **destroy** the servo.

Pulse Widths = Positions

Servos interpret pulse widths as **positions**. Each position along the arc traced out by the rotating shaft has a corresponding pulse width. When we send a pulse to the servo, the control board calculates which way the shaft should rotate in order to reach the corresponding position. There are two ways that servo manufacturers can wire their servos: positions increasing **clockwise**, and positions increasing **counterclockwise**. The positions and corresponding pulse widths for Futaba and Blue Bird brand servos are shown in Figures 11.28 and 11.29. All servos have a **center position**, sometimes called the *neutral position*. Both servos below are in the **center position** as indicated by the dark pointers. Observe that the Futaba center position corresponds to 1.5ms and the Blue Bird's center is at the 1.8ms-position.[14]

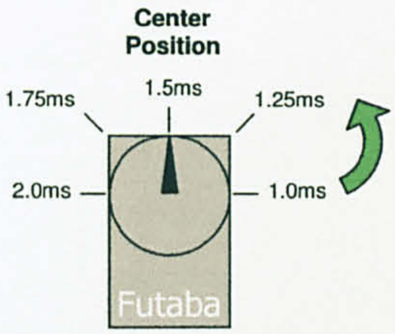


Figure 11.28. The pulse width positions of a **Futaba S3004** servo increase counterclockwise.

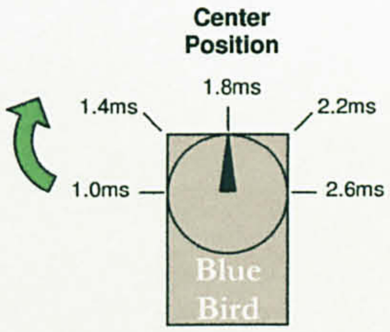


Figure 11.29. The pulse width positions of a **Blue Bird BMS-380** servo increase clockwise.

One popular servo model, Futaba's S3004, rotates counterclockwise with increasing pulse widths, while another servo, the Blue Bird BMS-380, rotates clockwise with increasing pulse widths. This difference is significant to note. If your servos were included as part of Robodyssey's Mouse kit then you probably have the Futaba S3004. Robodyssey does sell Blue Bird servos as well, but I assume that most readers own the Futaba servo, so I will focus my discussion on the S3004. Rest assured, the following discussion is valid for **all servo models** – you simply must keep in mind which servo you are using and program it accordingly. For a detailed comparison of a number of popular servo models, see *Appendix E: Servo Comparisons and Physical Limitations*.

Modified Versus Unmodified Servos

There is another important difference among servo brands: servos can be either **modified** or **unmodified**, and we must know which kind we are dealing with before we can program them. *Unmodified servos* are those that come straight off the hobby shop shelves and have never been "tampered" with. All you radio-control (R/C) hobby-types are certainly familiar with the unmodified servos, for they are used to control your airplanes, cars, or boats. *Modified servos* have been, well, modified – albeit by an expert and with good reason. Modified servos have been altered electrically and mechanically, making them capable of spinning *indefinitely* in either direction. This handy modification means that modified servos can act as drive motors for robots – drive motors that are easily controlled by the BX-24! Both modified and unmodified servos play an important role in robotics, as we are about to find out.

[14] These are approximate values, because center positions can vary greatly from servo to servo.

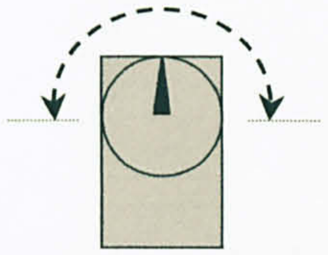


Figure 11.30. An **unmodified** servo can rotate only through 180º.

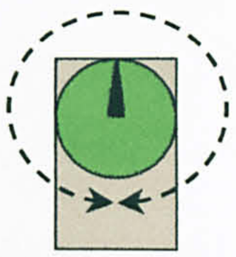


Figure 11.31. A **modified** servo is capable of full rotation in either direction. For our purposes, modified servos are illustrated in green.

You **cannot** tell the difference between modified and unmodified servos simply by looking at them.[15] If you purchased them from a hobby shop then you probably have **unmodified** servos. If you purchased a Robodyssey Mouse kit, the servos included with the robot have been **modified** for continuous rotation. As shown in Figures 11.30 and 11.31, an unmodified servo is designed to rotate only through an arc of roughly 180°, while the modified servo is able to rotate over 360° in either direction. Modified and unmodified servos respond differently to pulse width inputs. Let's look at the response of unmodified servos.

Unmodified Servo Control

When an **unmodified servo** detects a well-defined pulse, the servo shaft will begin rotating toward the position that corresponds to that pulse width. This is shown in Figure 11.32. Notice that the servo can move only a small distance with each pulse, so it may require several pulses to get the servo to actually reach the desired position.[16] If the shaft's current position is sufficiently **near** the desired final position, one pulse may be all that is necessary to move the required distance. If the shaft is already **at** the desired final position, the shaft will not rotate at all. (It is important to note that *the shaft rotates faster when it is far away from the desired position*. This fact will be significant when we cover advanced servo operations in *Chapter 13*!)

Let's look at an example: A Futaba S3004 servo is initially situated at the 1.40ms-position when it receives a train of 1.75ms-pulses, as shown in Figure 11.32. The servo's control board determines that the shaft must rotate **counterclockwise** to reach the 1.75ms-position, and furthermore, it determines that the *initial* rotation speed should be **high** because the 1.40ms-position is far away from the 1.75ms-position. Since the first pulse does not move the shaft to the desired position, the process is repeated and the shaft continues to rotate counterclockwise with **decreasing speed**. After a few pulses, the shaft reaches the desired 1.75ms-position.

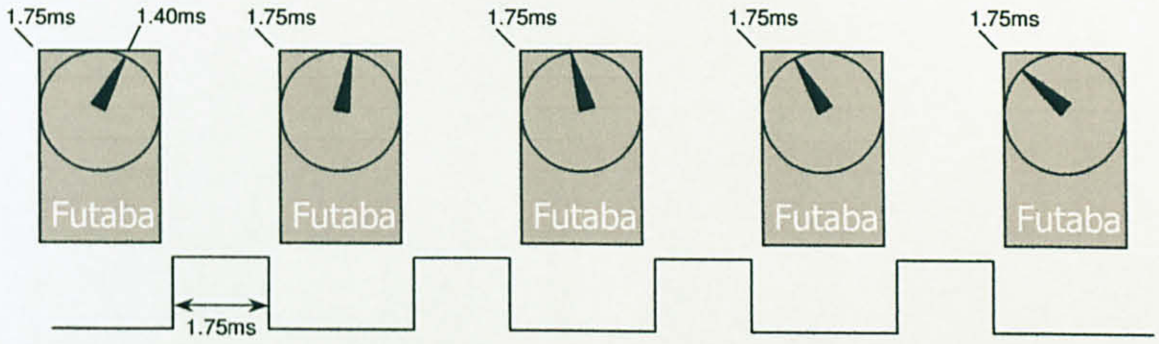


Figure 11.32. This unmodified Futaba S3004 servo is originally at the 1.40ms-position when it receives a train of 1.75ms-pulses. The servo's control board interprets the pulses and causes the shaft to rotate (counterclockwise) toward the 1.75ms-position. After a few pulses, the shaft is in the desired 1.75ms-position.

[15] In my figures, you **can** tell which are modified servos simply by looking because I've given them green shafts.
[16] I have found that a train of 17 individual pulses is sufficient to rotate *my* servos 180º; servo models are slightly different and may require more or fewer pulses to get the job done.

If additional 1.75ms-pulses are sent to the servo, they are ignored since the shaft is already in the 1.75ms-position. This is shown in Figure 11.33, below.

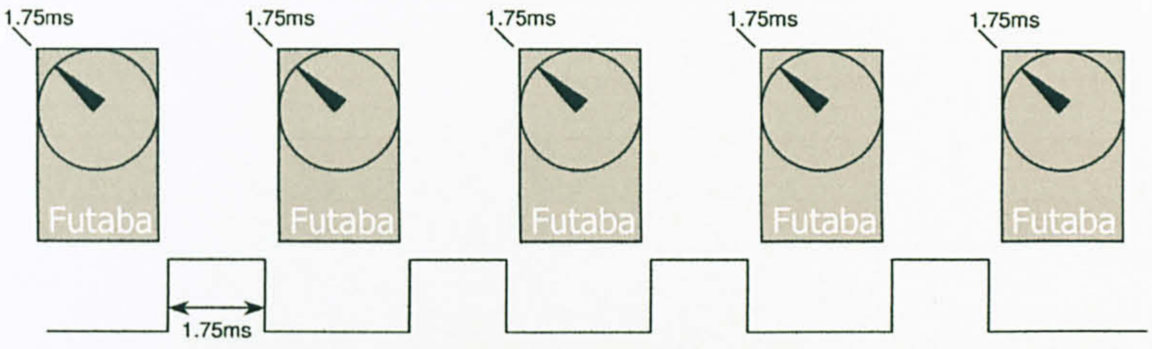


Figure 11.33. Additional 1.75ms-pulses are sent to the unmodified servo but the control board ignores them since the servo is already at the 1.75ms-position.

Figure 11.34 below shows another train of 1.75ms-pulses being sent to the Futaba servo, but this time the shaft will rotate **clockwise** since the desired 1.75ms-position is to the **right** of the original 2.0ms-position. In addition, the shaft will spin at a relatively **slow speed** since the 1.75ms-position is near the 2.0ms-position.

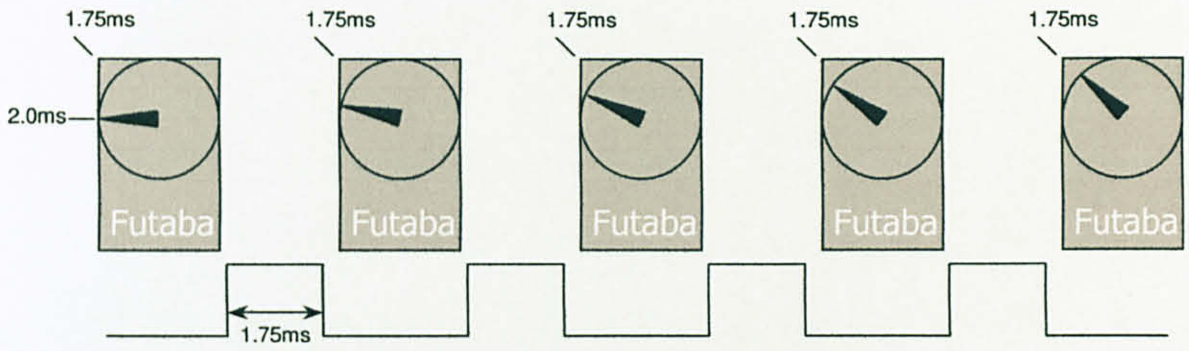


Figure 11.34. In this example, the unmodified Futaba servo rotates clockwise to reach the 1.75ms-mark.

Follow these general rules when programming a **Futaba S3004 unmodified servo**:

- A train of pulses whose pulse widths are approximately **1.0ms** will always rotate the shaft **fully clockwise** to its right-most position.
- A train of pulses whose pulse widths are approximately **1.5ms** will rotate the shaft to its **center position**.
- A train of pulses whose pulse widths are approximately **2.0ms** will always rotate the shaft **fully counterclockwise** to its left-most position.

For more details on the Futaba S3004 and other servos, consult *Appendix E: Servo Comparisons and Physical Limitations* for handy data tables and comparisons.

If you would like to write code for an unmodified servo, I strongly urge you to read *Chapter 13* and *Appendices E and G*. If you are not careful, unmodified servos can be easily damaged.

Modified Servo Control

A **modified servo** behaves exactly like an unmodified servo, with two major exceptions:

1. A modified servo is capable of spinning indefinitely either clockwise or counterclockwise.
2. A modified servo receiving a train of pulses will come to *rest* only if the pulse widths correspond to the servo's *center position*.

Modified servos are able to spin indefinitely because the servo's feedback mechanism has been removed with a handy bit of rewiring. This means the servo never knows the actual position of its shaft. Instead, it is tricked into thinking that the shaft is *always* in the **center position**. Therefore, any pulse whose pulse width does **not** correspond to the servo's center position will cause the servo to rotate either clockwise or counterclockwise. Moreover, the greater the difference between that pulse width and the center position, the faster the shaft will rotate – but that's for another chapter.

If a stream of pulses with **1.0ms** pulse widths is sent to a modified Futaba S3004 servo, the shaft will spin **clockwise** and will continue spinning clockwise until the pulses stop. Here's why: the modified Futaba servo always thinks its shaft is at the 1.5ms center position, so when it receives a pulse of 1.0ms it rotates the shaft clockwise, as shown in Figure 11.35. When the next 1.0ms-pulse is received, the servo still believes it is at the 1.5ms center position and continues to rotate the shaft clockwise. The servo will **never** reach its desired 1.0ms-position, and its shaft will continuously rotate clockwise as long as the stream of 1.0ms pulses is being sent to the servo.

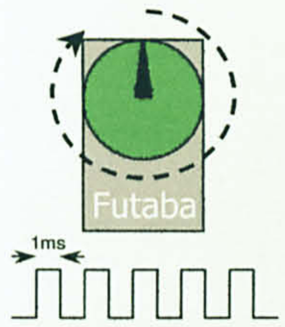


Figure 11.35. A train of **1ms** pulses will rotate a modified servo **clockwise**.

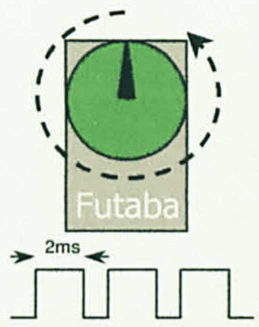


Figure 11.36. A train of **2ms** pulses will rotate a modified servo **counterclockwise**.

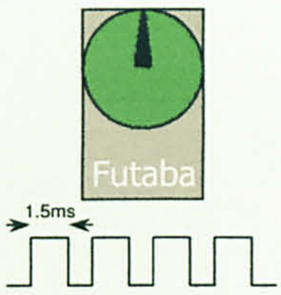


Figure 11.37. A train of **1.5ms** pulses will cause a modified servo to remain stationary in the center.

Now consider a stream of **2.0ms** pulses sent to a modified Futaba servo. In this case, the shaft will spin **counterclockwise** until the pulses cease, as shown in Figure 11.36. So, while an unmodified servo will rotate to a particular position and hold that position, the modified servo is tricked into thinking that the shaft is still at the center position. Roboticists are clever folks, aren't they?

Finally, if a stream of **1.5ms** pulses is sent to a modified Futaba servo (Figure 11.37), the servo will ignore the pulse train because it thinks it is already at the 1.5ms-position.[17] In this case, the servo will **not rotate**.

Follow these general rules when programming a **Futaba S3004** **<u>modified</u>** **servo**:

- Any pulse **less than** approximately **1.5ms** will rotate the shaft **clockwise**.
- A train of pulses whose pulse widths are approximately **1.5ms** will not rotate the servo as the servo always assumes it is at its **center position**, regardless of its actual physical position. (Thus, the 1.5ms pulse train will cause the shaft to remain stationary.)
- Any pulse **greater than** approximately **1.5ms** will rotate the shaft **counterclockwise**.

[17] The 1.5ms center position is an approximation, and could vary by as much as ±0.25ms.

Obviously, the servos we will use to drive our robot must be *modified* servos, since we need them to continually spin the robot's wheels. So, are you ready to take your Mouse for a test drive?

11.10 Let's Do The Hokey-Pokey!

Put Your LEFT Foot In ...

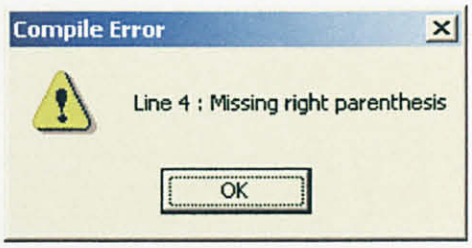
Mouse Move

It's finally time to write some code and get the Mouse moving. Create a project entitled "MouseMove" and place it in a folder named "Mouse Move". Make sure the left and right Futaba servos are properly connected to the RAMB and then enter the following:

```
Public Sub Main()
    Call PulseOut(5, 0.0020, 1)   ' Left servo counterclockwise
End Sub
```

Double-check that the middle number, (0.0020) was entered correctly. This is very important; otherwise, your servo could be damaged! Once you are sure the number is correct, compile and download this code to the BX-24 and see what happens. Don't blink – you may miss it! If everything was installed and programmed correctly, the Mouse's **left wheel** should have rotated a few degrees **counterclockwise**. That is, the left wheel of the Mouse should have moved one step **forward.**[18] Nothing to write home about yet, but it's a start.

(If you received the **Compile Error** message shown at the right, check to make sure that you actually did include the right parenthesis in your code. Chances are your right parenthesis is sitting there, as pretty as you please. If so, this probably means you were *lazy* and did not include the `Call` keyword in your code! Put it in now and recompile. The error message should not appear again.)


Compile Error

Line 4 : Missing right parenthesis

Let's make some sense of the `PulseOut` procedure. The syntax for this procedure is as follows:

`Call PulseOut`(*Pin, PulseWidth, State*)

where *Pin* is the pin number we wish to address, *PulseWidth* is the duration of the electrical pulse to be sent to the *Pin*, and *State* specifies whether the pulse is high or low. The data type for *Pin* is a Byte. *PulseWidth* is a Single data type that can have values that range from about 1.085μs (0.000001085s) to 71.1ms (0.0711s).[19] The *State* argument has the Byte data type and can be either high (1) or low (0).[20]

The line `Call PulseOut(5, 0.0020, 1)` procedure tells the BX-24 to send out one high pulse with a 0.0020s (2.0ms) pulse width to the left servo via pin 5 on the BX-24 (pin 0 on the RAMB). Since we are using modified servos, the 0.0020s pulse rotated our modified Futaba servo counterclockwise, making the left wheel move forward. The pulse generated by our code is illustrated in Figure 11.38.

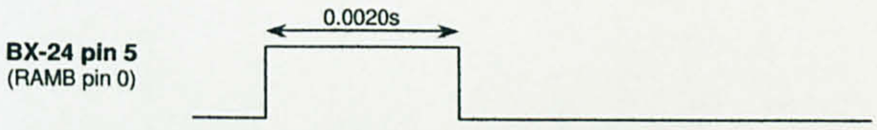


Figure 11.38. A diagram of the electrical pulse generated by `Call PulseOut(5, 0.0020, 1)`. A high pulse, whose duration is 0.0020s (2.0ms), is sent to pin 5 on the BX-24 (pin 0 on the RAMB).

[18] If you are using servos made by a company other than Futaba, your Mouse may move backwards rather than forwards. Keep reading to see how to correct your code.
[19] Just because the BX-24 can accept this wide range of pulse widths, don't assume that your servos can!
[20] A high pulse sends a +5V pulse out of the signal pin, and a low pulse sends 0V pulse.

... Put Your LEFT Foot Out ...

Now add the following lines of code to your program:

```
Delay(0.5)                          ' 500ms delay
Call PulseOut(5, 0.0010, 1)         ' Left servo clockwise
```

Run the program and you should observe the Mouse's left wheel move forward one "step", pause for half a second, and then move backwards one "step". The line that reads `Call PulseOut(5, 0.0020, 1)` will make the left servo turn counterclockwise a small amount, and the line `Call PulseOut(5, 0.0010, 1)` will make the left servo turn clockwise approximately by the same amount. The net result is that the Mouse should end up where it started. The only difference in these two lines of code is the *PulseWidth* argument – `0.0020` and `0.0010`; the value `0.0020` makes the Futaba servo rotate counterclockwise and `0.0010` causes the servo to rotate clockwise.[21]

Recall that servos must receive pulse widths only within a well-defined range. Most servos, including the Futaba S3004 that are shipped with the Robodyssey Mouse, have a minimum pulse width-limit around 0.0010s (1.0ms) and a maximum limit around 0.0020s (2.0ms). Sending the servo a pulse width outside this range may damage its control board, so check your code carefully to ensure that the pulse widths you are sending out are within the proper range.

Servo manufacturers suggest placing a small **delay** of at least 0.02s (20ms) between pulses to prevent overdriving the servo! The delay of 0.5s (500ms) in our program is more than sufficient.)

The Pulse of Our Program

Examine Figure 11.39, which will analyze what your program is doing. The program begins execution at point **A**, with the line `Public Sub Main()`. Very quickly the program reaches point **B** and the line of code that reads `Call PulseOut(5, 0.0020, 1)`, which sends a high pulse to the servo connected to pin 5 on the BX-24. The pulse width is 0.0020s, so the left wheel turns one step counterclockwise at this instant, causing the left wheel to move forward. At point **C**, pin 5 no longer outputs a pulse since the pulse duration has elapsed. The next line of code, `Delay(0.5)`, begins at point **C** and will delay any processing done by the BX-24 for 0.5s (500ms). The final line of code, `Call PulseOut(5, 0.0010, 1)`, sends another high pulse to pin 5 at point **D**. The pulse width here is 0.0010s, so the left wheel rotates one step clockwise, sending the left wheel backward. A brief instant after the termination of the final pulse at point **E**, the program terminates at point **F**, when the line `End Sub` is reached.

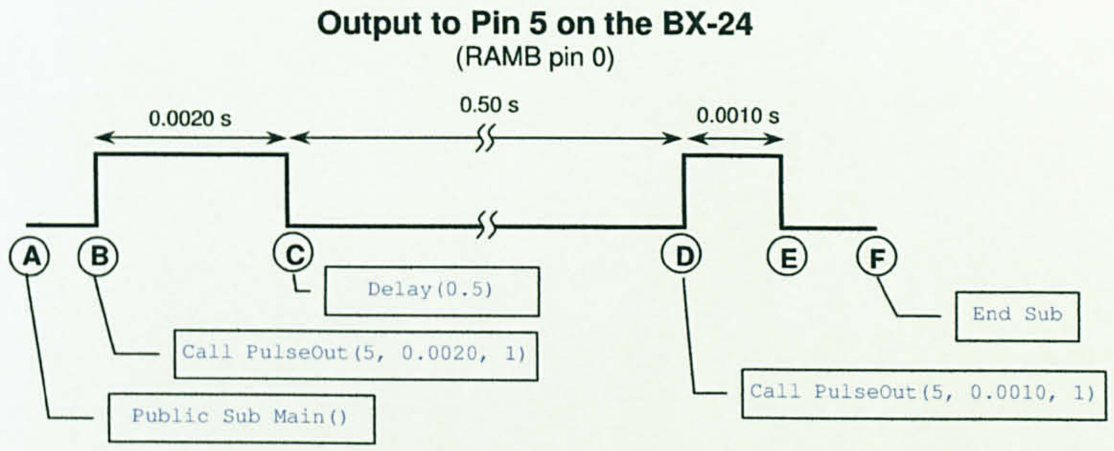

Output to Pin 5 on the BX-24
(RAMB pin 0)
0.0020 s
0.50 s
0.0010 s
A
B
C
D
E
F
Delay(0.5)
Call PulseOut(5, 0.0020, 1)
Public Sub Main()
End Sub
Call PulseOut(5, 0.0010, 1)

Figure 11.39. Two high pulses, whose durations are 2.0ms and 1.0ms, are sent to pin 5 on the BX-24 (pin 0 on the RAMB). A delay of 500ms separates the two pulses. The first pulse turns the Mouse's left wheel counterclockwise (forward) and the second pulse turns the Mouse's left wheel clockwise (reverse).

[21] This may be reversed for other servos.

Put Your RIGHT Foot In, Put Your RIGHT Foot Out...

All of this can be repeated using the **right servo**, allowing both wheels to get into the act. Let's modify our existing code to make **both** the left and right wheels take one step backward and one step forward.

To make the Mouse move **forward**, the *left* wheel must move *counterclockwise*, as we've already seen. The opposite is true for the right wheel: in order for the *right* wheel to move the Mouse forward, it must turn *clockwise* as shown in Figure 11.40. This is flip-flopped when we want the Mouse to move **backwards**: by moving the left wheel clockwise and the right wheel counterclockwise, the Mouse will be propelled backwards as shown in Figure 11.41.

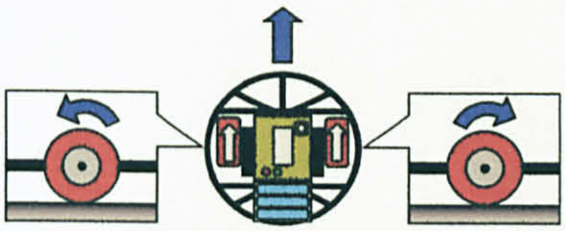


Figure 11.40. In order for the Mouse to move forward, the left wheel must rotate counterclockwise while the right wheel rotates clockwise.

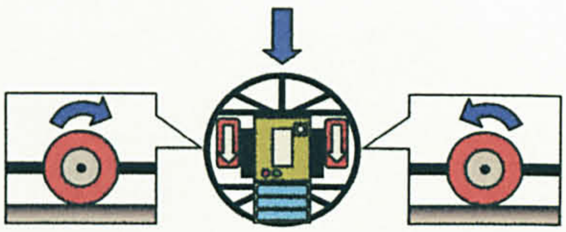


Figure 11.41. In order for the Mouse to move in reverse, the left wheel rotates clockwise while the right wheel rotates counterclockwise.

We can make this happen with code by sending a clockwise pulse to the right wheel immediately after we make the left wheel rotate counterclockwise. This will move the Mouse one step forward. Then, after the left wheel rotates clockwise, we can turn the right wheel counterclockwise, moving the Mouse backwards.

Our code should now look like this:

```
Public Sub Main()
   ' Mouse moves forwards
   Call PulseOut(5, 0.0020, 1)       ' Left servo counterclockwise
   Call PulseOut(6, 0.0010, 1)       ' Right servo clockwise

   Delay(0.5)                        ' 500ms delay

   ' Mouse moves backwards
   Call PulseOut(5, 0.0010, 1)       ' Left servo clockwise
   Call PulseOut(6, 0.0020, 1)       ' Right servo counterclockwise
End Sub
```

Do you see what's going on? The left wheel needs a pulse of 0.0010s and the right wheel needs a pulse of 0.0020s to make the Mouse move backwards because pulses of 1.0ms and 2.0ms make the Futaba servos turn clockwise and counterclockwise, respectively. You should convince yourself that the Mouse does, in fact, move in reverse when the left wheel rotates clockwise and the right wheel rotates counterclockwise. When thinking about moving and turning the Mouse, always keep in mind:

- ✓ ***A pulse width whose duration is less than 1.5ms will rotate Futaba servos clockwise.***
- ✓ ***A pulse width whose duration is greater than 1.5ms will rotate Futaba servos counterclockwise.***

Add Some CONSTants

It can get confusing having to remember the proper pin numbers and pulse widths when mobilizing your robot, but we can make robot programming much easier for ourselves if we incorporate some constants in our program. At the beginning of the `Main` program, create a constant to represent the BX-24 pin number that controls the left servo. Name the constant `LeftServo` and assign it the value 5, since we connected our left servo to BX-24 pin 5 (RAMB pin 0). Recall that the data type for the *Pin* argument of the `PulseOut` procedure is a Byte, so we must define our constant accordingly.[22]

[22] Revisit *Chapter 5: Variables, Constants, and Data Types* to refresh your memory about using `Const`.

However, why stop here when we can create another constant that will further simplify our program? We should create a constant named `Left_Forward` and give it a value of 0.0020. We can then replace the line that reads `Call PulseOut(5, 0.0020, 1)` with `Call PulseOut(LeftServo, Left_Forward, 1)`, which does exactly the same thing. (Using these constants makes programming the Mouse almost too easy!) To me, it is certainly easier to read and understand the line `PulseOut(LeftServo, Left_Forward, 1)` than `PulseOut(5, 0.0020, 1)`.

To make it complete, create and define four more constants that can be used to propel the Mouse forwards and backwards. Specifically, name them `LeftServo`, `Left_Reverse`, `Right_Forward`, and `Right_Reverse`, and set them to the appropriate values. Be certain that the values for the direction constants are between the minimum servo pulse width value of 0.0010s and the maximum value of 0.0020! Otherwise, you could damage your servos.

The following code shows how I defined and used the new constants. Running it will produce the same robot motion as before, but now the code is more versatile and much easier to read.

```
Public Sub Main()
   ' Servo Pin Constants
   Const LeftServo as Byte = 5                  ' Pin #0 on RAMB
   Const RightServo as Byte = 6                 ' Pin #1 on RAMB

   ' Servo Direction Constants (Pulse widths)
   Const Left_Forward as Single = 0.0020        ' Counterclockwise rotation
   Const Left_Reverse as Single = 0.0010        ' Clockwise rotation
   Const Right_Forward as Single = 0.0010       ' Clockwise rotation
   Const Right_Reverse as Single = 0.0020       ' Counterclockwise rotation

   'Mouse moves forwards
   Call PulseOut(LeftServo, Left_Forward, 1)
   Call PulseOut(RightServo, Right_Forward, 1)

   Call Delay(0.5)                              ' 500ms delay

   ' Mouse move backwards
   Call PulseOut(LeftServo, Left_Reverse, 1)
   Call PulseOut(RightServo, Right_Reverse, 1)
End Sub
```

11.11 Do the Cha-Cha!

Now that the Mouse can take baby steps, let's spice it up a bit and do the Cha-Cha! That is, let's make the Mouse take ten steps forward and five steps back. I know, I know. This isn't the way the Cha-Cha is *traditionally* done, but then again mice don't *traditionally* dance the Cha-Cha!

We *could* copy and paste the lines that make the Mouse take all of these steps, but why bother when we have For-To-Next loops at our disposal? Add a **Cha-Cha section** to our existing program, but first separate your existing code from this new stuff below with a two-second delay.

```
' ***** Do the Cha-Cha ******
Call Delay(2.0)                   ' Separation delay

Dim i as Integer
Const NumF as Integer = 10
Const NumB as Integer = 5
```

```
' Move forward
For i = 1 to NumF
     Call PulseOut(LeftServo, Left_Forward, 1)
     Call PulseOut(RightServo, Right_Forward, 1)
     Call Delay (0.25)
Next

' Move backward
For i = 1 to NumB
     Call PulseOut(LeftServo, Left_Reverse, 1)
     Call PulseOut(RightServo, Right_Reverse, 1)
     Call Delay (0.25)
Next
```

I created the constants, `NumF` and `NumB`, to represent the number of forward and backward steps. Both were incorporated in For-To-Next loops with a counter, `i`. The creation of these constants was perhaps unnecessary. We could have run our loops with numeric literals `10` and `5`. However, programming with constants makes for efficient code and it is a good habit to get into. Finally, the four `PulseOut` commands and quarter-second delays leave us with one swingin' Mouse.

If you would like to see it "dance" again, simply press the reset button on the RAMB! As an exercise at the end of this chapter, modify the code to make the Mouse repeat this dance any number of times. Create your own dance steps by varying the values of `NumF`, `NumB`, and the delay. Experiment and have some fun.

Do the Cha-Cha Faster!

Rodents are supposed to be fast, so let's speed up the Mouse's dance by reducing the delay time between each step. Try replacing `Delay (0.25)` with `Delay (0.1)`. Run the program and you'll see that the dance is now faster, but still quite choppy.

Let's make the dance go *as fast as possible* and see what happens. The manufacturer of our servos recommends a minimum delay time of 0.020s (20ms) between consecutive pulses to any one servo. This gives the servo ample time to fully rotate. Therefore, using `Delay(0.02)` is the smallest delay the servomotors can handle, pulsing the wheels about 50 times per second.

Speed Is a Virtue

As explained above, the Mouse's speed can be controlled by varying the delay times between servo pulses. We can alter the speed of the robot more readily with a `Speed` variable and a few constants. It makes sense to define three speeds for our Cha-Cha application: `Fast`, `Medium`, and `Slow`. Let's arbitrarily pick delay times of 0.02s, 0.10s, and 0.50s for `Fast`, `Medium`, and `Slow`, respectively. Add these lines to the beginning of your application, below the program's other constants:

```
' The Mouse's speed is controlled by these delay times (in seconds):
Const Fast as Single = 0.02             ' 50 steps per second (fastest)
Const Medium as Single = 0.10           ' 10 steps per second
Const Slow as Single = 0.50             ' 2 steps per second
Dim Speed as Single                     ' Use to adjust delay interval
```

If we wish to have the Mouse move at the fastest possible speed, all we have to do is define the `Speed` variable as `Fast` as shown here:

```
Speed = Fast
```

You can put this line anywhere in the program – just make sure that it comes before you need to use it. To use the `Speed` variable, simply change all the delay calls in your program as follows:

```
Call Delay(Speed)                ' Controls speed
```

Do you see how Speed is used within the Delay procedure? Nifty, huh? Now it is easy to alter the Mouse's speed by simply altering the Speed variable! Of course, you could incorporate the desired speed value directly into the Delay call:

```
    Call Delay(Fast)                        ' Short delay = fast speed
```

Feel free to alter the values of the constants, but remember that the smallest possible delay (i.e., the Mouse's fastest speed) is 0.02s (20ms)! In other words, Fast is as fast as is safe.

The following Self Test shows how you can alter your program to make the Mouse move eight steps forward at Fast speed and eight steps backward at Slow speed. Try doing this yourself before looking at my solution.

Self Test # 1. Spring Ahead, Fall Back

Problem: Alter the Cha-Cha part of your current project so that the Mouse will move eight steps forward at Fast speed and eight steps backward at Slow speed. Take note of the distances traveled in each leg of the journey

Solution: This is an almost trivial solution, but you will need to carefully alter the necessary constant and variable values. (I've indicated these in bold.)

```
Const NumF as Integer = 8
Const NumB as Integer = 8

Speed = Fast
' Move forward
For i = 1 to NumF
     Call PulseOut(LeftServo, Left_Forward, 1)
     Call PulseOut(RightServo, Right_Forward, 1)
     Call Delay (Speed)
Next

Speed = Slow
' Move backward
For i = 1 to NumB
     Call PulseOut(LeftServo, Left_Reverse, 1)
     Call PulseOut(RightServo, Right_Reverse, 1)
     Call Delay (Speed)
Next
```

Compare the distance traveled by the Mouse in the forward and backward directions. Running the program, I find that my Mouse moves forward about one inch and backwards about three. This difference is due to the Speed value (that is, the delay time between pulses). At Fast speed, new PulseOut commands are received by the servo while it is still carrying out the previous command and complete rotations are not possible.

11.12 To Everything Turn, Turn, Turn

Going forwards and backwards is great, but it wouldn't be much of a robot if we couldn't make it turn. There are many ways to do this, and in this section we will use For-To-Next loops to make the Mouse turn in circles. To do so is actually quite easy; perhaps you've already figured it out. Here are a few ways that it can be done.

Turning on a Dime – Literally

To make the **quickest and tightest turn** possible, we must simultaneously *counter-rotate* each of the Mouse's wheels. This has the effect of pivoting the Mouse to the left about its center – literally turning it on a dime. Therefore, to make the Mouse **rotate to the left**, we must reverse the left wheel while the right wheel is rotating forward. In other words, for sharp left turns, both wheels must turn clockwise, as shown in Figure 11.42. To make the Mouse **rotate to the right**, reverse the right wheel and drive the left wheel forward; for sharp right turns, both wheels must rotate counterclockwise.

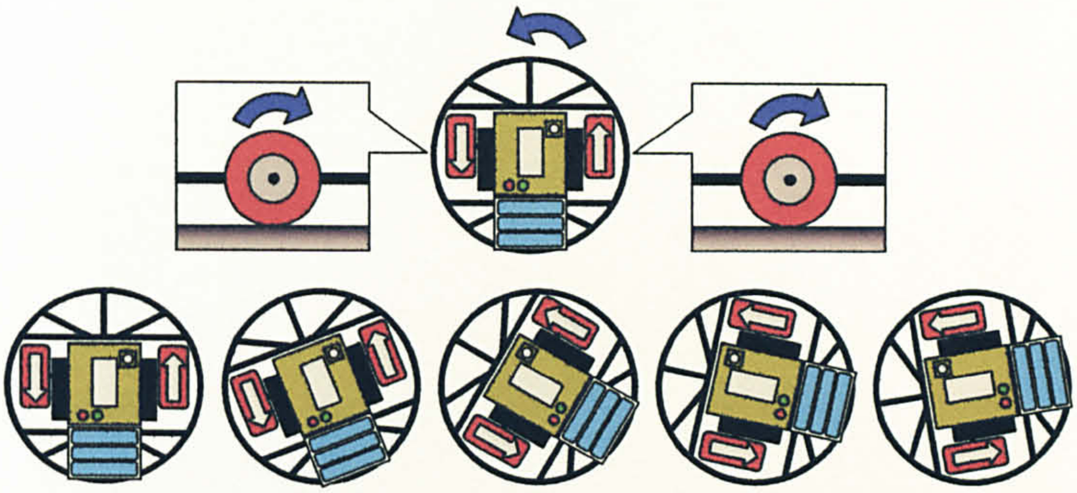


Figure 11.42. To make the Mouse turn left, as if pivoting about its center, rotate both wheels clockwise. That is, turn the left wheel backwards and the right wheel forwards.

To make the Mouse spin ten "steps" to the left, try this bit of code. (I added another two-second delay in my program to separate this new code from the old code. What can I say – I'm a creature of habit.)

```
' ***** Center Rotation ******
Call Delay(2.0)                  ' Separation delay
Speed = Slow                     ' Choose a speed

' Turn ten steps to the left
For i = 1 to 10
     Call PulseOut(LeftServo, Left_Reverse, 1)
     Call PulseOut(RightServo, Right_Forward, 1)
     Delay (Speed)
Next
```

I arbitrarily chose to use the `Slow` speed for this exercise – the delay time is 0.50s (500ms). After 10 pulses, how far did your Mouse rotate? Mine rotates approximately 90° with fresh batteries and 70° with run-down batteries. (Changing the `Speed` variable will affect how far ten pulses will rotate the robot – the greater the speed, the more pulses are required.)

To turn to the right, simply reverse the process by rotating the left wheel forward and the right wheel backward:

```
' Turn ten steps to the right
For i = 1 to 10
     Call PulseOut(LeftServo, Left_Forward, 1)
     Call PulseOut(RightServo, Right_Reverse, 1)
     Delay (Speed)
Next
```

The Wheels on the Mouse Go 'Round and 'Round ...

One question worth asking is, "*How many pulses does it take to turn my Mouse a total of* 360°?" I wish I could say that all servos behave identically, but I can't. Even if they did, my Mouse would move in a different way from yours due to differences in battery power, smoothness of the tail wheel, and friction forces within the servo and between the wheels and tabletop. Even our delay times play an important role. For example, *my* Mouse, with fully charged batteries on a smooth surface, is capable of pivoting 360° about its center in 22 steps with a delay time of 0.5s (`Speed = Slow`) and in 57 steps with a delay time of 0.02s (`Speed = Fast`).

Take the time to work Challenge Problem #1 at the end of this chapter and fill in the worksheet documenting the number of pulses required to rotate your Mouse 360°. It will come in handy as a reference as you complete this book or when you decide to compete in a local robot competition!

Sidewinder Turning

You need not always turn your Mouse by rotating it about its center – you can pivot it about one of its wheels. To do this, simply keep one wheel stationary and make the other rotate. For instance, to make a **wide turn** to the left, rotate the right wheel clockwise and leave the left wheel alone, as shown in Figure 11.43.

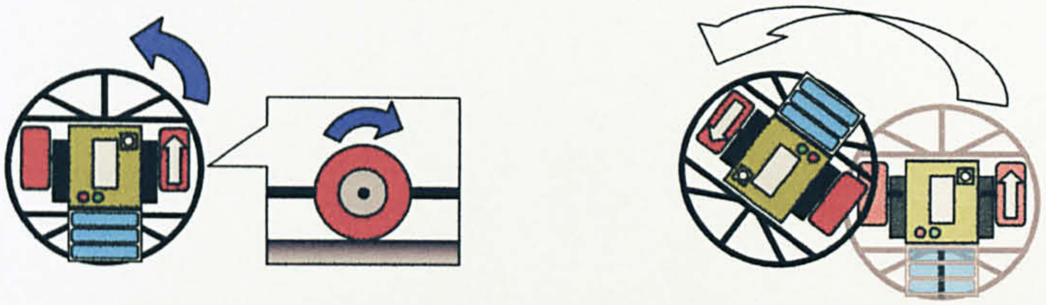


Figure 11.43. Wide turns can be made by rotating one wheel while keeping the other stationary. Here the Mouse is pivoting about its left wheel by rotating the right servo clockwise.

This type of turning action reminds me of the motion of the sidewinder snake. To see this sidewinder-turning in action, perform the following Self Test, which will make the Mouse take three large strides.

Self Test # 2. The Side-Winding Mouse

Problem: Make the Mouse move left-to-right by alternately rotating the left and right servos forward. If done properly, this motion will resemble that of a sidewinder snake. Make three of these large lumbering steps as quickly as possible.

Solution: The Mouse's first step is a half-circle turn pivoting 180° about the right wheel. When the first turn is completed, rotate the Mouse 180° about the left wheel. Repeat this process of rotating about the right wheel and then about the left two more times.

To make the Mouse turn as quickly as possible, use the `Fast` speed (with a delay of 0.02s between pulses to the servo). At this rate, I know that *my* Mouse requires 70 pulses to complete one 180° "step". Furthermore, I know that the Mouse is to make six such steps, or three pairs of right-left strides. To do this most efficiently, we should use *nested For-To-Next loops*, using one counter to keep track of the three of big strides and another counter to count the individual pulses required to make each step.

Check it out!

Solution continued on the next page...

Self Test # 2. The Side-Winding Mouse (*Continued from previous page*)

Solution: (*Continued*)

```
' *****  Side-Winder Turning *****
Call Delay(2.0)                                 ' Separation delay

Dim StrideCounter as Integer                    ' Our stride counter
Const NumOfStrides as Integer = 3               ' Easy to change!
Const PulsesPerStep as Integer = 70             ' 70 pulses = 180 deg.

Speed = Fast

For StrideCounter = 1 to NumOfStrides
     ' Pivot about the RIGHT wheel:
     For i = 1 to PulsesPerStep
          Call PulseOut(LeftServo, Left_Forward, 1)
          Call Delay(Speed)
     Next

     ' Pivot about the LEFT wheel:
     For i = 1 to PulsesPerStep
          Call PulseOut(RightServo, Right_Forward, 1)
          Call Delay(Speed)
     Next
Next
```

To run this program you will need about 2.5 feet of desk or floor space, so clear away an area before running the code!

Both center-pivot and sidewinder turning have advantages and disadvantages. Because it requires fewer pulses, center-pivot turning can be done **more rapidly**. On the other hand, sidewinder turning is **more precise** since the Mouse will turn fewer degrees per pulse. Of course, center-pivot turning requires virtually no additional room to turn around, while sidewinder turning needs a considerable amount of space.

11.13 It Keeps Going, and Going, and Going, ...

There is no reason that our Mouse ever has to stop moving (unless the batteries run down or the robot collides with an immovable object!). Therefore, we will finish this chapter with a rather simple program that will allow the Mouse to move forward indefinitely.

Why don't **you** try to figure out how to do it, and then examine the following Self Test to see how I did it?

Self Test # 3. Forward – Ho!

Problem: Make the Mouse move forward indefinitely at `Medium` speed.

Solution: Using a For-To-Next loop is fine if you wish to make a pre-determined number of steps like we did with the Cha-Cha, but now we want the Mouse to run indefinitely. How should we **do** this? With a **Do-Loop**, of course!

Solution continued on the next page...

Self Test # 3. Forward – Ho! (*Continued from previous page*)

Solution: (*Continued*)

```
' ***** Forever Forward! *****
Call Delay(2.0)                    ' Separation delay
Speed = Medium

Do
   Call PulseOut(LeftServo, Left_Forward, 1)
   Call PulseOut(RightServo, Right_Forward, 1)
   Call Delay (Speed)
Loop
```

Since we've programmed the Mouse to never stop, be prepared to **manually stop** the program's execution with the Stop button on the Downloader or with the power switch on the RAMB. Otherwise, your Mouse may run off the table and crash to the floor! Challenge Problem #4 at the end of this chapter asks you to modify this program so the LEDs blink as the Mouse moves. How does this affect the Mouse's speed?

11.14 Troubleshooting and Final Comments

Wheel Wobbles

Run the code from Self Test #3, but alter it so that it runs at `Fast` speed. Carefully examine the Mouse's wheels as they rotate. If one wheel wobbles, make sure that the axle screw, which attaches the wheel to the servo shaft, is tight – but do **not** over tighten! If the wobble continues, remove the wheel, rotate it a few degrees and reattach it to the shaft. Repeat this procedure until the wobble stops or is lessened.

Straight Shooter

Program your Mouse to move forward; take note of its motion. The Mouse **should** move in a perfectly straight line, but some robots will veer in a slight arc as shown in Figure 11.44.

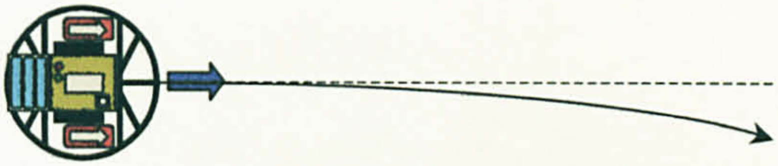


Figure 11.44. Sometimes a robot will veer from the desired straight-line path and travel in a slight arc.

If you experience this problem, there are several things to check that may correct the problem:

- ❑ If you are using the rear tail wheel, replace it with the phenolic ball.
- ❑ Make sure that you are sending the correct pulse widths to your servos – the **minimum** pulse width should be 0.0010s and the **maximum** should be 0.0020s.
- ❑ Run the Mouse at `Fast` speed.
- ❑ Place the Mouse on different surfaces – sometimes the wheels will slip on hard or slick surfaces.
- ❑ Recharge the batteries. All robots behave more predictably when they operate at full power.
- ❑ Make sure your servos are firmly attached to the Mouse chassis.
- ❑ Try calling the `PulseOut` commands in reverse order. If you are pulsing out the right servo first, try pulsing to the left one first.

If these suggestions don't make your Mouse move in a straight line, don't fret; I will show you how to tweak the robot's movement in *Chapter 13: Advanced Servo Operations.*

Smoothing Out the Curves

My students often want their Mouse to turn in smooth arcs rather than with sharp 90° turns, as shown in the figures below. I tell them this is certainly possible to do, and in fact, it is covered in *Chapter 13: Advanced Servo Operations*. I also point out that all robot programmers start off turning their robots with sharp turns.[23]

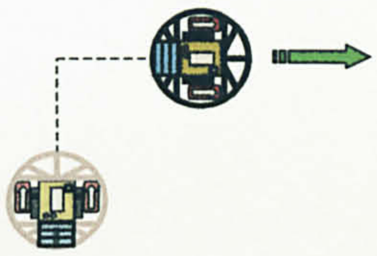


Figure 11.45. It is easy to make a robot turn in sharp 90° angles.

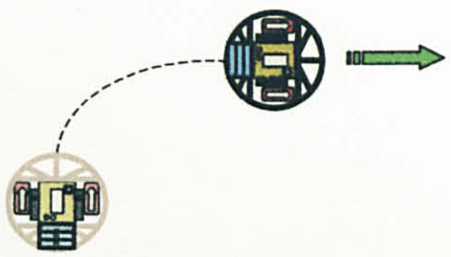


Figure 11.46. Soon we will learn how to make the robot turn in smooth arcs.

Personally, I think that it is important for novice programmers to practice the rudimentary basics before making their robot behave more gracefully. With that in mind, try the following Challenge Problems, and put to use the skills you learned in the previous chapters. I strongly suggest that you pick a few problems that interest you and sink your teeth into them.

Happy problem solving!

Challenge Problems

Remember: **always keep your pulse widths between the minimum and maximum allowed values!** That is, between **0.0010s and 0.0020s**. Use the constants, `Left_Forward`, `Right_Forward`, etc, to ensure you don't enter a wrong number. Also, **always use a delay of at least 0.02s between successive pulses**.

11-1. Determine how many pulses it takes to turn your Mouse 360° about its center, and fill in the worksheet below. Vary the delay time between pulses and the direction of rotation. You should simulate actual conditions and remove the serial cable before performing the test.

Number of Pulses to Rotate the Mouse 360°

Pulse Widths: Maximum: ______ Minimum: ______

		Turning Left		Turning Right	
Delay time:	0.02s	# of Pulses:		# of Pulses:	
Delay time:	0.1s	# of Pulses:		# of Pulses:	
Delay time:	0.5s	# of Pulses:		# of Pulses:	
Delay time:		# of Pulses:		# of Pulses:	
Delay time:		# of Pulses:		# of Pulses:	

[23] To make the turns more consistent, you may wish to put a 0.25s or so delay between any forward movements and your turns. This small delay will give the Mouse time to coast to a stop before turning.