PET ROBOT

By

FARALIZA BT MOHAMED

FINAL PROJECT REPORT

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Pet Robot

by

Faraliza Bt Mohamed

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 & Electronics Engineering)

Approved:

Dr Mumtaj Begam
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2008

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.

Faraliza Bt Mohamed

ABSTRACT

The idea of the PET ROBOT is highly appropriate due to the rapid changing technology in the modern world and the humans changing ways of life. This technology could not only be the replacement for house pets but with detailed design and innovation it could be an assistant to humans at homes. Pet robot uses a microcontroller to control its functions. The microcontroller will carry out instructions from the designed coding that is implemented to the microcontroller. Coding is designed and compiled using PIC Programming software. Different types of sensors are placed to the robot to give it intelligence. The pet robot will be not only be able to move forward, backward and turn but with the ability to 'see' by implementing sensors, the robot is also able to avoid object obstacles along the way. Besides that, the robot can react to certain external input such as performing certain functions when it detects light and can react to sound. The project work requires both mechanical field for movement and electrical field for controlling the robot.

ACKNOWLEDGEMENT

Alhamdulillah, thank you to Allah s.w.t for giving me the strength to complete this project.

First of all, I would like to thank Dr Mumtaj Begam, my project supervisor for leading me and showing me the correct ways in pursuing this project. Without her I would be lost and may not be able to complete this project on time.

I would also like to thank the technicians for giving me a chance to use the equipments in the lab as well as giving me guidance on how to use them properly and safely. They are most appreciated.

Secondly, I would like to thank my parents as well as my friends who had indirectly supported in completing this project. Their eagerness to see my project complete raises my spirit and strengthens my motivation to complete this project. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1. PET ROBOT

The title of this project is 'PET ROBOT'. The idea is to build a robot that acts and behaves similar to a pet. It is called a pet robot due to its ability to perform functions imitating a real life pet. The functions of the robot will be controlled by a programmed chip.

1.1 Background of Study

The word robot gives meaning of a machine that can do work by itself, often work that humans do. [10] The concept of robots is a very old yet the actual word robot was invented in the 20th century from the Czechoslovakian word robota or robotnik meaning slave, servant, or forced labor. [12] Robots are very visible machines, ranging from small, miniature machines, to large crane size constructions with intelligence varying from simple programming to perform mechanical tasks, such as painting a car or lifting cargo, to highly complex reasoning algorithms mimicking human thought. [11] Historically, we have sought to endow inanimate objects that resemble the human form with human abilities and attributes. From this is derived the word anthrobots, robots in human form. Robots are especially desirable for certain work functions because, unlike humans, they never get tired; they can endure physical conditions that are uncomfortable or even dangerous; they can operate in airless conditions; they do not get bored by repetition; and they cannot be distracted from the task at hand. Robots have been useful in industrials, hazardous duty, maintenance work, fire – fighting, medical, space explorations as well as wars. Early industrial robots handled radioactive material in atomic labs and were called master/slave manipulators. They were connected together with mechanical linkages and steel cables. Remote arm manipulators can now be moved by push buttons, switches or joysticks. Current robots have advanced sensory systems that process information and appear to function as if they have brains. Their "brain" is actually a form of computerized artificial intelligence (AI). AI allows a robot to perceive conditions and decide upon a course of action based on those conditions. [12]

1.2 Problem statement

The ability to produce a functioning robot with good stability and control takes high technology as well as research. Robots are closely related to AI (artificial intelligence) where scientists are still on research to produce a robot which is capable of thinking and making its own decision (unprogrammed).

The main idea of this project is to build a robot that can imitate a pet (for example a cat or a dog). There are a few points of significance in having a pet robot to human beings. When pet robots are designed as close as being to a real life pet, it could be the next innovation of replacing real life pets. Owning a pet is something people desire to have but in this modern evolving era, people are too busy to handle and take care of pets at home. By having a pet robot instead of a real one, people can now have the pleasure of owning a pet without the fuss of maintaining or care taking them. Besides that, pet robots can also be a companion to humans. It can play the role of a 'mans best friend' just like dogs. For example with the rising statistics of senior citizens, pets can be a great companion to them and accompany them throughout their remaining life. Unfortunately, majority of them do not have the capability of holding the responsibility of maintaining a pet (for example giving them baths and feeding them). By replacing these pets with a pet robot, they are able to keep a pet without pressuring their ability to take care of these them. Another application of pet robot in our everyday life is that it can be a good assistant to humans. By enhancing and adding more features to the robot, it can help people in various ways. For example, it can help blind people guiding them in walks by implementing sensors to the robot to detect object obstacles blocking their way. It can also be enhanced to supervise children. The pet robot can be a toy to the child and also be a nanny for parents to ensure their child is safe and sound.

Throughout the years many robots have been built and enhanced to perform different tasks for humans. Every type of robot was given specific functions and task just by enhancing the basic foundation of a robot. In this project, the author is rebuilding the foundation of all robots and adding features to make it as similar as a pet robot. The pet robot will have basic fundamental functions. By building the basic pet robot it can later be enhanced with more features and more sophisticated code programming to achieve the goals mentioned above.

1.3 Objective and Scope of Study

The main idea of this project is not only to build a robot, but to make it able to imitate the behavior of pets. It is difficult to make the robot to fully imitate all the behaviors of a cat or a dog, but some personality can be implemented to the robot as so it is acceptable as a pet. The behavior and characteristics of the robot is very subjective and general. Some of the ideas considered for this project are:

1) Random movement functions

   Most pets have their own behavior and characteristics. We need not order them on how to move or how to act. In order to implement these criteria to the robot, it will need to have random movements on its own. It will choose its own path and its own way of movements to go forward, reverse, turn or turn, reverse and turn again, it is all to be decided randomly. The movements will be different in each cycle. This will give the robot an essence of life. It will also make the robot seem more active.

2) Detecting and avoiding object obstacles

   Like most living creatures, the robot has 'eyes' which enable it to see what is in front of it. With this feature, the robot is able to see the object blocking its path. The robot is then given the intelligence to avoid the object and prevent it from bumping into things. This gives the robot a characteristic of independence.

3) Light detecting

   Every living creature has a natural feeling of fear. Therefore by implementing this feature it could reduce the impression of robotic towards the robot. The robot will have a fear of darkness. When it is in the dark, it will stop and start to glow in the dark as if it wants us to find it. Besides glowing in the dark, it will also start to behave strangely by moving in a peculiar way (for example shaking) to show its reaction of fear.

These are the main characteristics that the author has considered to implement in the robot. Once the main circuit has been completed, more functions and characteristics could be added to enhance the robot.

1.3.1 Objective:

1) To design and build a robot with the ability to:

- Move forward
- Move backward
- Turn right and turn left
- Detect object obstacles
- Avoid object obstacles
- Detect light
- React to light

2) To design the codings and implement them in a microcontroller to control the functions and movements of the robot.

3) To program the robot to have its own behavior and characteristics by having random movements

1.3.2 Scope of study

To achieve the objective of this project, studies and research on areas related with robotics is concentrated. In order to build a basic robot, basic functions such as moving forward, reverse and turning will have to be applied to the robot. This will require a combination of electrical and mechanical field knowledge and application where integrated electrical circuits will guide the mechanical movements of the robot.

In the mechanical area, the robot must have components that will enable it to move forward, reverse and turn. Commonly there are two types of robot to enable these basic functions. First is the walking legged robot and the other is a wheeled robot. Both of these two options are considered to design a robot most suitable for the specifications of a pet robot.

In the electrical area, the circuit for the brain of the robot is built. The main component of the brain is a microcontroller which will control the movement of the robot. A programmable microcontroller is needed to provide flexibility to the functions and its special features. There are many types of microcontroller and the most suitable one will be selected to be used in this project. To control the robots basic movement (forward, reverse, turn), a dc motor is needed and it will be controlled by the chip.

In order to program the microcontroller, a basic knowledge in programming is needed. Flexibility in coding can provide a wider variety and possibility to the functions of the robot. Software in these areas is needed to design and implement the coding to the chip.

Since the robot is required to detect objects and lights, sensors are needed to perform these functions. When these sensors are triggered, they will send a signal to the microcontroller. The microcontroller will then send out a signal to the necessary component to show reaction to the sensors. There are many types of sensors to choose from but for this project only two types of sensors are being considered that is the infrared sensor and the ultrasonic sensor. Another type of sensor needed for the robot is the light sensor. Photocells are used and will serve the purpose of detecting light when they are illuminated. It will be integrated to a circuit to send signal to the microcontroller to perform the appropriate action. This will enable the robot to differentiate between dark and bright areas.

CHAPTER 2

LITERATURE REVIEW AND/OR THEORY

Research has been done on different areas of this project to learn more on how to build a pet robot. The research of this project has been divided into three sections which is the microcontroller circuits, the mechanical movements and the programming.

2.1 Microcontroller and Circuits

There will be several circuits implemented to the robot. All of these circuits will be combined and integrated together to create the fundamental of a basic robot. All input and output signals of each circuit will be sent to the main circuit which will play the role of commanding and giving instructions to other components. Basically there are three major circuits:

a) Main circuit

b) H-bridge circuit

c) Infrared sensor circuit

d) Light sensor circuit

2.1.1 Main Circuit

All the movements and decisions of the robot will be controlled by one main circuit. This circuit contains the PIC microcontroller which is programmed to control the functions of the robot. The output the pins of the microcontroller is then connected to the motors of the robot and other output components.

The main circuit contains these major components:

- Microcontroller PIC 16F84A
- Voltage Regulator
- Oscillator Clock

2.1.1.1 Microcontroller

The brain of the robot will be controlled by a microcontroller chip. Various types of chip have been researched and the author prefers to use PIC 16F84A. This is due to a few advantages that this chip holds. This programmable microcontroller is commonly used in integrated circuits. The size of the robot is designed to be small to achieve its motive of imitating a pet. PIC 16F84A is small enough to be placed in the main circuit of the robot. Although it has less number of inputs and output pins, it is enough to cater for all the functions for the pet robot in this project. This chip will be programmed to perform the required tasks. The program memory contains 1K words, which translates to 1024 instructions, since each 14-bit program memory word is the same width as each device instruction .The data memory (RAM) contains 68 bytes. Data EEPROM is 64 bytes.

There are also 13 I/O pins that are user-configured on a pin-to-pin basis. Some pins are multiplexed with other device functions. These functions include:

• External interrupt
• Change on PORTB interrupts
• Timer0 clock input
Features:
• Operating speed: DC - 20 MHz clock input
DC - 200 ns instruction cycle
• 1024 words of program memory
• 68 bytes of Data RAM
• 64 bytes of Data EEPROM

• 14-bit wide instruction words
• 8-bit wide data bytes
• 15 Special Function Hardware registers
• Eight-level deep hardware stack
• Direct, indirect and relative addressing modes
• Four interrupt sources:
- External RB0/INT pin
- TMR0 timer overflow
- PORTB<7:4> interrupt-on-change
- Data EEPROM write complete

Peripheral Features:
• 13 I/O pins with individual direction control
• High current sink/source for direct LED drive
- 25 mA sink max. per pin
- 25 mA source max. per pin
• TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features:
• 10,000 erase/write cycles Enhanced FLASH Program memory typical
• 10,000,000 typical erase/write cycles EEPROM Data memory typical
• EEPROM Data Retention > 40 years
• In-Circuit Serial Programming™ (ICSP™) - via two pins
• Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST)
• Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
• Code protection
• Power saving SLEEP mode
• Selectable oscillator options

Below is a pin diagram of PIC 16F84A:

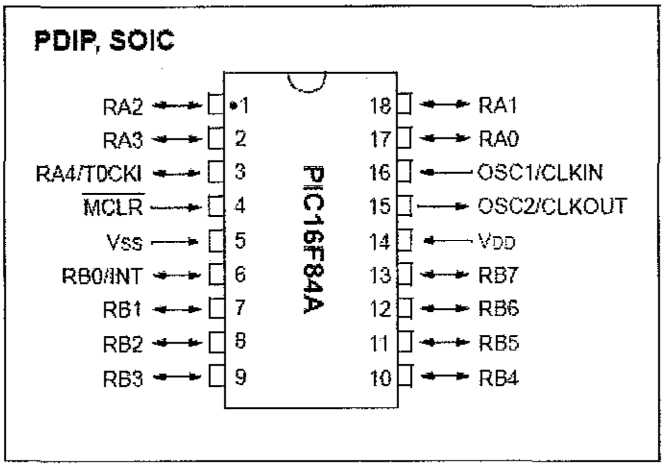


Figure 1: Pin Diagram of PIC 16F84A

The software that the author used to program the PIC16F84A is the PIC C Compiler and it will be simulated by PIC simulator IDE. The datasheet for PIC 16F84A is shown in appendix A.

2.1.1.2 Voltage Regulator

A voltage regulator is an electrical regulator designed to automatically maintain a constant voltage level. Essentially, all a voltage regulator does is, obviously, regulate voltage; That is, it limits the voltage that passes through it. Each regulator has a voltage rating; For example, the 7805 IC (these regulators are often considered to be ICs) is a 5-volt voltage regulator. No matter how many volts is put into it, it will output only 5 volts. This means that by connecting 9-volt battery, a 12-volt power supply, or virtually anything else that's over 5 volts, and have the 7805 will give a of 5 volts out. There are also 7812 (12-volt) and 7815 (15-volt) three-pin regulators in common use. The pinout for a three-pin voltage regulator is as follows [13]:

1. Voltage in
2. Ground
3. Voltage out

For example, with a 9-volt battery, connect the positive end to pin 1 and the negative (or ground) end to pin 2. A 7805 would then give a +5 volts on pin 3. Voltage regulators are simple and useful. There are only two important drawbacks to them: First, the input voltage must be higher than the output voltage. For example, we cannot give a 7805 only 2 or 3 volts and expect it to give a 5 volts in return. Generally, the input voltage must be at least 2 volts higher than the desired output voltage, so a 7805 would require about 7 volts to work properly. The other problem: The excess voltage is dissipated as heat. At low voltages (such as using a 9-volt battery with a 7805), this is not a problem. At higher voltages, however, it becomes a very real problem and you must have some way of controlling the temperature so you don't melt your regulator[13].

This is why most voltage regulators have a metal plate with a hole in it; That plate is intended for attaching a heat sink to [13]. Figure 2 shows the voltage regulator pins.

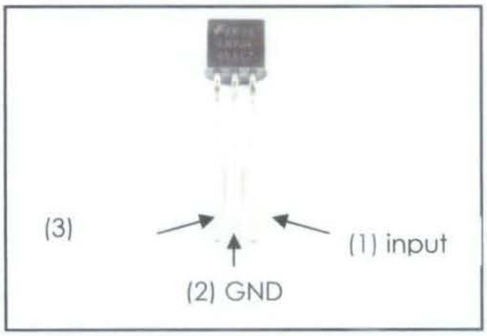


Figure 2: Voltage Regulator 7805

The circuit is supplied with a 9Volt Battery. The PIC only uses 5volts. A 5 volt voltage regulator 7850 is used to step down the power supply from 9V to 5V. Figure 3 shows the connection diagram of the voltage regulator.

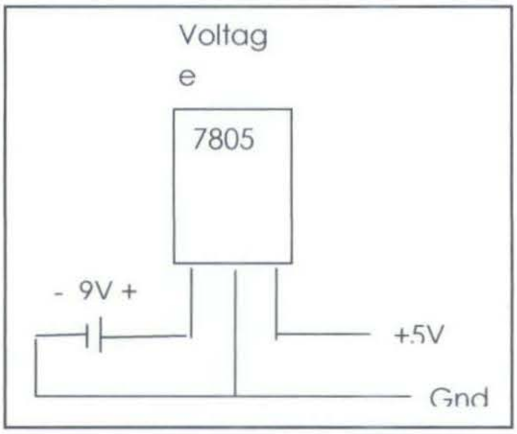


Figure 3: Voltage Regulator Connection Diagram

2.1.1.3 Oscillator Clock

A crystal oscillator is an electronic circuit that uses the mechanical resonance of a vibrating crystal of piezoelectric material to create an electrical signal with a very precise frequency. This frequency is commonly used to keep track of time (as in quartz wristwatches), to provide a stable clock signal for digital integrated circuits, and to stabilize frequencies for radio transmitters/receivers.

The crystal oscillator circuit sustains oscillation by taking a voltage signal from the quartz resonator, amplifying it, and feeding it back to the resonator. A regular timing crystal contains two electrically conductive plates, with a slice or tuning fork of quartz crystal sandwiched between them. During startup, the circuit around the crystal applies a random noise AC signal to it, and purely by chance, a tiny fraction of the noise will be at the resonant frequency of the crystal. The crystal will therefore start oscillating in synchrony with that signal. As the oscillator amplifies the signals coming out of the crystal, the signals in the crystal's frequency band will become stronger, eventually dominating the output of the oscillator. Natural resistance in the circuit and in the quartz crystal filter out all the unwanted frequencies.One of the most important traits of quartz crystal oscillators is that they can exhibit very low phase noise. In many oscillators, any spectral energy at the resonant frequency will be amplified by the oscillator, resulting in a collection of tones at different phases. In a crystal oscillator, the crystal mostly vibrates in one axis. Therefore, only one phase is dominant. This property of low phase noise makes them particularly useful in telecommunications where stable signals are needed, and in scientific equipment where very precise time references are needed.[2]

Figure 4: Crystal Clock Oscillator

2.1.2 H-BRIDGE

An H-bridge is an electronic circuit which enables DC electric motors to be run forwards or backwards. These circuits are often used in robotics. H-bridges are available as integrated circuits, or can be built from discrete components. Figure 5 shows the H-bridge connection circuit.

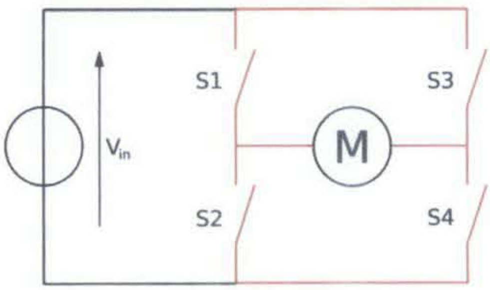


Figure 5: H-Bridge Connection circuit

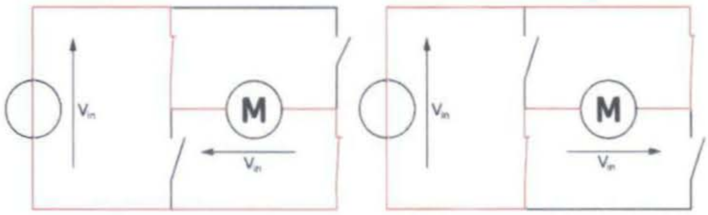


Figure 6: Two States of H-Bridge

Using the nomenclature above, the switches S1 and S2 should never be closed at the same time, as this would cause a short circuit on the input voltage source. The same applies to the switches S3 and S4. This condition is known as shoot-through.

The H-Bridge arrangement is generally used to reverse the polarity of the motor, but can also be used to 'brake' the motor, where the motor comes to a sudden stop, as the motors terminals are shorted, or to let the motor 'free run' to a stop, as the motor is effectively disconnected from the circuit. Table 7 summarises the operation [3].

Table 1: H-Bridge Summarize Function

S1	S2	S3	S4	Result
1	0	0	1	Motor moves right
0	1	1	0	Motor moves left
0	0	0	0	Motor free runs
0	1	0	1	Motor brakes

2.1.3 Infrared Radiation

Infrared (IR) radiation is electromagnetic radiation of a wavelength longer than that of visible light, but shorter than that of microwaves. The name means "below red" (from the Latin infra, "below"), red being the color of visible light with the longest wavelength. Infrared radiation has wavelengths between about 750 nm and 1 mm, spanning five orders of magnitude. Humans at normal body temperature can radiate at a wavelength of 10 microns.

Infrared light lies between the visible and microwave portions of the electromagnetic spectrum. Infrared light has a range of wavelengths, just like visible light having wavelengths that range from red light to violet. "Near infrared" light is closest in wavelength to visible light and "far infrared" is closer to the microwave region of the electromagnetic spectrum. The longer, far infrared wavelengths are about the size of a pin head and the shorter, near infrared ones are the size of cells, or are microscopic.

Shorter and near infrared waves are not hot at all - in fact you cannot even feel them. These shorter wavelengths are the ones in TV's remote control [7].

The Infrared emitter detector circuit is very useful to make a line following robot, or a robot with basic object or obstacle detection. Infrared emitter detector pair sensors are fairly easy to implement, although involved some level of testing and calibration to get right. They can be used for obstacle detection, motion detection, transmitters, encoders, and color detection (such as for line following) [8].

2.1.4 Light Sensor

One of the functions of the robot is when it detects that there is no lights, it will stop every movement and blink a set of LEDs. Light Dependent Resistor (LDR) is a light sensor that can be used for this function.

2.1.4.1 Light Dependent Resistor (LDR)

LDRs or Light Dependent Resistors are very useful especially in light/dark sensor circuits. Normally the resistance of an LDR is very high, sometimes as high as 1000 000 ohms, but when they are illuminated with light, resistance drops dramatically.

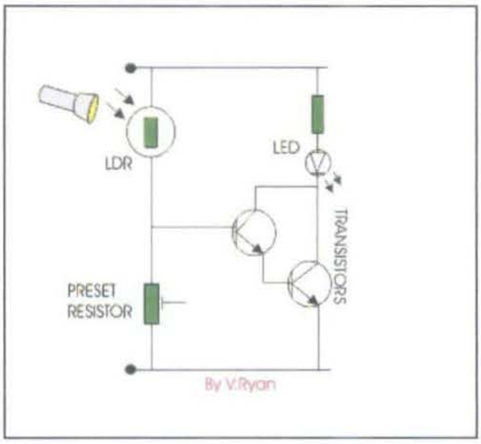


Figure 7: LDR Circuit

The figure above shows a simple LDR circuit with transistors and LEDs. When the light level is low the resistance of the LDR is high. This prevents current from flowing to the base of the transistors. Consequently the LED does not light. However, when light shines onto the LDR its resistance falls and current flows into the base of the first transistor and then the second transistor. The LED lights. The preset resistor can be turned up or down to increase or decrease resistance, in this way it can make the circuit more or less sensitive [4].

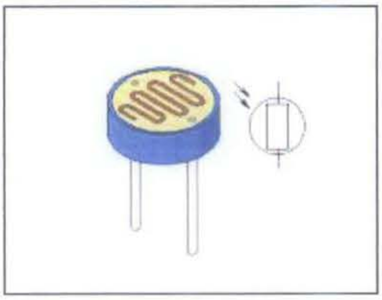


Figure 8: Light Dependent Resistor

2.2 Mechanical Movement

The most important mechanical aspects of this robot is it is constructed so it can perform the basic functions of a robot which is moving forward, reverse and turn. In the early stage of this project, two types of robot are considered which are the legged type and the wheeled type. The author has done studies on the mechanical attachment of these two types of robot and the advantages and disadvantages are considered. For legged robot, it is more sophisticated compared to a wheeled robot. It will required much more control and more motors. It will require a more complex circuit to give it more control over the legs. Legged robot is useful in unlevel terrains where as wheeled is robot require more control to establish stability. The ultimate problem with legged robot is balancing. The robot's body and legs need to be designed to achieve proper balancing to ensure stability during its different movements.

The wheeled robot requires a simpler circuit and minimum number of motors can be used. It is much faster and easier to balance compared to a legged robot. It is less complicated to build and requires less control.

Figure 9: Legged Robot (Left) and Wheeled Robot (Right)

The figure 9 shows a six legged robot (right) and a four wheeled robot (left). Most legged robot is built on six legs to achieve maximum stability and most wheeled robot are built with four wheels. The figure shows the difference of construction complexity between a legged robot and a wheeled robot.

2.2.1 Four Wheeled Robot

A four wheeled robot is suitable to build a pet robot. When deciding to build a four wheeled robot, the author needs to decide the different mechanical wheel connection of the pet robot.

1. Using one motor to connect all wheels.

   In this method, all four wheels are dependent on one another. There will be only one motor controlling all the wheels at one time. The functions therefore will only be limited to move forward and backward. The robot will not be able to make a turn.

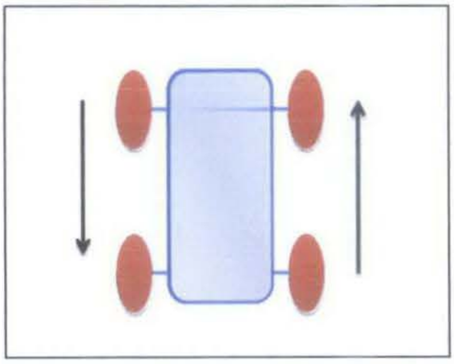


Figure 10: Wheel robot with one motor

2. Using two motors for four wheels, each front and back.

   In this method, one motor is connected to two wheels. Two wheels in the front are connected to one motor and the other two at the back to another motor. The wheels at the back are connected to the motor in a function that it moves forward and backward. As for the front two wheels it is connected in such a way that it can move forward and backward and the angular position of the motor can be changed. This can cause the robot to be able to make a turn.

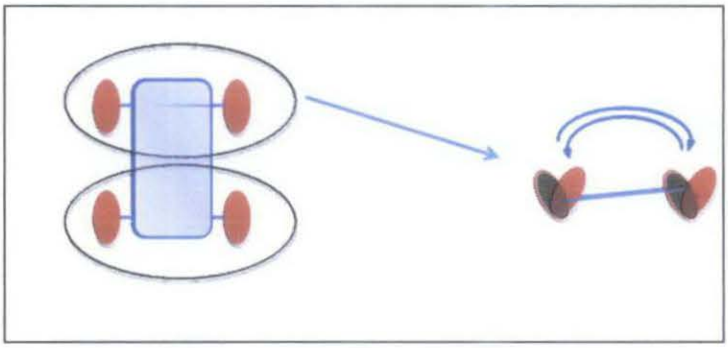


Figure 11: Wheel robot with two motors (front, back)

3. Two motors, side by side

   In this method, one motor is connected to two wheels. Two wheels at one side will be connected to a motor and the other will be connected to a different motor. These two sets of wheels will move independently. By having this connection, the robot is able to move forward, reverse and turn. It can turn to a large radius by having one side of the wheels to move and the other is not.

It is also able to turn in a smaller radius by having one side of the wheels to move forward and the other side to move in reverse.

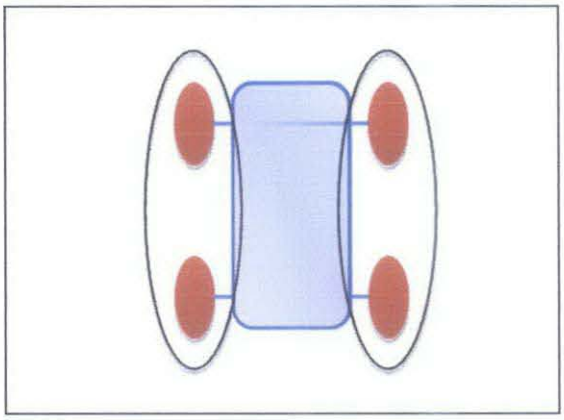


Figure 12: Wheel robot with two motors (left,right)

2.3 Printable Circuit Board (PCB)

A printed circuit board, or PCB, is used to mechanically support and electrically connect electronic components using conductive pathways, or traces, etched from copper sheets laminated onto a non-conductive substrate. Alternative names are printed wiring board (PWB), and etched wiring board. A PCB populated with electronic components is a printed circuit assembly (PCA), also known as a printed circuit board assembly (PCBA).

PCBs are rugged, inexpensive, and can be highly reliable. They require much more layout effort and higher initial cost than either wire-wrapped or point-to-point constructed circuits, but are much cheaper, faster, and consistent in high volume production. Much of the electronics industry's PCB design, assembly, and quality control needs are set by standards that are published by the IPC organization [9].

2.3.1 Patterning (etching)

The vast majority of printed circuit boards are made by bonding a layer of copper over the entire substrate, sometimes on both sides, (creating a "blank PCB") then removing unwanted copper after applying a temporary mask (eg. by etching), leaving only the desired copper traces. A few PCBs are made by adding traces to the bare substrate (or a substrate with a very thin layer of copper) usually by a complex process of multiple electroplating steps [9].

There are three common "subtractive" methods (methods that remove copper) used for the production of printed circuit boards:

1. Silk screen printing uses etch-resistant inks to protect the copper foil. Subsequent etching removes the unwanted copper. Alternatively, the ink may be conductive, printed on a blank (non-conductive) board. The latter technique is also used in the manufacture of hybrid circuits.

2. Photoengraving uses a photomask and chemical etching to remove the copper foil from the substrate. The photomask is usually prepared with a photoplotter from data produced by a technician using CAM, or computer-aided manufacturing software. Laser-printed transparencies are typically employed for phototools; however, direct laser imaging techniques are being employed to replace phototools for high-resolution requirements.

3. PCB milling uses a two or three-axis mechanical milling system to mill away the copper foil from the substrate. A PCB milling machine (referred to as a 'PCB Prototyper') operates in a similar way to a plotter, receiving commands from the host software that control the position of the milling head in the x, y, and (if relevant) z axis. Data to drive the Prototyper is extracted from files generated in PCB design software and stored in HPGL or Gerber file format. [9]

2.3.2 Lamination

Some PCBs have trace layers inside the PCB and are called multi-layer PCBs. These are formed by bonding together separately etched thin boards.[9]

2.3.3 Drilling

Holes, or vias, through a PCB are typically drilled with tiny drill bits made of solid tungsten carbide. The drilling is performed by automated drilling machines with placement controlled by a drill tape or drill file. These computer-generated files are also called numerically controlled drill (NCD) files or "Excellon files". The drill file describes the location and size of each drilled hole.

When very small vias are required, drilling with mechanical bits is costly because of high rates of wear and breakage. In this case, the vias may be evaporated by lasers. Laser-drilled vias typically have an inferior surface finish inside the hole. These holes are called micro vias.

It is also possible with controlled-depth drilling, laser drilling, or by pre-drilling the individual sheets of the PCB before lamination, to produce holes that connect only some of the copper layers, rather than passing through the entire board. These holes are called blind vias when they connect an internal copper layer to an outer layer, or buried vias when they connect two or more internal copper layers and no outer layers.

The walls of the holes, for boards with 2 or more layers, are plated with copper to form plated-through holes that electrically connect the conducting layers of the PCB. For multilayer boards, those with 4 layers or more, drilling typically produces a smear comprised of the bonding agent in the laminate system. Before the holes can be plated through, this smear must be removed by a chemical de-smear process, or by plasma-etch.[9]

CHAPTER 3

METHODOLOGY/PROJECT WORK

The robot was built step- by-step through different sections. Figure 13 shows the flow diagram of the building the robot.

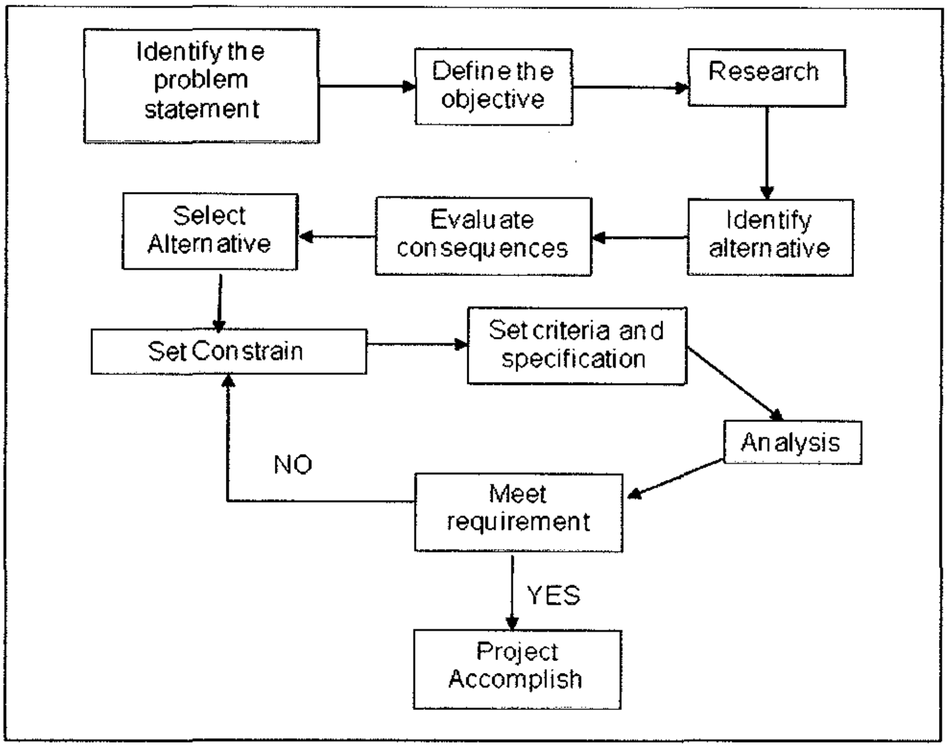


Figure 13: Project Flow Diagram

Process of designing the robot consists of three different sections, the body (mechanical parts), circuit (electrical) and programming (PIC16F84A).

3.1 The Body

The bottom part of the body is very important to determine how the robot will be able to move. The options of weather to build a legged robot or wheeled robot was laid out in theory and after all advantages and disadvantages has been considered, the author has decided to build a wheeled robot. In this project, the auther will not be building the robots body but will use the base of the robot from what is available on the market and reconstruct its circuitry to function as required.

3.1.1 Wheeled Robot

The robot is designed to be a wheeled robot. This is because robots with wheels are able to move faster and has better balancing control compared with legged robots. This is type of mechanical connection is suitable for this project as common pets are usually fast and need to have better stability to be able to move in different ways. The robot will have 4 wheels controlled by two motors. Each motor will control two sets of wheels independently. Two wheels are placed in front and the other two at the back. Figure 14 shows the motor connections to the wheels:

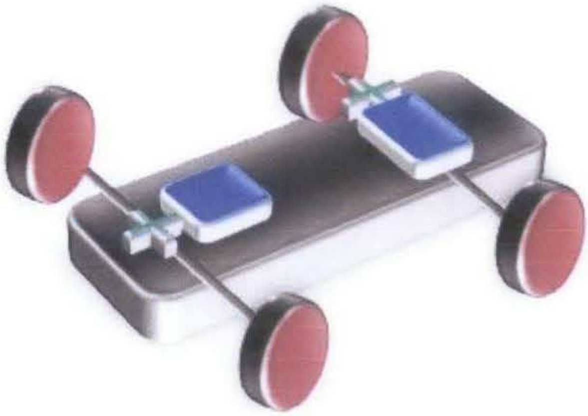


Figure 14: Motor Connection

The mechanical connection of the back wheels to its motor will enable the robot to move forward and reverse. The wheels will move forward when the motor is connected to the positive voltage and reverse when the motor is connected to negative voltage. The positive and negative voltage input to the motor can be controlled by the PIC. The mechanical connection of the front motor to its wheels will enable the wheels to change angle and thus allowing the robot to move right or left. Whether the wheels are to turn left or right is once again controlled by the positive or negative input voltage to the motor.

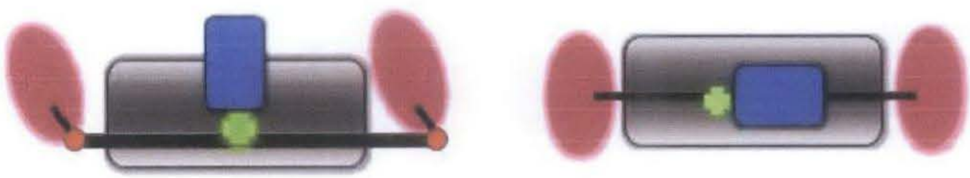


Figure 15: Front Wheels and Back Wheels

3.2 Designing the Circuit

The first step is to design the circuit for the PIC16F84A. This will be the main circuit.

The secound step is to build the circuit for the sensors

Finally a circuit for light sensor to enable the robot to detect light

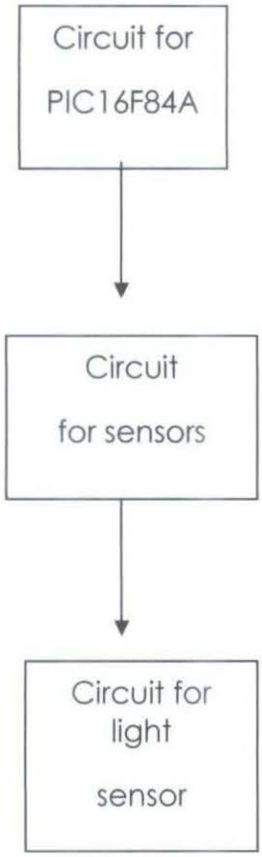


Figure 16: Designing the Circuits flow Diagram

3.3 Programming the PIC

The first step is to do research and self learning on how to program a PIC using C Compiler

The Code is then designed in the software according to the functions. It is compiled, and errors are checked and corrected.

The code is then implemented to the PIC16F877

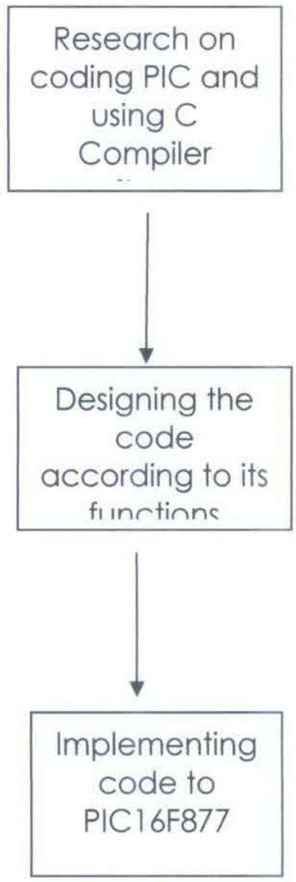


Figure 17: Programming the chip flow diagram

3.3.1 PIC C Compiler Software

There are many soft wares used to program a PIC. In this project, the PIC C compiler Software is used to program the PIC 16F877. This software is chosen because it is easy to work with and is user friendly.

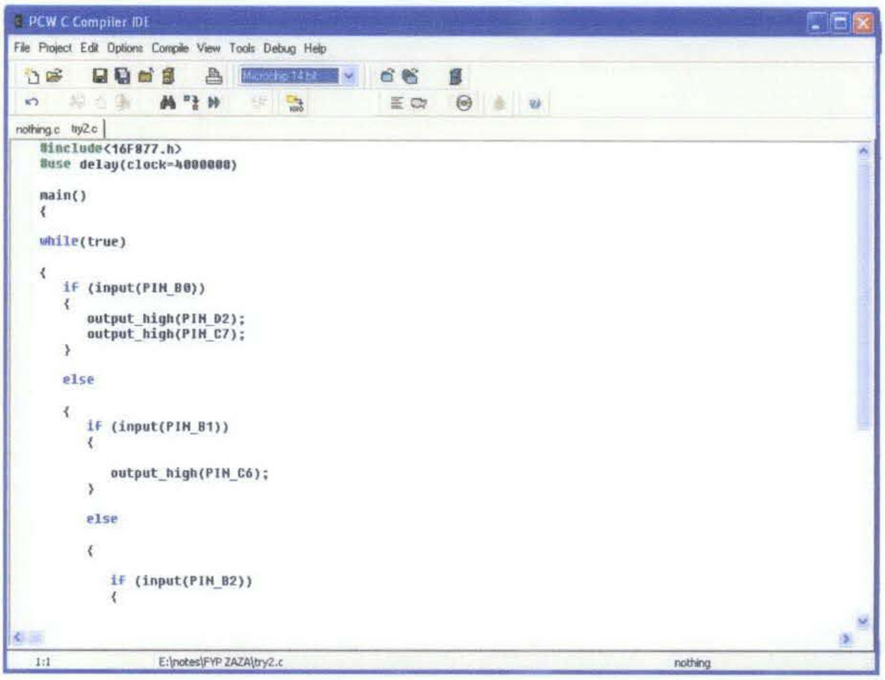


Figure 18: C Compiler Software

The figure 18 shows the interface of PIC C Compiler. The code is written and designed in the workspace area.

After the code is completed, it is compiled. Codes with errors are then analyzed and corrected.

The successfully compiled codes need to be tested whether it functions properly before it is implemented onto the chip. This is done by using software called PIC

Simulator IDE. The code is loaded into this software and the simulation. The inputs and outputs of the microcontroller can be viewed from the virtual microcontroller view window and any errors from the coding can be detected.

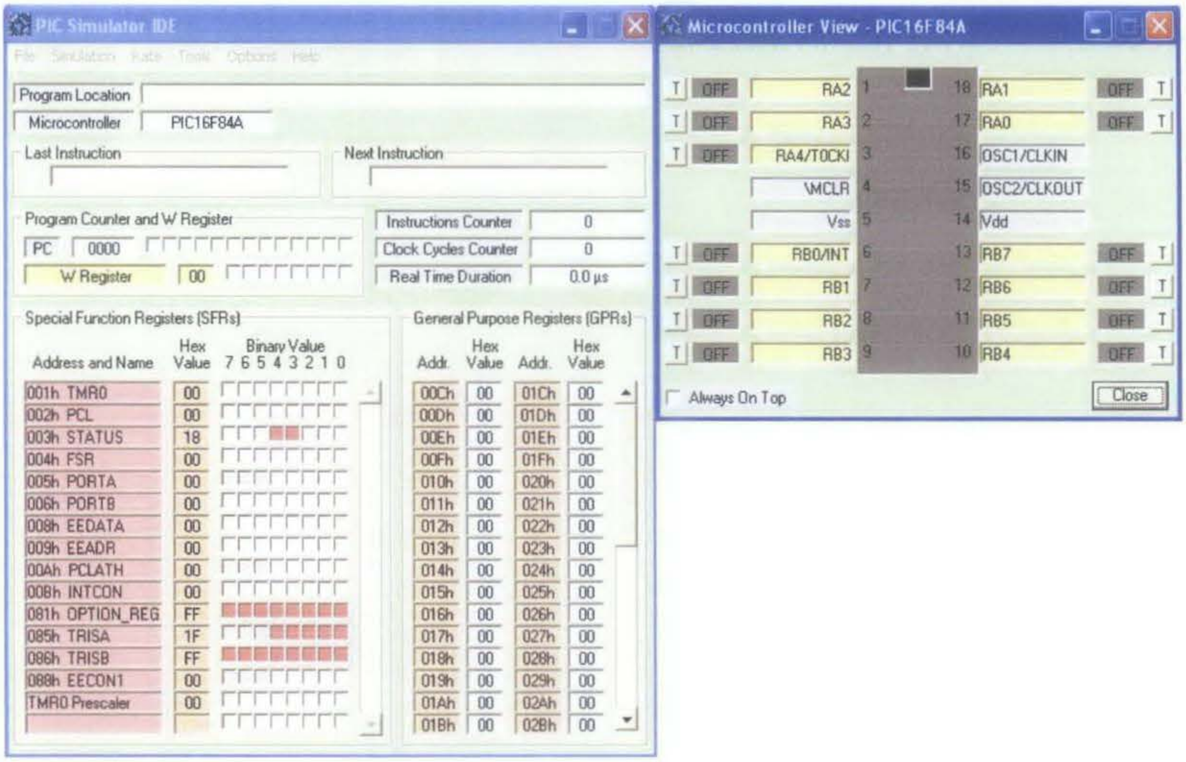


Figure 19: PIC Simulator IDE

Figure 19 shows the interface of PIC simulator IDE. By using the PIC simulator IDE, errors can be detected easily and time can be saved. This is because the codings are checked before implementing them onto the PIC.

After the code is checked with the PIC Simulator IDE, it now can be implemented onto

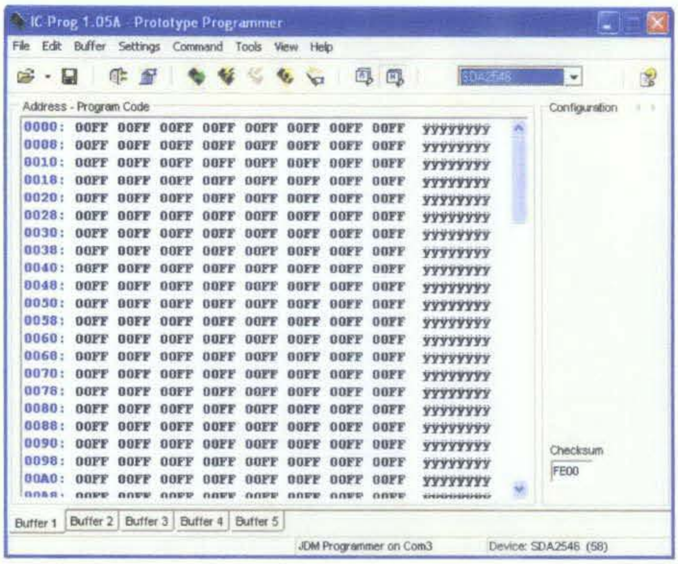


Figure 20: PIC Programming Software

the microcontroller. This is done by using a PIC Programming Software (as shown in figure 20) and a universal PIC programming device. The code that have been compiled and checked is loaded to the PIC programming software. The chip is inserted in the PIC programming device as shown below:

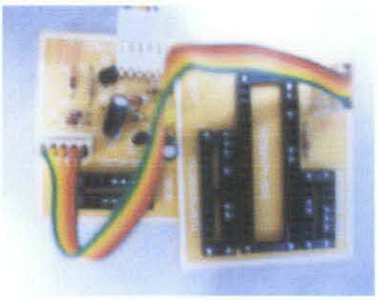


Figure 21: PIC Programming Device

The microcontroller chip is now programmed and is ready to be used in the main circuit of the robot.

3.4 Implementing the Circuits

After collecting all the components for the circuits, it can be implemented to the circuit board. For initial implementation for designing and testing, the components are soldered onto a simple breadboard. Although this circuit board is functioning correctly and successfully, it is not a very stable circuit. Short circuits can occur because the cuprum line is connected all over the board. This problem can be solved by implementing the circuits onto a printable circuit boards (PCB)

3.4.1 Printable Circuit Board Method

Creating a PCB involves 2 main process. The first process involves using a software called Eagle 4.13 Light. The schematic drawing of the designed circuit is first created using this software. The figure 22 is an example of a schematic drawing using Eagle software:

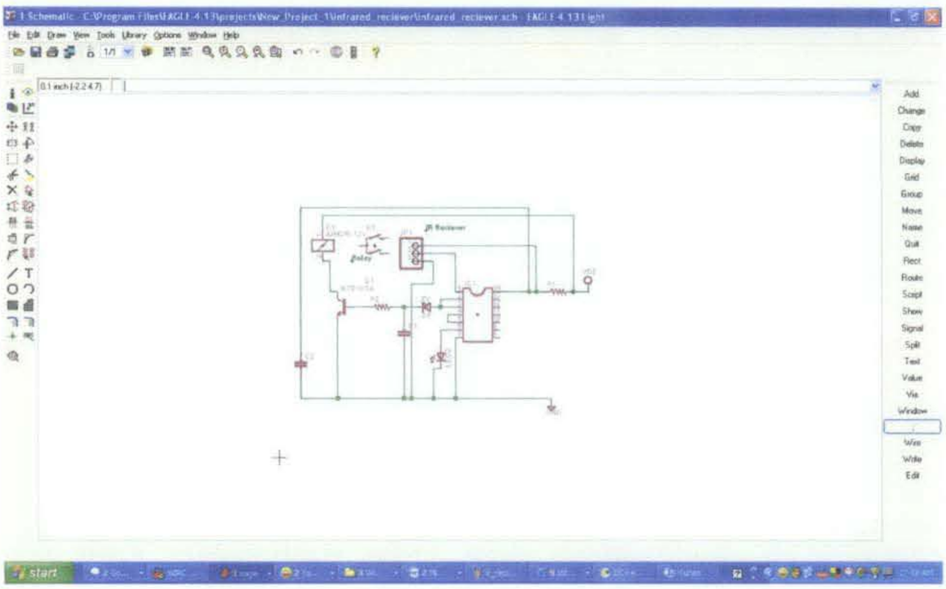


Figure 22: Schematics Drawings in Eagle 4.13 Light Software

After all the components are in place, the next step is to convert the drawings to a virtual circuit board as shown below. Here all the components will be arranged in a way where no wires can cross each other and cause a short circuit. The software will automatically create a route for the path of the cuprum line which will be etched later.

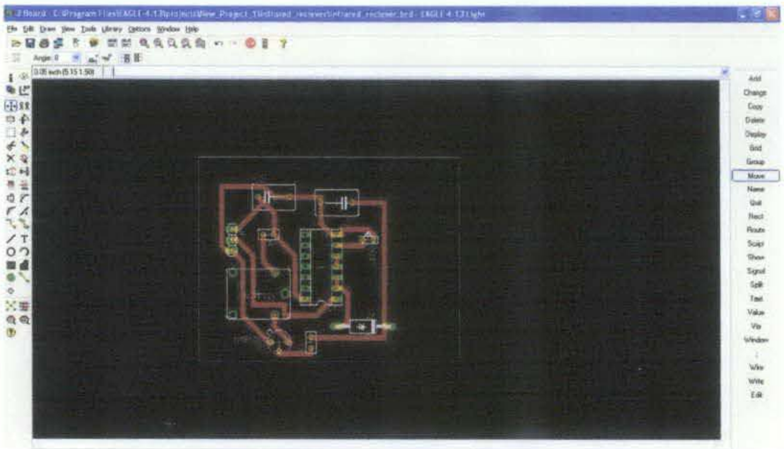


Figure 23: PCB Route Designing

The second process involves drilling holes and etching cuprum onto the circuit board. This will be done at the lab assisted by the technician on duty.

CHAPTER 4

RESULTS & DISCUSSIONS

4.1 RESULTS

After considering the theories and methods of each section of the robot, the author has implemented and design the to robot to meet the objective of this project.

4.1 .1 Robot Function Flow Diagram

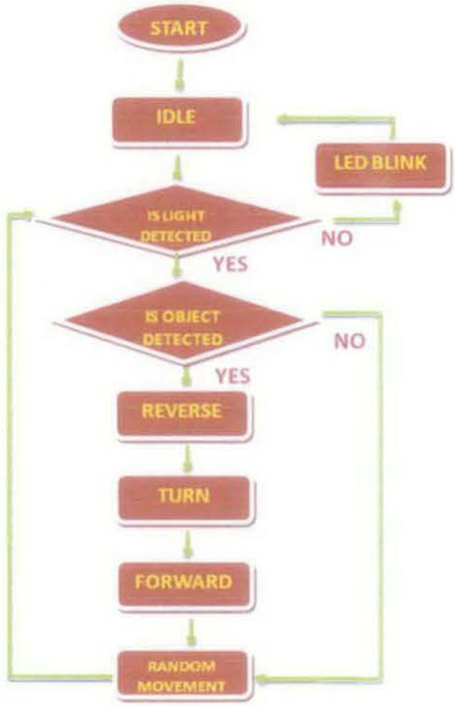


Figure 24: Function Flow Diagram

Figure 24 shows a flow diagram designed to lay out the functions of the robots. It consists of the main and basic functions of the robot. The codings designed will be base on this flow diagram.

Firstly, the robot is started. This consists of turning on the power switch of the robot. An LED indicator which is placed on the main circuit will light up to indicate that it has been powered up.

Next, the robot will be in idle mode. This means that the robot will delay any movements for a few seconds before performing any functions. This mode is important to ease the flow of the next step.

The robot will now check for the first condition that is to check weather there is any light. If there is light, then the robot will go to a mode where a number of LEDs will blink. The LEDs will blink until there is light turned on and move on to the next step function.

If there is no light detected, the robot will then check for the next condition which is to check weather there is an object blocking its way in front of it. If there is an object, the robot will reverse, turn and move forward avoiding the object.

If there is no object detected, the robot will then go to having random movements mode. In this mode, the robot will constantly check weather there is light detected and object detected and interrupt the random movement when one of the condition is met.

4.1.2 Main Circuit

4.1.2.1 Microcontroller PIC16F84A Circuit

Figure below shows the pins of the PIC 16F84A. Pin 14 (VDD) and pin 4 (MCLR) is connected to 5V. Pin 5 (VSS) is connected to ground. Pin 15 and 16 (OSC1&2) is connected to a 4MHz two legged clock which will run the circuit. The outputs that will connect to the motor circuit are B4,B5, B6, and B7. Pin B4 and B5 is connected to the back motor. Pin B5 will allow negative voltage to flow to the motor which will make the wheels turn in reserve mode. Pin B4 will allow positive voltage to flow to the motor which will make the wheels turn in forward mode. Pin B6 and B7 is connected to the front motor which will determine weather the robot is to turn left or right. Pin B6 will allow positive voltage to flow through the motor and enables the wheel to turn right while pin B7 enables flow of negative voltage through the motor and enables the wheel to turn left. An LED is connected to pin B3 as an 'ON' indicator. This LED will light up when the circuit is connected to a power supply.

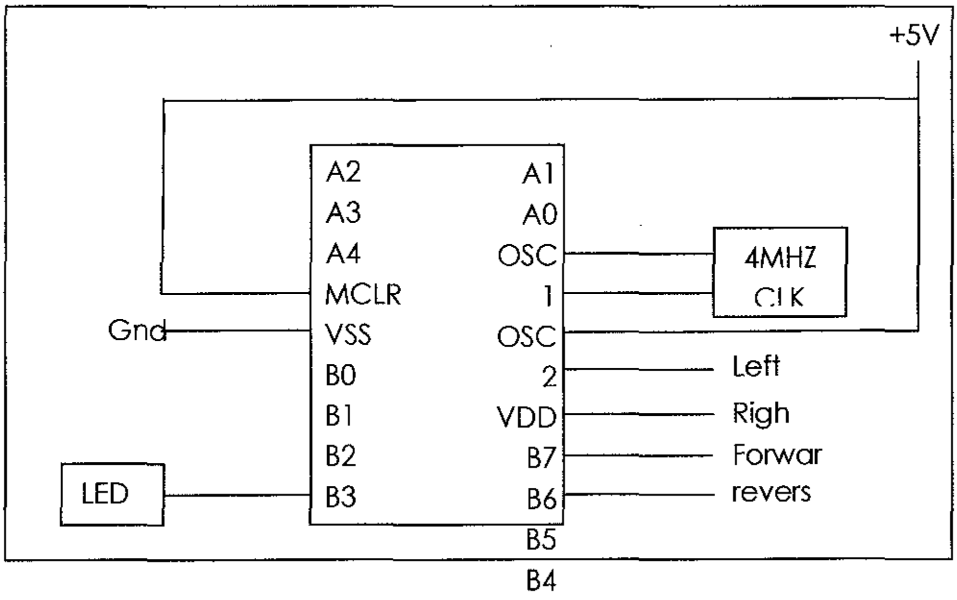


Figure 25: Pin Connection of PIC 16F84A

The circuit diagram is then implemented to a circuit board. All the components are soldered accordingly. Figure below shows the completed main circuit:

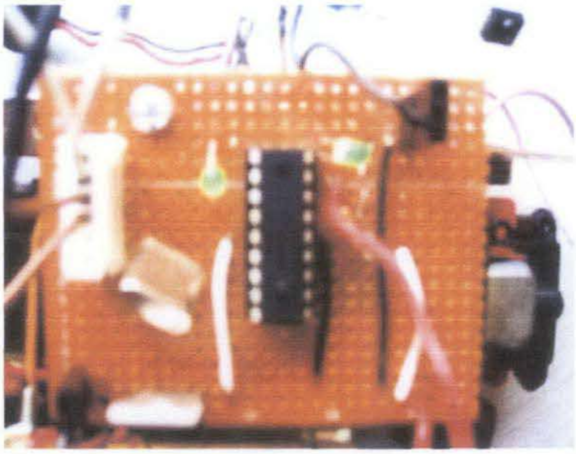


Figure 26: Main Circuit

When the circuit is successfully connected, a simple program is encoded to the PIC to test the movement of the robot. Below shows a simple coding used to test the robot.

```
main()
{
while(true)
{
    output_high(PIN_B3);          //LED ON indicator lights up
    output_low(PIN_B4);           //reverse mode off
    output_high(PIN_B5);          //forward mode on
    output_low(PIN_B6);           //right mode off
    output_low(PIN_B7);           //left mode off
    delay_ms(5000);              // delay for 5 secs

    output_high(PIN_B3);          //LED ON indicator lights up
    output_high(PIN_B4);          //reverse mode on
    output_low(PIN_B5);           //forward mode off
    output_low(PIN_B6);           //right mode off
    output_low(PIN_B7);           //left mode off
    delay_ms(5000);              //delay for 5 secs

    output_high(PIN_B3);          //LED ON indicator lights up
    output_low(PIN_B4);           //reverse mode off
    output_low(PIN_B5);           //forward mode off
    output_high(PIN_B6);          //right mode on
    output_low(PIN_B7);           //left mode off
    delay_ms(5000);              //delay for 5 secs

    output_high(PIN_B3);          //LED ON indicator lights up
    output_low(PIN_B4);           //reverse mode off
    output_low(PIN_B5);           //forward mode off
    output_low(PIN_B6);           //right mode on
    output_high(PIN_B7);          //left mode off
    delay_ms(5000);              //delay for 5 secs
}
}
```

After the code is implemented to the pic, the robot seems to function correctly according to the code. This shows the implementation of the main circuit is successful.

4.1.3 PIC Programming

Below is the coding that has been redesigned. Sub functions has been included inside the previous coding to simplify it. An initial start-up sequence was also inserted to check the initial condition of the robot when it is turned on before starting any other functions. The initial start-up sequence will check each forward, reverse, left and right mode of the robot is fully functioning.

```
#include<16F84a.h>
#use delay(clock=4000000)
#fuses  XT,NOPROTECT,NOWDT

//all subfunctions
void all_pause(void);
void delay_gap(void);
void intervention(void);

//start main function
main()
{
//INITIATE START-UP SEQUENCE

    output_high(PIN_B3);        //B3 is POWER indicator LED ON
    output_low(PIN_B4);         //B4 is LEFT
    output_high(PIN_B5);        //B5 is RIGHT
    output_low(PIN_B6);         //B6 is FORWARD
    output_low(PIN_B7);         //B7 is REVERSE
    delay_ms(200);            //DELAY BEGIN AFTER STARTUP

  output_low(PIN_B3);         //B3 is POWER indicator LED ON
    output_high(PIN_B3);        //B3 is POWER indicator LED ON
    output_low(PIN_B4);         //B4 is LEFT
    output_low(PIN_B5);         //B5 is RIGHT
```

```
    output_low(PIN_B6);         //B6 is FORWARD
    output_low(PIN_B7);         //B7 is REVERSE
    delay_ms(200);            //DELAY BEGIN AFTER STARTUP

  output_high(PIN_B3);         //B3 is POWER indicator LED ON
    output_low(PIN_B3);         //B3 is POWER indicator LED ON
    output_low(PIN_B4);         //B4 is LEFT
    output_low(PIN_B5);         //B5 is RIGHT
    output_high(PIN_B6);         //B6 is FORWARD
    output_low(PIN_B7);         //B7 is REVERSE
    delay_ms(100);            //DELAY BEGIN AFTER STARTUP

    output_low(PIN_B3);         //B3 is POWER indicator LED ON
    output_low(PIN_B3);         //B3 is POWER indicator LED ON
    output_low(PIN_B4);         //B4 is LEFT
    output_low(PIN_B5);         //B5 is RIGHT
    output_low(PIN_B6);         //B6 is FORWARD
    output_high(PIN_B7);         //B7 is REVERSE
    delay_ms(100);            //DELAY BEGIN AFTER STARTUP
```

This coding will create an initiate start up sequence. In this sequence, the robot will first check weather all of the wheels movements are working. It will command the back wheels to go forward, then to go reverse. This is to check the functionality of the back wheels. Then it will check the functionality of the front wheels and command the wheels to go right first and then left.

```
//START LOOP
while(true)
{
 output_high(PIN_B3);         //B3 is POWER indicator LED ON
if (INPUT(PIN_B2))
{
intervention();
}
else
{
   //FORWARD LEFT FOR 5 MILLISECONDS
   output_high(PIN_B4);
   output_low(PIN_B5);
   output_high(PIN_B6);
   output_low(PIN_B7);


   delay_ms(700);
```

```
all_pause();

    //REVERSE RIGHT FOR 5 MILLISECONDS
    output_low(PIN_B4);
    output_high(PIN_B5);
    output_low(PIN_B6);
    output_high(PIN_B7);
    delay_ms(400);

all_pause();

    //FORWARD LEFT FOR 5 MILLISECONDS
    output_high(PIN_B4);
    output_low(PIN_B5);
    output_high(PIN_B6);
    output_low(PIN_B7);
    delay_ms(700);
```

This is the loop function that will make the robot go forward for 5 milisecounds and stop for 5 milliseconds repeating constantly. This will make the robot move slower that just making it go forward without stopping for a few seconds. The speed of the robot is controlled by using this coding method.

```
void delaygap()
{
    output_high(PIN_B3);
    output_low(PIN_B4);
    output_low(PIN_B5);
    output_low(PIN_B6);
    output_low(PIN_B7);
    delay_ms(200);
}
```

This is the sub function for stopping all movements for 200 milliseconds.

```
void all_pause()
{   output_low(PIN_B4);
    output_low(PIN_B5);
    output_low(PIN_B6);
    output_low(PIN_B7);
    delay_ms(500);
}
```

This is the sub function for pausing all movements for 5 miliseconds.

```
void intervention()
{
    output_high(PIN_B4);
    output_low(PIN_B5);
    output_low(PIN_B6);
    output_low(PIN_B7);
    delay_ms(200);

    output_low(PIN_B4);
    output_high(PIN_B5);
    output_low(PIN_B6);
    output_low(PIN_B7);
    delay_ms(200);}
```

This is the sub function that will put a high output to the forward mode of the robot.

4.1.4 Light Sensor Circuit

When the LDR detects light, it will give a signal to the PIC and the PIC will command the robot to perform its normal function such as moving forward, backward and turning under certain conditions. But if the photocell does not detect any light, the PIC will command the robot to be in idle mode while blinking the LED. Below is the diagram of the light sensor circuit:

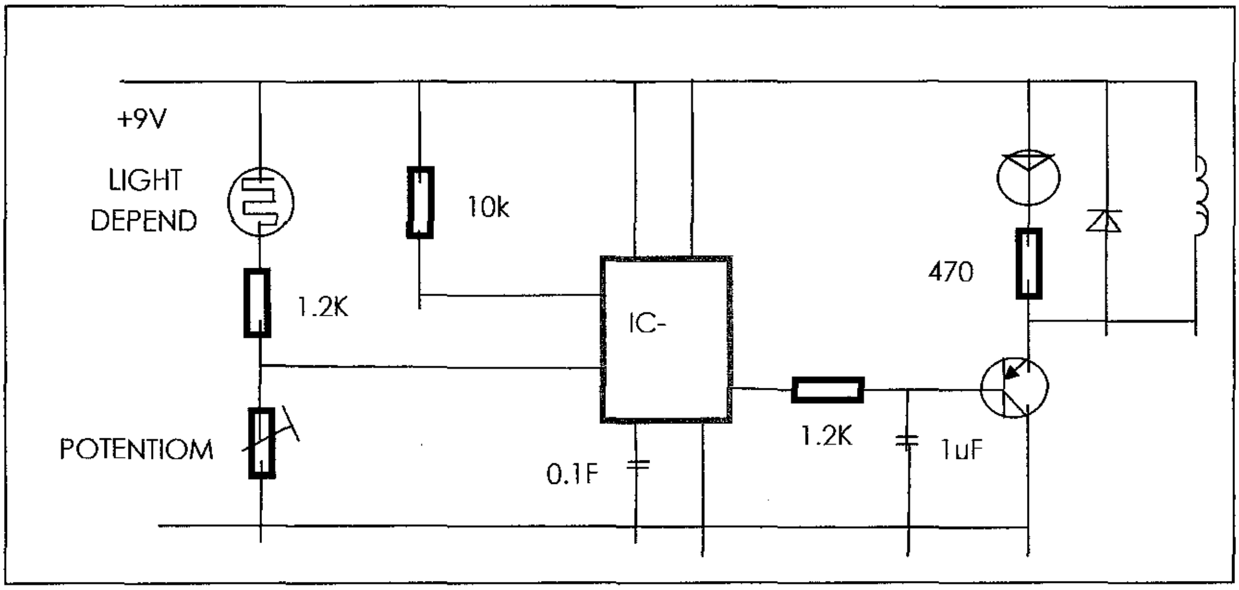


Figure 27: Light Sensor Circuit

When LDR detects light, a signal will be sent to a relay. This signal will trigger the relay. The output of the relay is then connected to the main circuit which contains the pic. Pin B2 is assigned as an input to the pic and is then connected to the output of the relay. By this connection, when ever the LDR detects light, a an input signal will be sent to the pic. By using programming, the pic can be set to control the function that the robot should perform when there light. Below is a simple code program that shows an output function when the LDR sends a signal to the input pic indicating there is light. This function uses an 'If Else' function in the programming. If there is an input signal in pin B2, the output signal B4 will be high. Else, B4 will be low. This means if there is light, the back wheels will move forward. If there is not light, the robot will not move at all.

```
main()
{while(true)
{
 output_high(PIN_B3);
  if (input(PIN_B2))                  //if there is light
 {
   output_high(PIN_B4);        //forward mode on
  }
 else                               //if there is no light
 {
     output_low(PIN_B4);       //forward mode off
  }
}
}
```

After implementing the following program to the circuit, the robots seems to function as instructed in the coding. this shows the circuit successfully detects the input from the LDR when there is light and gives command to the output to perform certain functions.

4.1.5 Infrared Transmitter and Receiver

In order for the robot to detect object obstacles, an infrared circuit will be implemented to the robot. The infrared circuit consists of sender circuit which contains the infrared emitter diode and receiver circuit which consists of infrared receiver device. Below is a circuit of the infrared circuit sender and receiver.

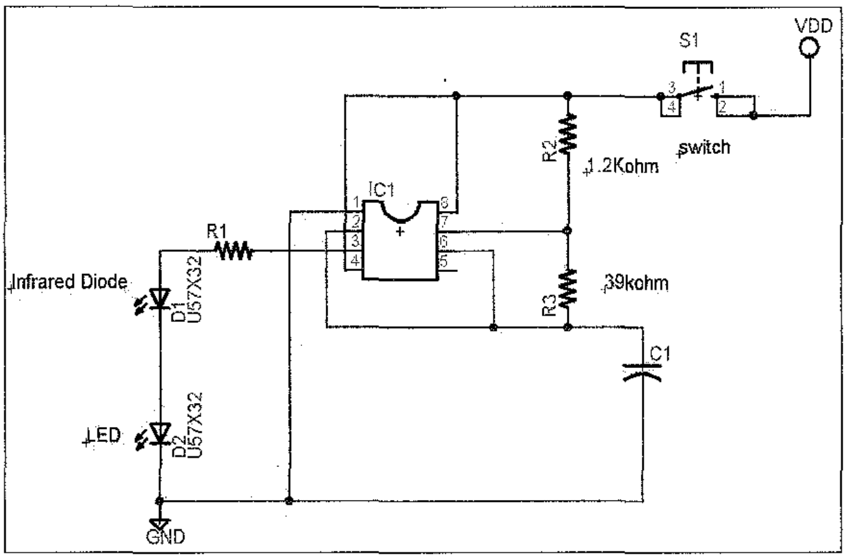


Figure 28: Infrared Transmitter Circuit

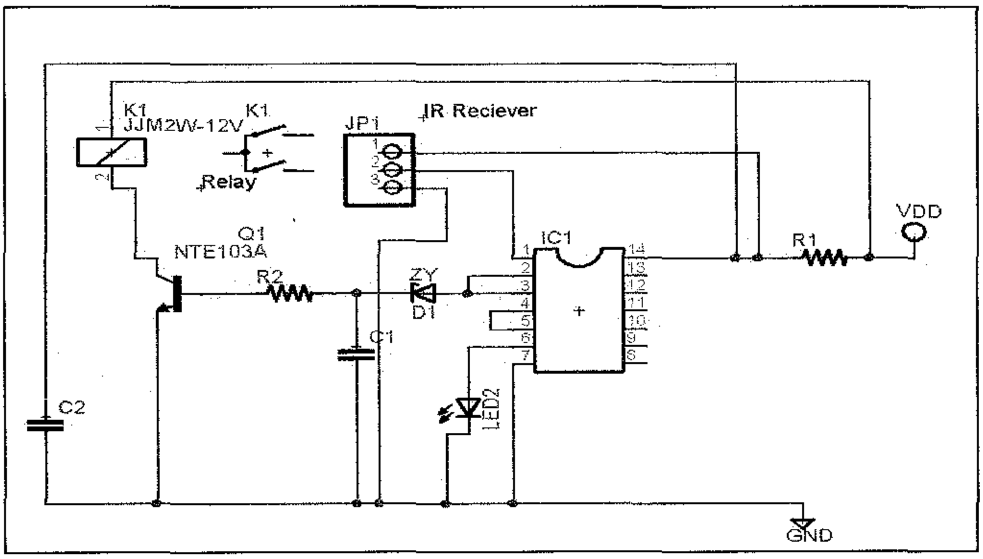


Figure 29: Infrared Receiver Circuit

Both of these circuits will be placed in front of the robot. It is powered up by an on switch placed at the side of the robot. The transmitter circuit contains the infrared diode and will emit the infrared wave. The infrared wave can not be seen by the human eye. There for an LED in placed to this circuit to indicate that the infrared diode is emitting the waves. When there is an object in front of it, the infrared wave will hit the object, reflecting it and making it bounce back hitting the receiver circuit which contains the infrared receiver component. This will then trigger the relay in the receiver circuit which acts as a switch allowing current to flow through and signal to pass by to the PIC. The PIC will then send a signal to the appropriate output to change the movement of the robot to avoid from bumping into the object.

4.1.6 Overall circuit connection

When all the circuits are finished, they are all connected and integrated together to form a robot. Below is a diagram of the overall connection:

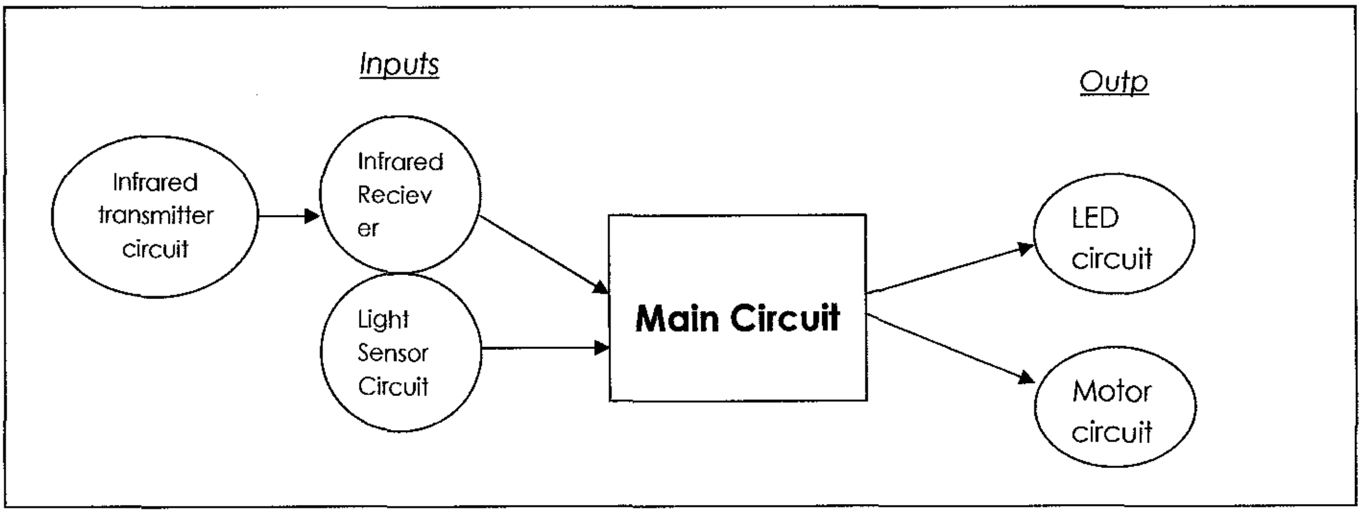
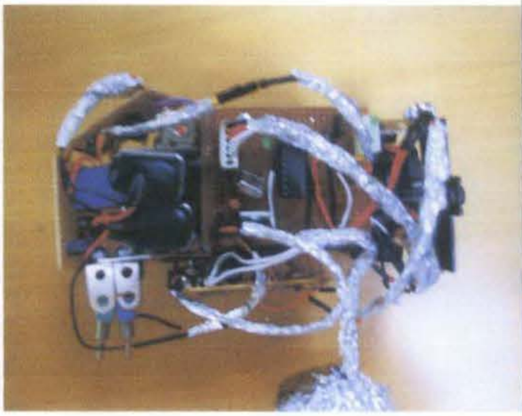
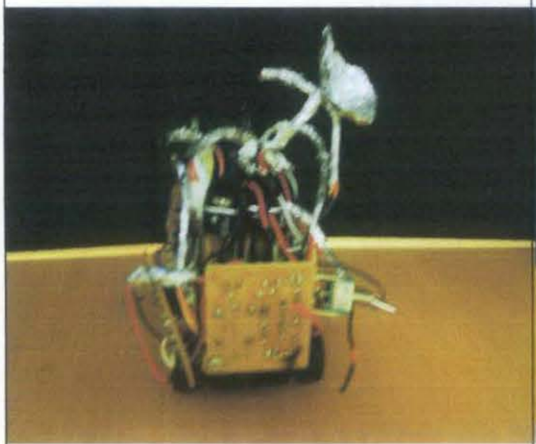
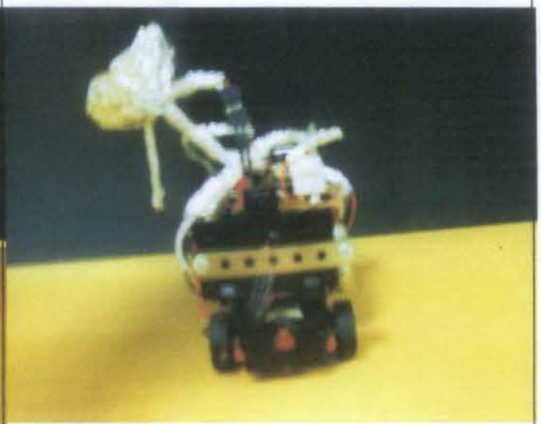


Figure 30: Overall Circuit Connection

All inputs and outputs will be connected to the main circuit. The input circuits are the infrared receiver circuit and the light sensor circuit. The infrared transmitter circuit is connected to the infrared receiver circuit. The output circuits are the LED circuits and the motor circuit which will then be connected to the wheels.

4.1.6 Pictures of Pet Robot

	
Top View	Left View
	
Back View	Front View

4.2 Discussion

4.2.1 Speed Control

When the first code was implemented to the robot, the author noticed that the robot was moving very fast with a constant speed. The robot like pets, must be able to move slowly or very fast every now and then. The speed needs to vary differently to give some characteristics an essence of life. It is also important to slower down the speed in situations where a condition need to be checked. For example, the robot needs to move slowly before checking for object obstacles else it will end up bumping into it. To control the speed, the author needs to implement a coding to the PIC which will slow down the signal sent out to the motor which controls the wheels. Each time a high signal is sent to an output of the wheels, it will be followed by a low signal to the same output and delay it for a certain time. This is repeated for a number of times. For example, the speed control for forward mode is as below:

```
output_high(PIN_B5);        //high output to forward pin
delay_ms(100);              //forward mode for 0.1 secs
output_low(PIN_B5); //low output to forward pin
delay_ms(100);              //delay stop for 0.1 secs
```

Implementation of this code to the chip will cause the forward wheels to run for 0.1 second and stop running for 0.1 second. When this cycle is repeated constantly, it will make the robot seem to be moving smoothly but at a slower rate then by just making the signal high all the time.

4.2.2 Circuit's Stability and Sensitivity

During troubleshooting, all the components were connected onto a regular simple breadboard. After the circuits are finalized, it is then transferred onto a veraboard which is a type of board that has cuprum connected all over the circuit. This board requires soldering of the components onto the board. When this is done, all the circuits are then put together and tested. The author later notices that the circuit is not very stabile and is very sensitive as the connection of cuprum under the board can easily cause a short circuit which can cause failure to the robot's functions and can also cause chips and components to burn. This is later avoided by transferring these circuits on to a printable circuit board which has better connections and the probability of a short circuit is less compared to a vera board. The main circuit remains connected to a veraboard. This is because by using this board, additional inputs and outputs can easily be implemented without changing the main circuits. This makes the robot more flexible and provides a wider range of possibilities to its functions.

4.2.3 Light sensor sensitivity

A light sensor is implemented onto the robot to enable it to detect light. A variable resistor is placed in the circuit to adjust the sensitivity of the LDR. Although the variable resistor helps adjust the robot to react to how much brightness and darkness, there is still a problem when the robot is a situation where it's not so bright and not so dark. At this point the robot will confuse and start to do movements in dark mode and bright mode alternately. The robot can only work in a condition where it is purely bright or purely dark.

4.2.4 Reprogrammable chip

This robot uses a microcontroller PIC 16F84A. This PIC can be reprogrammed over and over again. This feature allows can us to change the movement of the robot from time to time. Changes can be made easily by just altering the codes and implement them back onto the PIC. The robot is more flexible on its functions and features. It can be enhanced by adding more inputs and outputs. Additional circuits could be easily added without building a new main circuit or changing other existing circuits. More intelligence could be added that enables it to meet certain goals depending on its purposes.

CHAPTER 5

CONCLUSION & RECOMMENDATIONS

5.1 Conclusion

The robot is now able to perform all the basic functions such as moving forward, reverse and turn. The main circuit which contains the microcontroller PIC 16F84A is built and implemented onto the robot that controls all functions. it is connected to the motor circuit and is able to move the mechanical movement of the robot. This circuit is also able to detect input signals by other external circuits and trigger the appropriate output according to the codes designed. The codes are designed to move the basic movements of the robot and conditions have been implemented to enable the robot to make decisions according to certain inputs. The circuit of the light sensor is designed, built and implemented onto the robot to enable it to detect light. When light is detected, a signal will be sent to the PIC of the main circuit and the PIC will instruct the robot to react to it. The robot is able to differentiate between brightness and darkness. The infrared sensor has also been designed, built and implemented onto the robot. it is placed in front of the robot to detect object in front of it. The robot is able to detect object blocking its way about one feet away. All the circuits to achieve the objective of this project is now completed and running successfully. The robot is now able to move forward, reverse, turn, detect and avoid object obstaclés, move randomly and sense light. Enhancement of the coding will be done which will add more feature to the robot and make it more similar to a pet robot. Once this coding is completed, the overall coding can be finalize and thus implemented onto the robot to finalize the final product of the Pet Robot.

5.2 Recommendations

The objective of this project is to built a basic robot which imitates the behavior of a pet. The robot is built through designing basic circuits to perform basic functions. This topic of a pet robot is very wide and general. The functions and purposes of this pet robot is not limited to a specific standard. Improvements and enhancements can be easily done by adding more circuits to implement more inputs and outputs of the robot making it more sophisticated and interesting. The PIC used is a PIC16f84A which is a reprogrammable chip. This enables us to alter the codings of the robot to change its movements and add inputs and output easily. The characteristics and behavior of this robot can be erased and a new personality can be implemented to it. Other features such as sound detecting and sound making can be implemented later to make the pet robot more real and give it more life.

This PIC 16f84A can be changed to a PIC16F877 which has the same function but with more input and output pins. This enables us to add more circuits and widen the robots functions and features by using the same coding designed in PIC 16f84A.

More sensors can be added to the robot to give it more intelligence to on avoiding bumping into objects. The infrared sensor can be implemented not only at the front of the robot but also at the back to ensure the robot does not crash into anything while it is in reverse mode.

REFERENCES

1. http://en.wikipedia.org/wiki/Voltage_regulator
2. http://en.wikipedia.org/wiki/Crystal_oscillator
3. http://en.wikipedia.org/wiki/H-bridge
4. http://www.technologystudent.com/elec1/ldr1.htm
5. www.reuk.co.uk
6. en.wikipedia.org/wiki/Infrared
7. http://science.hq.nasa.gov/kids/imagers/ems/infrared.html
8. http://www.societyofrobots.com/schematics_infraredemitdet.shtml
9. http://en.wikipedia.org/wiki/Circuit_board
10. Macmillan English Dictionary Fundamental Student Edition
11. http://links999.net/robotics/robots/robots_introduction.html
12. http://inventors.about.com/library/inventors/blrobots.htm
13. http://www.geocities.com/siliconvalley/2072/3pinvolt.htm

APPENDIX

PIC16F84A

18-pin *Enhanced* FLASH/EEPROM 8-Bit Microcontroller

High Performance RISC CPU Features:

- Only 35 single word instructions to learn
- All instructions single-cycle except for program branches which are two-cycle
- Operating speed: DC - 20 MHz clock input
  DC - 200 ns instruction cycle
- 1024 words of program memory
- 68 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
  - External RB0/INT pin
  - TMR0 timer overflow
  - PORTB<7:4> interrupt-on-change
  - Data EEPROM write complete

Peripheral Features:

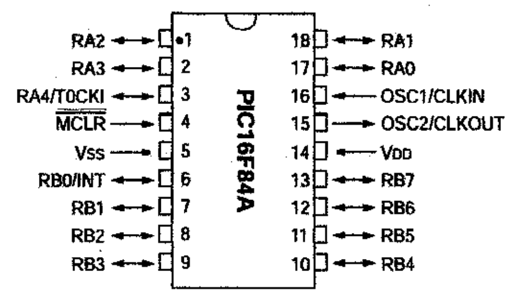
- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - 25 mA sink max. per pin
  - 25 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features:

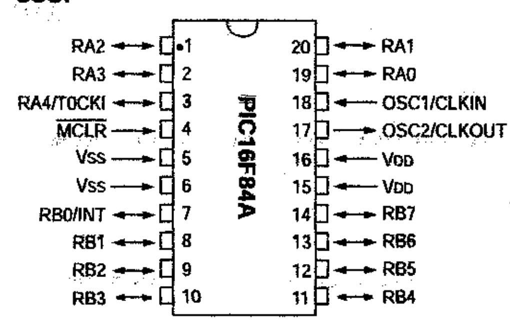
- 10,000 erase/write cycles *Enhanced* FLASH Program memory typical
- 10,000,000 typical erase/write cycles EEPROM Data memory typical
- EEPROM Data Retention > 40 years
- In-Circuit Serial Programming™ (ICSP™) - via two pins
- Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options

Pin Diagrams

PDIP, SOIC



SSOP



CMOS Enhanced FLASH/EEPROM Technology:

- Low power, high speed technology
- Fully static design
- Wide operating voltage range:
  - Commercial: 2.0V to 5.5V
  - Industrial: 2.0V to 5.5V
- Low power consumption:
  - < 2 mA typical @ 5V, 4 MHz
  - 15 µA typical @ 2V, 32 kHz
  - < 0.5 µA typical standby current @ 2V

SourceBoost Technologies

BoostC C Compiler
for PICmicro
Reference Manual

Index

BoostC compiler

Introduction

Thank you for choosing BoostC. **BoostC** is our next generation C compiler that works with PIC16, PIC18 and some PIC12 processors.

This ANSI C compatible compiler supports features like source level symbolic debugging, signed data types, structures/unions and pointers.

The **BoostC** compiler can be used within our **SourceBoost IDE** (Integrated Development Environment), or it can be integrated into Microchip MPLAB.

BoostC Compiler specification

Base Data types

Size	Type name	Specification
1 bit	*bit, bool*	boolean
8 bit	*char*	signed, unsigned
16 bit	*short, int*	signed, unsigned
32 bit	*long*	signed, unsigned

Special Data types

- **Single bit** - single bit data type for efficient flag storage.
- **Fixed address** - fixed address data types allow easy access to target device registers.
- **Read only** - code memory based constants.

Special Language Features

- References as function arguments.
- Function overloading.
- Function templates.

Code Production and Optimization Features

- **ANSI 'C' compatible** - Makes code highly portable.
- Produces optimized code for both PIC16 (14bit core) and PIC18 (16bit core) targets.
- **Support for Data Structures and Unions** - Data structures and arrays can be comprised of base data types or other data structures. Arrays of base data types or data structures can be created.
- **Support for pointers** - pointers can be used in "all the usual ways".
- **Inline Assembly** - Inline assembly allows hand crafted assembly code to be used when necessary.
- **Inline Functions** - Inline functions allows a section of code to be written as a function, but when a reference is made to it the inline function code is inserted instead of a function call. This speeds up code execution.

- **Eliminates unreachable (or dead) code** - reduces code memory usage.
- **Removal of Orphan (uncalled) functions** - reduces code memory usage.
- **Minimal Code Page switching** - code where necessary for targets with multiple code pages.
- **Automatic Banks Switching for Variables** - allows carefree use of variables.
- **Efficient RAM usage** - local variables in different code sections can share memory. The linker analyzes the program to prevent any clashes.
- **Dynamic memory management.**

Debugging features

- **Source Level and Instruction Level Debugger** - linker Generates COF file output for source level debugging under SourceBoost D[ebugger].
- **Step into, Step over, Step out and Step Back** – these functions operate both at source level and instruction level.
- **Multiple Execution Views** - see where the execution of the code is at source level and assembly level at the same time.
- **Monitoring variables** - variables can be added to the watch windows to allow their values to be examined and modified. There is no need to know where a variable is stored.

Full MPLAB integration

- Use of the MPLAB Project Manager within MPLAB IDE.
- Creation and Editing of source code from within MPLAB IDE.
- Build a project without leaving MPLAB IDE environment.
- Source level debugging and variable monitoring using:
    - **MPLAB simulator;**
    - **MPLAB ICD2;**
    - **MPLAB ICE2000.**

Librarian

- Allows generation of **library files** - this simplifies management and control of regularly used, shared code.
- Reduce compilation time - using library files reduces compilation time.

Code Analysis

- **Call tree view** - **SourceBoost IDE** can display the function call tree.
- **Target Code Usage** - From the complete program, down to Function level the code space usage can be viewed in SourceBoost IDE.
- **Target RAM Usage** - From the complete program, down to Function level the RAM usage can be examined and reviewed in SourceBoost IDE.

stallation

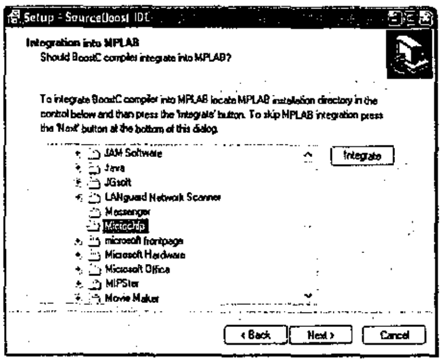
e BoostC compiler cannot be downloaded or installed on its own. BoostC is part the SourceBoost software package that includes the SourceBoost IDE and other nguage Suites. It is available for download from our site p://www.sourceboost.com

ıen you buy a license, you will activation code(s) and detailed instructions on w to activate the compiler and other tools you have licensed.

install SourceBoost IDE and BoostC on your system, please follow these simple ıps:

- Execute the installer sourceboost.exe and follow on-screen directions.

- Please pay attention to the integration dialog:

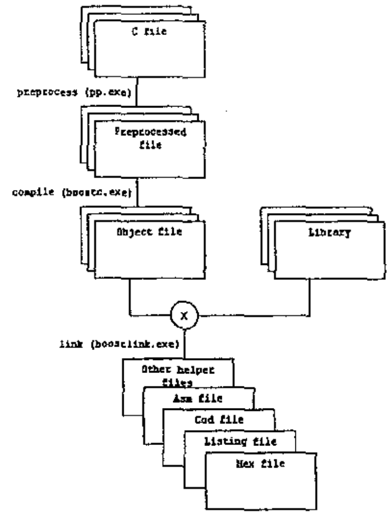


  To integrate BoostC with MPLAB, choose the correct Microchip installation directory, then click on "Integrate" before stepping to the next installation wizard dialog.

- The rest of the installation process is straightforward. At the end, SourceBoost IDE is ready to be used on your system. Should any difficulty arise, please double check your system configuration and mail all details to support@sourceboost.com

- **Please note:** if the installation step "MPLAB Integration" is skipped, the necessary MPLAB integration files will be installed to the \mplab subdirectory of the chosen SourceBoost installation directory. These files can always be manually copied to the correct location – please see the "MPLAB Integration" section later in this manual.

Compilation model and toolchain



Preprocessor

The preprocessor **pp.exe** is automatically invoked by the compiler.

Compiler

There are actually two separate compilers: one for pic16 and one for pic18 targets.

When you work under SourceBoost IDE, there is no need to specify which one to use: the IDE picks the correct compiler based on the selected target.

The output of the compiler is one or more **.obj** files, that are further processed by **librarian** or **linker**, in order to get a **.lib** or **.hex** file.

Linker

BoostLink Optimizing Linker links **.obj** files generated by compiler into a **.hex** file that is ready to send to target. It also generates some auxiliary files used for debugging and code analysis.

Librarian

Librarian is built into **BoostLink** linker executable and gets activated by `-lib` command line argument. There is a dedicated box in the Option dialog inside **SourceBoost IDE** that changes project target to library instead of hex file.

To create a target independent library, include *boostc.h* instead of *system.h* into the library sources. This way no target specific information (like target dependent constants or variables mapped to target specific registers) is included into the library. Note that this is the only case in which *system.h* does not be included into the code.

Differences with C2C compilation model

The main difference between **BoostC** and our previous generation **C2C** compiler is that the latter had a built-in linker and created an **.asm** file needing to be assembled using an external assembler (like MPASM), while the **BoostC** toolsuite doesn't need any external tools and directly generates the target **.hex** file.

Another difference is in how compilers handle read-only variables located in code memory. **BoostC** uses the special data type specifier '*rom*', while **C2C** placed any variable defined as '*const*' into code memory.

'LAB integration

›stC C compiler can be integrated into Microchips MPLAB integrated ·elopment environment (IDE). The MPLAB integration option should be selected ing the SourceBoost software package installation.

ase note: To use BoostC under MPLAB the MPLAB integration button must be ssed during the SourceBoost package installation. This copies some files and s the required registry keys required for integration to work.

case the installation step "MPLAB Integration" failed, the files in the ourceBoost>\mplab directory can be manually copied into

IPLAB IDE>\Third Party\MTC Suites for **MPLAB 7.x**, or

IPLAB IDE>\LegacyLanguageSuites for **MPLAB 6.x**.

the above examples, <MPLAB IDE> refers to the MPLAB installation directory d <SourceBoost> refers to the SourceBoost IDE and compilers installation ectory.

:atures

ıen **BoostC** is integrated into MPLAB IDE it allows the following:

- Use of the MPLAB Project Manager within MPLAB IDE.
- Creation and Editing of source code from within MPLAB IDE.
- Build a project without leaving MPLAB IDE.
- Source level debugging and variable monitoring using: MPLAB simulator, MPLAB ICD2, MPLAB ICE2000.

›tting the MPLAB Language Tool Locations

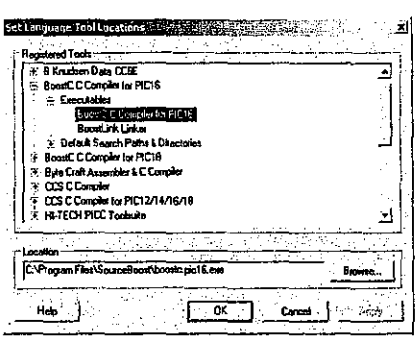
›te: this process only needs to be performed once.

ıe procedure below specifies paths assuming the default installation folder has ›en used for the SourceBoost software package.

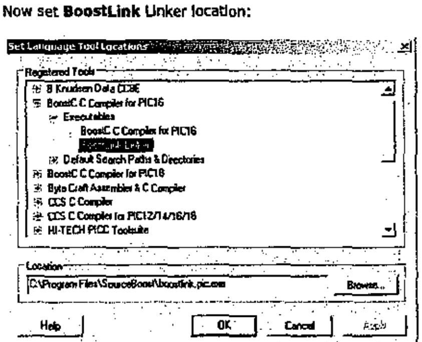
1. Start MPLAB IDE.

2. Menu **Project ➡ Set Language Tool Locations.**
   Note: If BoostC C compiler does not appear in the Registered Tools list, then the integration process during the SourceBoost installation was not performed or was unsuccessful.

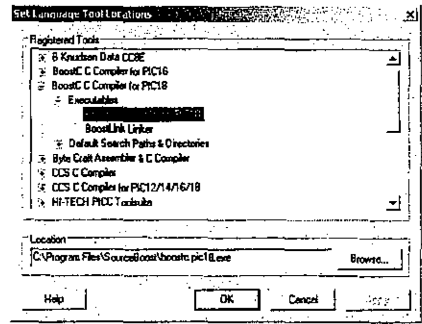
:. Set **BoostC C** compiler for PIC16 location:



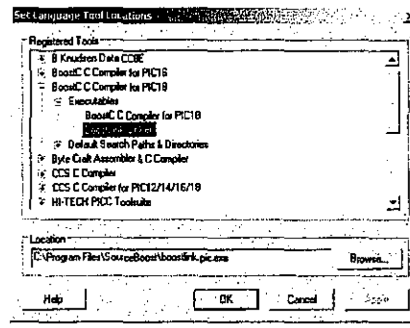
4. Now set **BoostLink** Linker location:



5. Set **BoostC C** compiler for PIC18 location:



6. Eventually, set **BoostLink** Linker location in the PIC18 tree:

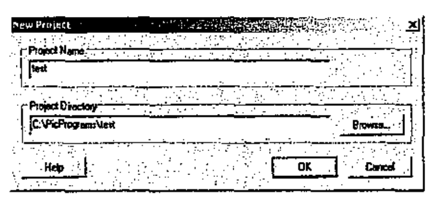


Creating a project under MPLAB IDE

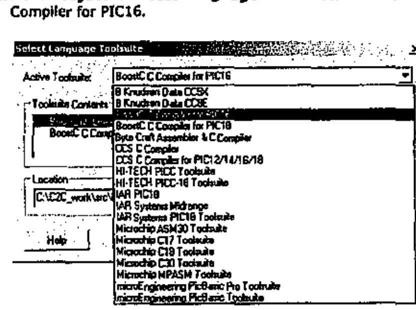
Before attempting to do this, please ensure that the **"Setting the MPLAB language tool locations"** process illustrated in the above section has been successfully performed.

The following steps will help you create a project under MPLAB IDE, that will be built using the BoostC C compiler, compiling for a PIC16 Target. The project name is test and the project and source code will be located in folder C:\PicPrograms\test

1. Menu **Project ➡ New.** Enter a project name and directory.
   Note: this can be an existing directory containing a SourceBoost IDE project.

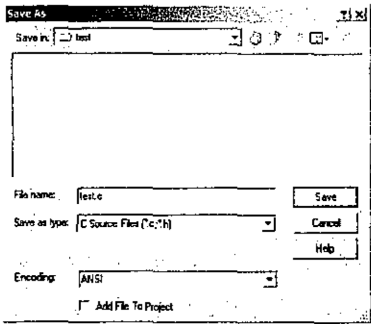


2. Menu **Project ➡ Select Language Toolsuite.** Select the **BoostC C** Compiler for PIC16.

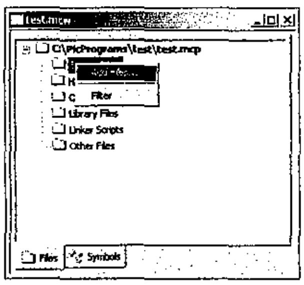


3. Menu **File ➡ New.** Type code into the Untitled window.
   Note: If you already have Source Files, this step and steps 4 can be skipped.

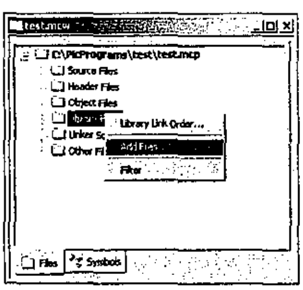
4. Menu **File ➡ Save As.** Locate the project folder using the Save As dialog box.



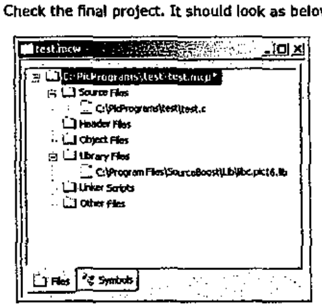
5. Add the test.c source file to the project by right clicking on Source Files in the project tree – as shown below.



. Add the libc.pic16.lib file (found in the C:\Program Files\SourceBoost\Lib folder) to the project by right clicking on Library Files in the project tree.



'. Check the final project. It should look as below:



8. Menu **Project ➡ Build** (or press the build button on the tool bar). The code should then be built.

ı can now use the MPLAB simulator, ICD2 or ICE to run the code, or a grammer to program a device. Please refer to the "Using ICD2" section of this :ument before using ICD2 to avoid potential problems.

ease of project browsing, you can also add the project header files to the ject tree in the same way as the source files where added.

Using ICD2

The are a few things to be aware of when using or planning using ICD2:

1. **RAM usage**: ICD2 uses some of the target devices RAM, leaving less room for the actual application.

   In order to reserve the RAM required by ICD2, and prevent Boost Linker from using it, the *icd2.h* header file must be included in the source code, eg:

```
#include <system.h>
#include <icd2.h>          // allocates RAM used by ICD2

void main()
{
    while(1);
}
```

2. **SFR usage**: ICD2 uses some Special Function Registers. This prevents the use of some peripheral devices when using ICD2 to debug code.

   Important: It is down the user to ensure that the ICD2 special function registers are not accessed. On some targets these registers reside at the same address as other peripheral device special function registers. Please check the documentation provided in the MPLAB IDE help for ICD2 resource usage in order to prevent problems.

3. **Break point overrun**: Due to timing skew in the target device (caused by instruction prefetch), execution will pass the instruction address where a breakpoint is set before it stops.

4. **NOP at ROM address 0**: See the BoostLink command line option -icd2 to add a NOP at ROM address 0.

Command line options

To get full list of BoostC compiler and BoostLink linker command line options run compiler or linker from command line.

BoostC command line

```
BoostC Optimizing C Compiler Version x.xx
http://www.sourceboost.com
Copyright(C) 2004-2007 Pavel Baranov
Copyright(C) 2004-2007 David Hobday

Licensed to <license info>

Usage: boostc.pic16.exe [options] files

Options:
  -t name     target processor (default name=PIC16F648A)
  -On         optimization level (default n=1)
              n=0 - optimization turned off
              n=1 - optimization turned on
              n=a - aggressive optimization turned on
              n=p - 32 bit long promotion turned on
  -Wn         warning level (default n=1)
              n=0 - no warnings
              n=1 - some warnings
              n=2 - all warnings
  -Werr       treat warnings as errors (default off)
  -i          debug inline code (default off)
  -Su         disable initialization of uninitialized static variables
  -d name     define 'name'
  -m          generate dependencies file (default off)
  -v          verbose mode turned on (default off)
  -I path1;path2  additional include directories
```

Optimization

Code optimization is controlled by -O command line option and *#pragma*.

Optimize flags:

-O0 no or very minimal optimization

-O1 regular optimization (this option is recommended for most applications)

-Oa aggressive optimization (produces shorter code and optimizes out some variables - this can make debugging more difficult!)

-Op promotes results of some 16 bit operations to 32 bits (can result in more efficient code is some cases).

ostLink command line

```
stLink Optimizing Linker Version x.xx
p://www.sourceboost.com
yright(C) 2004-2007 Pavel Baranov
yright(C) 2004-2007 David Hobday

ensed to <license info>

ge: boostlink.pic.exe [options] files
ions:
t name       target processor
On           optimization level 0-1 (default n=1)
             n=0 - no optimization
             n=1 - pattern matching and bank switching optimize on
v            verbose mode
d path       directory for project output
p name       project (output) name for multiple .obj file linking
ld path      directory for library search
rb address   ROM base (start) address to use
rt address   ROM top (end) address to use
wcs s1 s2 s3 Use software call stack. Hardware stack is allocated by
               specifying stack depths s1,s2,s3 (optional)
               s1 = main and task routines hardware stack allocation
               s2 = ISR hardware stack allocation
               s3 = PIC18 low priority ISR hardware stack allocation
srnoshadow   ISR No use of Shadow registers
srnocontext  ISR No context Save/restore is added to ISR(PIC18 only)
cd2          Add NOP at first ROM address for correct ICD2 operation
exela        Always add extended linear address record to .hex file

itches for making libraries:
-lib         make library file from supplied .obj and .lib files
-p name      project (library output file) name
```

)

is command line option causes the code generated by the linker to start at the dress specified. Boot loaders often reside in the low area of ROM.

ample

) 0x0800

s s1 s2 s3

command line option to the linker tells it to use a software call stack in ion to the hardware call stack. This allows subroutine calls deeper than the ardware call stack of the PIC. A function call that is made on the software call uses an extra byte of RAM to hold the return point number. This option must sed when using **Novo RTOS**. Where possible the hardware stack is used for ency. By specifying the depth of hardware stack to use for main (and Novo) ***s1***, ISR (interrupt service routine) ***s2*** and low priority ISR (PIC18 only) ***s3***, des control over when the software call stack is used instead of the hardware tack. The software call stack is applied to functions higher up in the call tree, lls lower down the call tree still use the hardware call stack. If no hardware depths are specified, then the software stack is only used in functions that ain or call functions that contain a **Novo RTOS** Sys_Yield() function.

nple:

s 6 2

routine will use hardware call stack up to a depth of 6 and then start using vare call stack. Interrupt routine will use hardware call stack up to a depth of 2 start using software call stack. An ISR uses hardware call stack depth of 1 to the address of the point where the code was interrupted, so in this example it leaves a hardware call stack depth 1 for subsequent calls within the ISR.

noshadow

command line switch tells the linker not to use the PIC18 shadow registers for rrupt service routine (ISR) context saving. This option is required as a work ind for silicon bugs in some PIC18's.

nocontext

option only works with PIC18's. When use this prevents the linker adding a code for context saving. This allow the programmer to generate their own imal ISR context saving code, or have none at all.

imple:

```
context saving example
ssumes that the ISR code will only modify w and and bsr

create context saving buffer at fixed address
r context[ 2 ]@0x0000;

d interrupt()

 asm movff _bsr, _context
 asm movwf _context+1
 ....
 asm movwf _context+1
 asm movff _bsr, _context
```

nclude

ntax: *#include* **<filename.h>**
or
#include **"filename.h"**

ments: **filename** is any valid PC filename. It may include standard drive and path information.
In the event no path is given, the following applies:

a) If filename appears between "", the directory of the projects is searched first.

b) If the delimiters <> are used, only the IDE *include path list* is searched for filename.

If the file is not found, an error will be issued and compilation shall stop.

irpose: Text from the include file **filename.h** is inserted at the point of the source where this directive appears, at compile time.

camples: #include <system.h>

-icd2

Use this command line switch to add a NOP instruction at the first ROM address used (usually address 0). This is required on some devices for correct operation of Microchip ICD2 (In Circuit Debugger).

-hexela

Always add extended linear address record to .hex file. Without this switch an extended linear address record is only added to the .hex file if required by addresses included in the .hex file.

libc Library

When a project is being linked, **SourceBoost IDE** adds *libc.pic16.lib* or *libc.pic18.lib* to the linker command line, if it can find this library in its default location.
The *libc* library contains necessary code for multiplication, division and dynamic memory allocation. It also includes code for string operations.

Code entry points

Entry points depend on the code address range using by the BoostLink linker. By default, the linker uses all available code space, but it's also possible to specify code start and end addresses that linker should use through linker command line options.

For PIC16:

Reset (main) entry point	<code start> + 0x00
Interrupt entry point	<code start> + 0x04

For PIC18:

Reset (main) entry point	<code start> + 0x00
Interrupt entry point	<code start> + 0x08
Low priority ISR entry point	<code start> + 0x18

SourceBoost IDE

The **SourceBoost IDE** is thoroughly covered in a separate manual.

fine

tax: *#define* **id statement**
or
#define **id(a, b...) statement**

nents: **id** is any valid preprocessor identifier.

statement is any valid text.

a, b and so on are local preprocessor identifiers, that in the given form model a function's formal parameters, separated by commas.

pose: Both forms produce a basic string replacement of **id** with the given **text.** Replacement will take place from the point where the *#define* statement appears in the program, and below.

The second form represents a preprocessor pseudo-function. The local identifiers are positionally matched up with the original text, and are replaced with the text passed to the macro wherever it is used.

mples:

```
#define LEN 16
#define LOWNIBBLE(x) ((x) & 0x0F)
...
a = 69;
le = a + LEN;       // becomes le = a + 16;
b = LOWNIBBLE(a);   // same as b = a & 0x0F;
```


Preprocessor

The **pp.exe** preprocessor is automatically invoked by the compiler. It executes a series of parametrized text substitutions and replacement (macro processing), besides evaluating special directives.

All preprocessor directives start with a '#'. Non standard directives are always contained in statements with a leading ANSI keyword *#pragma*, so to avoid potential conflicts when porting code to other compilers and/or with advanced source analysis tools (lint, static checkers, code formatters, flow analyzers and so on).

Directives

The following directives are supported by **pp**:

#include

#define

#undef

#if

#else

#endif

#ifdef

#ifndef

#error

#warning

These directives are individually explained in the following pages.

#undef

Syntax: *#undef* **id**

Elements: **id** is any valid preprocessor identifier previously defined via *#define*.

Purpose: Starting with the line where this directive appears, **id** will no more have meaning for the preprocessor, i.e. a subsequent *#ifdef* ***id*** shall evaluate to logical FALSE.

Please note that **id** can then be reused and assigned a different value.

Examples:

```
#define LEN 16
#define LOWNIBBLE(x) ((x) & 0x0F)
...
a = 69;
le = a + LEN;          // becomes le = a + 16;
#undef LEN             // LEN is not recognized anymore by pp
...
#define LEN 24         /* This is now valid and does not cause
"double definition attempt" errors. */
```


#if, #else, #endif

Syntax:

```
#if expr
    code
#else
    code
#endif
```

Elements: **expr** is any valid expression using constants, standard operators and preprocessor identifiers.
code is one or more valid C source code line.

Purpose: The preprocessor evaluates the constant expression **expr** and, if it is non-zero, will process the lines up to the optional *#else* or the *#endif*. Otherwise the optional *#else* branch code will be processed, if present.

The latter two preprocessor directives are also used with specialized forms of the *#if* directive (see ***#ifdef***, ***#ifndef***).

NOTE: **expr** cannot contain C variables ! Only constant expressions and operators can be used.

Examples:

```
// Conditionally initialize a RAM variable
#if STARTDELAY > 20
    slow = 1;
#else
    slow = 0;
#endif
```


def

ıtax:

```
#ifdef id
    code
#endif
```

ments: **id** is any valid preprocessor identifier.

code is one or more lines of valid C source code.

rpose: When the preprocessor encounters this directive, it evaluates whether the identifier **id** is in its symbol table (eg previously specified within a *#define* statement).
In case **id** is defined, the lines of **code** between *#ifdef* and *#endif* (or an optional *#else*, if present) will be processed.

In the opposite case, **code** statements between *#ifdef* and *#endif* will be ignored by the compiler.

NOTE: **Id** can not be a C variable ! Only preprocessor Identifiers created via *#define* can be used.

amples:

```
#define DEBUG
...
#ifdef DEBUG
    printf("Reached test point #1");
#endif
```


def

tax:

```
#ifndef id
    code
#endif
```

nents: **id** is any valid preprocessor identifier.

code is one or more lines of valid C source code.

pose: When the preprocessor encounters this directive, it evaluates whether the identifier **id** is in its symbol table (eg previously specified within a *#define* statement).

In case **id** is **not** defined, the lines of **code** between *#ifndef* and *#endif* (or an optional *#else*, if present) will be processed.

In the opposite case, **code** statements between *#ifndef* and *#endif* will be ignored by the compiler.

NOTE: **Id** can not be a C variable ! Only preprocessor identifiers created via *#define* can be used.

ımples:

```
#ifndef DEBUG
    printf("Debug disabled !");
#else
    printf("Reached test point #1");
#endif
```

#error

Syntax: *#error* **text**

Elements: **text** is any valid text.

Purpose: When the preprocessor encounters this directive, it stops compilation and issues and error. The user supplied **text** is printed as an informational message.

This directive is useful when coupled with the expression checking features of the preprocessor, to validate the coherence of configuration choices and defines made elsewhere in the sources and include files (or on the command line).

Examples:

```
#ifndef PWM_DEFAULT
  #error "MUST define a default value for speed !"
#endif
```


#warning

Syntax: *#warning* **text**

Elements: **text** is any valid text.

Purpose: When the preprocessor encounters this directive, it forces the compiler to issue a warning. The user supplied **text** is printed as an informational message.

This directive is useful when coupled with the expression checking features of the preprocessor, to validate the coherence of configuration choices and defines made elsewhere in the sources and include files (or on the command line).

Examples:

```
#ifndef NODEADDR
  #warning "ADDR not defined, will enter dynamic mode."
#endif
```

Pragma directives

Specific BoostC preprocessor directives all follow the ANSI keyword *#pragma*, so to avoid potential conflicts when porting code to other compilers and/or with advanced source analysis tools (lint, static checkers, code formatters, flow analyzers and so on).

The following directives are supported by **pp**:

#pragma DATA

#pragma CLOCK_FREQ

#pragma OPTIMIZE

These directives are individually explained in the following pages.

ragma DATA

ntax: *#pragma DATA* **addr, d1, d2, ...**

or

#pragma DATA **addr, "abcdefg1", "abcdefg2", ...**

:ments: **addr** is any valid code memory address.

d1, d2... are 8-bit integer constants.

"abcdefgX" is a character string, the ASCII values of the charcters will be stored as 8 bit value.

irpose: User data can be placed at a specific location using this construct. In particular, this can be used to specify target configuration word or to set some calibration/configuration data into on-chip eeprom.

:amples:

```
#pragma DATA 0x200, 0xA, 0xB, "test"
//Set PIC16 configuration word
#pragma DATA 0x2007, _HS_OSC & _WDT_OFF & _LVP_OFF
//Put some data into eeprom
#pragma DATA 0x2100, 0x12, 0x34, 0x56, 0x78, "ABCD"
```


agma CLOCK_FREQ

tax: *#pragma CLOCK_FREQ* **Frequency_in_Hz**

nents: **Frequency_in_Hz** is the processor's clock speed.

pose: The CLOCK_FREQ directive tells the compiler under what clock frequency the code is expected to run.

Note: delay code generated by the linker is based on this figure.

mples:

```
//Set 20 MHz clock frequency
#pragma CLOCK_FREQ 20000000
```

itialization of EEPROM Data

s often desirable to program the PIC on board EEPROM with initial data as part he programming process. This initial data can be included in the source code. 'ROM initialization data is set using the pragma directive: *#pragma DATA*.

ample:

```
· Initializes EEPROM with data: 0C 22 38 48 45 4C 4C 4F 00 FE 99
ragma DATA _EEPROM, 12, 34, 56, "HELLO", 0xFE, 0b10011001
```


#pragma OPTIMIZE

Syntax: *#pragma OPTIMIZE "Flags"*

Elements: **Flags** are the optimization flags also used on the command line.

Purpose: This directive sets new optimization, at function level.
It must be used in the global scope and applies to the function that follows this pragma.

The pragma argument should be enclosed into quotes and is same as argument of the -O compiler command line options.

Empty quotes reset the optimization level previously set by this pragma.

This is the current list of valid optimization flags:

- **0** no or very minimal optimization
- **1** regular optimization (recommended)
- **a** aggressive optimization
- **p** promotes results of some 16 bit operations to 32 bits

Examples:

```
//use aggressive optimization for function 'foo'
#pragma OPTIMIZE "a"

void foo()
{
  ...
}
```


nguage

section of the manual contains a condensed list of BoostC C compiler features. In no way intended to replace a complete C language manual or ANSI/ISO fication. It is targeted, instead, at the already expert C programmer that s a quick reference of BoostC and its peculiarities due to the specific PIC t platform.

ram structure

/ source file should include the general system header file, that in turn des target specific header (containing register mapped variables specific for :arget), some internal functions prototypes needed for code generation and manipulation function prototypes:

```
lude <system.h>
```

a types

data types

Size	Type
1 bit	bit, bool
8 bits	char, unsigned char, signed char
l6 bits	short, unsigned short, signed short
l6 bits	int, unsigned int, signed int
32 bits	long, unsigned long, signed long

difference between bit and bool data types is in the way how an expression ger than 1 bit) is assigned to a bit or bool operands.

it operands receive the least significant bit of the right side expression;

ool operands receive the value of the right side expression casted to bool.

example:

```
 a;
 b;
· x;

x & 2; // 'a' will be 'true' if the bit #1 in 'x' is set
       // and 'false' otherwise

x & 2; // 'b' will always be false, because bit #0
       // (the least significant bit) in the expression
       // result is zero - regardless of the value of 'x'
```

ctures and unions

ı struct and union keywords are supported.

Setting Device Configuration Options

In order for a program to be able to run on a target device the device configuration options need to be correctly set. For example having the wrong oscillator configuration setting may mean that the device has no clock, making it impossible for any code to be executed. Configuration data is set using the pragma directive: *#pragma DATA*.

Configuration options typically control:

- Oscillator configuration
- Brown out reset
- Power up reset timer
- Watchdog configuration
- Peripheral configurations
- Pin configurations
- Low voltage programming
- Memory protection
- Table read protection
- Stack overflow handling

The exact configuration options available depend on exactly which device is being used. The PIC18 devices have many more configuration options that the PIC16/PIC12 devices.

Configuration Example 1:

```
Rem Configuration for PIC16F874A
#pragma DATA _CONFIG, _CP_OFF & _PWRTE_OFF & _WDT_OFF & _HS_OSC & _LVP_OFF
```

Configuration Example 2:

```
Rem Configuration for PIC18F452
#pragma DATA   _CONFIG1H, _OSCS_OFF_1H & _HS_OSC_1H
#pragma DATA   _CONFIG2L, _BOR_ON_2L & _BORV_20_2L & _PWRT_OFF_2L
#pragma DATA   _CONFIG2H, _WDT_OFF_2H & _WDTPS_128_2H
#pragma DATA   _CONFIG3H, _CCP2MX_ON_3H
#pragma DATA   _CONFIG4L, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L
#pragma DATA   _CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
#pragma DATA   _CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H
#pragma DATA   _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
#pragma DATA   _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H
#pragma DATA   _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L
#pragma DATA   _CONFIG7H, _EBTRB_OFF_7H
```

You can find the defined configuration options for a given device by looking in the target devices BoostC header file (PIC18XXXX.h and PIC16XXXX.h) which can be found in the installation directory (typically "*C:\Program File\SourceBoost\include*"). Its also worth looking at relevant Microchip data sheet to find the exact function of each option.

Typedef

New names for data types can be defined using typedef operation:

```
typedef unsigned char uchar;
```

Enum

Enumerated data types are an handy type of automatically defined constant series. The declaration assigns a value of zero to the first symbolic constant of the list, and the assigns subsequent values (automatically incremented) to the following constants.

The user can, as well, arbitrarily assign numerical (signed) values at the beginning as well as in the middle of the series. Values following an explicit assignment use that value as a base and keep on incrementing from that point.

The data type for an enum type or typedef variable is, as per ANSI definition, *the smaller type that can contain the absolute maximum value of the constant series.*

```
enum ETypes { E_NONE=0, E_RED, E_GREEN, E_BLUE };

// Same as :
// #define E_NONE  0
// #define E_RED   1
// #define E_GREEN 2
// #define E_BLUE  3
```

Code size vs Data Types

Be sure to always use the smallest data types possible. The rule is simple: the bigger data types are used, the bigger code will be generated.
Thus, always follow these rules of thumb:

- Use char (8-bit or **byte**) as the default, everywhere;
- Use short or int (16-bit or **word**) for common arithmetic, counters and to hold ADC conversion results on advanced cores (with 10-bit or more internal ADC).
- Only as a as last resort, and only where absolutely necessary, use long (32-bit, **dword**) variables.

Another rule that also affects the size of produced code, though in a much smaller degree, is about sign.

Use **unsigned data types** wherever you can, and signed only when necessary. Unsigned math <u>always generates smaller (and typically faster) code</u> than signed.

Rom

Strings or arrays of data can be placed into program memory.

Such variables are declared using regular data types and rom storage specifier.

Such rom variables <u>must be initialized within declaration</u>:

```
rom char *text = "Test string";      // text with trailing zero
rom char *ptr  = "\0x64\11\12";      // 4 bytes: 0x64, 0x0B, 0x0C, 0x00
rom char *data = { 0x64, 11, 12 };   // 3 data bytes: 0x64, 0x0B, 0x0C
```

Please keep in mind that the rom storage specifier has several limitations:

- rom can be used with char data types only;
- there is no implicit cast between rom and regular data types. Though BoostC will not generate an explicit error for such a cast, it is expected that the operand should be casted back to its original type.
  If this is not done, the resulting code will behave unpredictably.
- a rom pointer is internally limited to 8-bits: the constant array size is thus limited to 256 elements. This is coherent with smaller cores constraints;
- access to rom elements has to be done exclusively through the [] operators and they **cannot be referenced with substring pointer initialized at runtime.** Please keep in mind that rom variables <u>must always and exclusively be initialized within declaration</u>;

Example of wrong referencing with a runtime initialized pointer:

```
/*
Part of the following code is WRONG and will FAIL because BoostC cannot
dynamically create the pointer to mystr[OFFSET] when the mystr array is
located in ROM.
*/

rom char *mystr = "Str_one \0 Str_two \0";

/* WRONG: a rom pointer must be initialized in declaration */
rom char *substr;

    substr = &mystr[OFFSET];          //** WRONG **
    cc = substr[0];                   //** WRONG **

    cc = mystr[OFFSET];               // Correct
```

Volatile

The *volatile* type specifier should be used with variables that:

a) Can be changed outside the normal program flow, and

b) Should **not** receive intermediate values within expressions.

For example, if a bit variable is mapped to a port pin, it is a good programming practice to declare such variable as *volatile*.

Code generated for expressions with *volatile* variables is a little longer when compared to 'regular' code:


```
latile bit pinB1@0x6.1; //declare bit variable mapped to pin 1, port B
```

urrently compiler generates different code only for expressions that assign values volatile bit variables. Also volatile variables are not checked for being initialized.

atic

oth global and local variables can be declared as static. This limits their scope to e current module.

nstants

nstants can be expressed in binary, octal, decimal and hexadecimal forms:

0bXXXX or XXXXb	binary number, where X is either 1 or 0
0XXXX	octal number, where X is a number between 0 and 7
XXXX	decimal number, where X is a number between 0 and 9
0xXXXX	hexadecimal number, where X is a number between 0 and 9 or A and F

rings

sides regular characters, strings can include escape sequences:

\nn	ASCII character, value nn is decimal
\xnn	ASCII character, value nn is hexadecimal
\a	ASCII character 0x07 (ALERT)
\b	ASCII character 0x08 (Backspace)
\t	ASCII character 0x09 (Horizontal Tab)
\r	ASCII character 0x0D (LF, Line Feed)
\v	ASCII character 0x0B (Vertical Tab)
\f	ASCII character 0x0C (Form Feed)
\n	ASCII character 0x0A (CR, Carriage Return)
\\	ASCII character 0x5C (the «\» character)
\'	ASCII character 0x27 (the «'» character)
\"	ASCII character 0x22 (the «"» character)
\?	ASCII character 0x3F (the «?» character)

bles

es can be declared and used in the standard ANSI C way.
ker will place variables at specific addresses. BoostLink analyzes the call and trees, so that it can re-use the same pool of RAM memory locations for es that don't collide with each other, being used disjointly by unrelated s active at different times.

a very effective way to minimize data memory usage.

er mapped variables

es can be forced to be placed at certain addresses. Syntax is the same as in acy C2C compiler:

```
ar@<addr>;
```

<addr> is an hex or decimal address.
chnique is used to access target specific registers from code.
note that system header files already contain all target specific registers, so no need to define them again in the user's code.

ables can also have fixed addresses. Their address includes bit position and made in 2 forms:

```
          //variable will be placed by linker at arbitrary position
0x40.1;   // dotted: access bit 1 of register 0x40
0x202;    // bit offset: access bit 2 of register 0x40 (0x40*8 + 2)
```

ess

'bit' variables, individual bits of every variable can be accessed using the '.' or:

```
ar;
= 1; //set bit 2 of variable 'var'
```

s

can have any number of dimensions. The only constraint is that an array t into a single RAM bank.

ers

s can be used in the standard general way, the only exception being s declared with the **rom** storage specifiers, that can only be accessed the [] operators.

Strings as function arguments

If a function has one or more char* arguments, it can be called with a constant string passed as an argument.
The compiler will reuse the same RAM memory allocated for such arguments when several similar calls are made within same code block.
For example, the code below will use the same memory block to temporarily store the strings "Date" and "Time":

```
...
foo( "Date" );
foo( "Time" );
...
```

Operators

If an operation result is not explicitly casted, it is promoted by default to 16 bit precision. For example, given the following expression:

```
long l = a * 100 / b; // 'a' and 'b' are 16 bit long variables
```

the result of the multiplication will be stored in a 16-bit long (**word**) temporary variable, that will be then divided by **b.** This 16-bit long result will eventually be stored in **l**. This is the ANSI 'C' standard behavior.

This behavior can be changed using the -Op compiler command line option or a local *#pragma OPTIMIZE* directive.
When this optimization is applied to the given expression, the multiplication result will be promoted to 32-bit long (**dword**) temporary variable, that will then be divided by **b:** the result, that is now 32 bit long, will eventually be stored in **l**.

Arithmetic

+ - * / % ++ --

Arithmetic Operator Examples

```
// +  is a binary operator. It is used to add or produce
//    the arithmetic sum of two operands.
// Example:
        c = a + b;
//  or
        c = 5 + 7;   // After the operation c = 12
```



```
// -  is a binary operator. It is used to subtract or produce
//    the difference of two operands. In other words, the second
//    operand is subtracted from the first operand.
// Example:
        c = a - b;
//  or
        c = 18 - 12;   // After the operation c = 6

// *  is a binary operator. It is used to multiply or produce
//    the arithmetic product of two operands.
// Example:
        c = a * b;
//  or
        c = 5 * 7;   // After the operation c = 35

// /  is a binary operator. It is used to divide or produce
//    the quotient of two operands. In other words, the first
//    operand is divided by the second operand.
// Example:
        c = a / b;
//  or
        c = 24 / 8;   // After the operation c = 3

// %  is a binary operator. It is used to produce the modulus
//    or remainder when two operands are divided.
// Examples:
      c = a % b;
// or
      c = 25 % 8;   // After the operation  c = 1
      c = 17 % 3;   // 17 % 3 = 5 with a remainder of 2
                    // After the operation  c = 2
      c = 17 % 4;   // 17 % 4 = 4 with a remainder of 1
                    // After the operation  c = 1

// ++  is a unary operator. It is used for pre-incrementing
//     or post-incrementing an operand.
// Examples:
      x = 10;
      c = x++;   // Post-increment.
                 // After the operation x = 11, c = 10
      x = 10;
      c = ++x;   // Pre-increment.
                 // After the operation x = 11, c = 11
```



```
/ --  is a unary operator. It is used for pre-decrementing
      or post-decrementing an operand.
/ Examples:
    x = 10;
    c = x--;   // Post-decrement.
               // After the operation x = 9, c = 10
    x = 10;
    c = --x;   // Pre-decrement.
               // After the operation x = 9, c = 9
```

ssignment

= += -= *= /= %= &= |= ^= <<= >>=

ssignment Operator Examples

```
/ =  is the ASSIGN operator. The value of the variable or
/    expression on the right side of the equal is assigned
/    to the variable on the left side of the equal.
/ Examples:
    x = 3;   // Whatever was in the variable x
             // has been replaced with 3.
    x = 2 + 4;   // Whatever was in the variable x
                 // has been replaced with 6.
    c = x + y;   // If x has a value of 12 and y has a value of 16.
                 // Whatever was in the variable c will
                 // be replaced with 28.

/ +=  is the combined ADD and ASSIGN operator. The variable
/     on the left side of the operator will be added to the
/     variable or expression on the right side. The result is
/     then assigned to the variable on the left side of the
/     operator.
/ Examples:
    x += 2;   // If x has an initial value of 14. After
              // the operation x will be 16.
    c += x + y;   // If c has an initial value of 10 and x has
                  // the value 12 and y has the value 16.
                  // After the operation c will be 38.

/ -=  is the combined SUBTRACT and ASSIGN operator. The variable
```



```
    or expression on the right side of the operator will be
    subtracted from the variable on the left side. The result
    is then assigned to the variable on the left side of the
    operator.
amples:
 x -= 2;   // If x has an initial value of 14. After
           // the operation x will be 12.
 c -= x + y;   // If c has an initial value of 38 and x has
               // the value 12 and y has the value 16.
               // After the operation c will be 10.

*=  is the combined MULTIPLY and ASSIGN operator. The variable
    on the left side of the operator will be multiplied by the
    variable or expression on the right side. The result is
    then assigned to the variable on the left side of the
    operator.
amples:
 x *= 2;   // If x has an initial value of 14. After
           // the operation x will be 28.
 c *= x + y;   // If c has an initial value of 10 and x has
               // the value 12 and y has the value 16.
               // After the operation c will be 280.

/=  is the combined DIVIDE and ASSIGN operator. The variable
    on the left side of the operator will be divided by the
    variable or expression on the right side. The result
    is then assigned to the variable on the left side of
    the operator.
kamples:
 x /= 2;   // If x has an initial value of 14. After
           // the operation x will be 7.
 c /= x + y;   // If c has an initial value of 280 and x has
               // the value 12 and y has the value 16.
               // After the operation c will be 10.

%=  is the combined MODULUS and ASSIGN operator. The variable
    on the left side of the operator will be divided by the
    variable or expression on the right side. The result,
    which is a remainder only, is then assigned to the
    variable on the left side of the operator.
xamples:
 x %= 2;   // If x has an initial value of 15. After
           // the operation x will be 1.
 y %= 7;   // If y has an initial value of 17. After
           // the operation y will be 3.
 c %= x + y;   // If c has an initial value of 19 and x has
```

```
                  // the value 4 and y has the value 3.
                  // After the operation c will be 5.

// &=  is the combined BITWISE-AND and ASSIGN operator. The variable
//     on the left side of the operator will be ANDed on a bit-by-bit
//     basis with the variable or constant on the right side. The
//     result is then assigned to the variable on the left side of
//     the operator.

// Examples:

    x &= y;   // If x has an initial value of 14 and y has the
              // value 5. After the operation x will be 4.

    c &= 0x07;   // If c has an initial value of 0x0E. After
                 // the operation c will be 0x06.

    y &= 0b11110001;   // If y has an initial value of 0b10001111.
                       // After the operation y will
                       // be 0b10000001.

// |=  is the combined BITWISE-OR and ASSIGN operator. The variable
//     on the left side of the operator will be ORed on a bit-by-bit
//     basis with the variable or constant on the right side. The
//     result is then assigned to the variable on the left side of
//     the operator.

// Examples:

    x |= y;   // If x has an initial value of 14 and y has the
              // value 5. After the operation x will be 15.

    c |= 0x07;   // If c has an initial value of 0x0E. After
                 // the operation c will be 0x0F.

    y |= 0b11110000;   // If y has an initial value of 0b10001110.
                       // After the operation y will
                       // be 0b11111110.

// ^=  is the combined BITWISE-XOR and ASSIGN operator. The variable
//     on the left side of the operator will be XORed on a bit-by-bit
//     basis with the variable or constant on the right side. The
//     result is then assigned to the variable on the left side of
//     the operator.

// Examples:

    x ^= y;   // If x has an initial value of 14 and y has the
              // value 5. After the operation x will be 11.

    c ^= 0x07;   // If c has a value of 0x0E. After the
                 // operation c will be 0x09.

    y ^= 0b11111000;   // If y has an initial value of 0b00011110.
                       // After the operation y will
                       // be 0b11100110.

// <<=  is the combined SHIFT-LEFT and ASSIGN operator. The variable
```



```
//      on the left side of the operator will be shifted left by the
//      number of places indicated by the variable or constant on
//      the right. The result is then assigned to the variable on
//      the left side of the operator.

// Examples:

    x <<= y;   // If x has an initial value of 14 and y has the
               // value 2. After the operation x will be 56.

    c <<= 0x01;   // If c has an initial value of 0x0E. After the
                  // operation c will be 0x1C.

    y <<= 0b00000010;   // If y has an initial value of 0b00011110.
                        // After the operation y will
                        // be 0b01111000.

// >>=  is the combined SHIFT-RIGHT and ASSIGN operator. The
//      variable on the left side of the operator will be shifted
//      right by the number of places indicated by the variable
//      or constant on the right. The result is then assigned to
//      the variable on the left side of the operator.

// Examples:

    x >>= y;   // If x has an initial value of 14 and y has the
               // value 2. After the operation x will be 3.

    c >>= 0x01;   // If c has an initial value of 0x0E. After the
                  // operation c will be 0x07.

    y >>= 0b00000010;   // If y has an initial value of 0b00011110.
                        // After the operation y will
                        // be 0b00000111.
```

Comparison

== != < <= > >=

Comparison Operator Examples

```
// ==  is a binary operator. It is used to see if one operand
//     IS equal to another operand.

// Example1:

    if( x == y )    // If c has an initial value of 0, and
    {               // x has the value 8 and y has the value 8.
       c = x + y;   // The final value for c will be 16.
    }

// Example2:

    if( x == y )    // If c has an initial value of 0, and
    {               // x has the value 8 and y has the value 5.
       c = x + y;   // The final value for c will be 0.
    }
```



```
 !=  is a binary operator. It is used to see if one operand
     is NOT equal to another operand.

Example1:

    if( x != y )    // If c has an initial value of 0, and
    {               // x has the value 8 and y has the value 5.
       c = x * y;   // The final value for c will be 40.
    }

Example2:

    if( x != y )    // If c has an initial value of 0, and
    {               // x has the value 8 and y has the value 8.
       c = x * y;   // The final value for c will be 0.
    }

 <   is a binary operator. It is used to see if one operand
     is LESS than another operand.

Example1:

    if( x < y )     // If c has an initial value of 0, and
    {               // x has the value 40 and y has the value 65.
       c = y - x;   // The final value for c will be 25.
    }

Example2:

    if( x < y )     // If c has an initial value of 0, and
    {               // x has the value 65 and y has the value 40.
       c = y - x;   // The final value for c will be 0.
    }

 <=  is a binary operator. It is used to see if one operand
     is LESS than or EQUAL to another operand.

Example1:

    if( x <= y )   // If x has a value of 22 and y has a value of 33.
       set_bit( PORTA, LED_bit );        // Turn LED ON
    else
       clear_bit( PORTA, LED_bit );  // Turn LED OFF

In this example the LED will be turned ON.

Example2:

    if( x <= y )   // If x has a value of 15 and y has a value of 8.
       set_bit( PORTA, LED_bit );        // Turn LED ON
    else
       clear_bit( PORTA, LED_bit );  // Turn LED OFF

In this example the LED will be turned OFF.

Example3:

    if( x <= y )   // If x has a value of 46 and y has a value of 46.
       set_bit( PORTA, LED_bit );        // Turn LED ON
```



```
else
   clear_bit( PORTA, LED_bit ); // Turn LED OFF

this example the LED will be turned ON.

  is a binary operator. It is used to see if one operand
  is GREATER than another operand.

ample1:

if( x > y )     // If c has an initial value of 0, and
{               // x has the value 28 and y has the value 14.
   c = x / y;   // The final value for c will be 2.
}

ample2:

if( x > y )     // If c has an initial value of 0, and
{               // x has the value 14 and y has the value 28.
   c = x / y;   // The final value for c will be 0.
}

>=  is a binary operator. It is used to see if one operand
    is GREATER than or EQUAL to another operand.

ample1:

if( x >= y )   // If x has a value of 25 and y has a value of 10.
   set_bit( PORTA, LED_bit );        // Turn LED ON
else
   clear_bit( PORTA, LED_bit ); // Turn LED OFF

n this example the LED will be turned ON.

xample2:

if( x >= y )   // If x has a value of 8 and y has a value of 15.
   set_bit( PORTA, LED_bit );        // Turn LED ON
else
   clear_bit( PORTA, LED_bit ); // Turn LED OFF

n this example the LED will be turned OFF.

xample3:

if( x >= y )   // If x has a value of 34 and y has a value of 34.
   set_bit( PORTA, LED_bit );        // Turn LED ON
else
   clear_bit( PORTA, LED_bit ); // Turn LED OFF

n this example the LED will be turned ON.
```

Logical

&& || !

Logical Operator Examples

```
// &&  is a binary operator. It is used to determine if both operands
//     are true. The operands are expressions that evaluate to true
//     or false.

// Example1:

    if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );   // Turn alarm OFF
    else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 70 the alarm will be turned OFF.

// Example2:

    if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );   // Turn alarm OFF
    else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 105 the alarm will be turned ON.

// Example3:

    if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );   // Turn alarm OFF
    else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 25 the alarm will be turned ON.


// ||  is a binary operator. It is used to determine if either
//     operand is true. The operands are expressions that evaluate
//     to true or false.

// Example1:

    if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );       // Turn LED ON
    else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 8 the LED will be turned ON.

// Example2:

    if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );       // Turn LED ON
    else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 6 the LED will be turned OFF.
```



```
// Example3:

    if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );       // Turn LED ON
    else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 4 the LED will be turned ON.


// !   is a unary operator. It is used to complement an evaluated
//     operand. The operand is an expression that evaluates to
//     true or false.

// Example1:

    if( !( pressure > 120 ) )
        clear_bit( PORTA, ALARM_bit );   // Turn alarm OFF
    else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If pressure has a value of 75 the alarm will be turned OFF.

// Example2:

    if( !( pressure > 120 ) )
        clear_bit( PORTA, ALARM_bit );   // Turn alarm OFF
    else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If pressure has a value of 125 the alarm will be turned ON.
```

Bitwise

& | ^ ~ << >>

Bitwise Operator Examples

```
// &   is a binary operator. It is used to produce the logical product
//     of two operands. The individual bits of two operands are ANDed
//     together to produce the final results.

// Examples:

    c = x & y;     // If x has a value of 14 and y has a value
                   // of 5. After the operation c will be 4.

    x = y & 0x07;  // If y has a value of 0x0E. After the
                   // operation x will be 0x06.

    x = y & 0b11110001;  // If y has a value of 0b10001111.
                         // After the operation x will
                         // be 0b10000001.
```



```
// |   is a binary operator. It is used to produce the logical sum of
//     two operands. The individual bits of two operands are ORed
//     together to produce the final results.

// Examples:

    c = x | y;     // If x has a value of 14 and y has a value
                   // of 5. After the operation c will be 15.

    x = y | 0x07;  // If y has a value of 0x0E. After the
                   // operation x will be 0x0F.

    x = y | 0b11110000;  // If y has a value of 0b10001110.
                         // After the operation x will
                         // be 0b11111110.


  ^   is a binary operator. It is used to produce the logical
      difference of two operands. The individual bits of two operands
      are XORed together to produce the final results.

Examples:

    x = y ^ 0x07;  // If y has a value of 0x0E. After the
                   // operation x will be 0x09.

    x = y ^ 0b11111000;  // If y has a value of 0b00011110.
                         // After the operation x will
                         // be 0b11100110.


 ~    is a unary operator. It is used to produce the complement of an
      operand. The individual bits of the operand are complemented.
      The ones become zeros and the zeros become ones.

Examples:

    x = ~y;   // If y has a value of 0x0E. After the
              // operation x will be 0xF1.

    x = ~0b01010111;  // After the operation x will
                      // be 0b10101000.


 <<   is a binary operator. The operand on the left side of the
      operator will be shifted left by the number of places
      indicated by the operand on the right.

Examples:

    c = x << y;    // If x has a value of 14 and y has a value
                   // of 2. After the operation c will be 56.

    x = y << 0x01; // If y has a value of 0x0E. After the
                   // operation x will be 0x1C.

    x = y << 0b00000010;  // If y has a value of 0b00011110.
                          // After the operation x will
                          // be 0b01111000.


 >>   is a binary operator. The operand on the left side of the
```



```
      operator will be shifted right by the number of places
      indicated by the operand on the right.

les:

  x >> y;   // If x has a value of 14 and y has a value of 2.
            // After the operation c will be 3.

  y >> 0x01;  // If y has a value of 0x0E. After the operation
              // x will be 0x07.

  y >> 0b00000010;  // If y has a value of 0b00011110.
                    // After the operation x will
                    // be 0b00000111.
```

nals

else statement
tch statement
ternary operator

nal Examples

```
 else  is a two-way decision making statement. If the
       expression evaluates to true, the first statement
       will be executed. If it evaluates to false the
       second statement will be executed.

e1:

 x > y )   // If x has a value of 25 and y has a value of 10.
set_bit( PORTA, LED_bit );      // Turn LED ON

clear_bit( PORTA, LED_bit ); // Turn LED OFF

s example the LED will be turned ON.

e2:

 x > y )   // If x has a value of 8 and y has a value of 15.
set_bit( PORTA, LED_bit );      // Turn LED ON

clear_bit( PORTA, LED_bit ); // Turn LED OFF

s example the LED will be turned OFF.


h  is a multi-way decision making statement. The variable is
   compared with the different cases. The case that matches
   will have its statements executed.

e1:
```

```
    switch( weight )
    {
        case 5:
            set_bit( PORTA, red_LED );         // Turn red LED ON
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            break;
        case 10:
            set_bit( PORTA, green_LED );       // Turn green LED ON
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
            break;
        default:
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
    }
// If the 'weight' variable has a value of 5 the red LED will
// be turned ON and the green LED will be turned OFF.

// Example2:

    switch( weight )
    {
        case 5:
            set_bit( PORTA, red_LED );         // Turn red LED ON
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            break;
        case 10:
            set_bit( PORTA, green_LED );       // Turn green LED ON
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
            break;
        default:
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
    }
// If the 'weight' variable has a value of 10 the green LED will
// be turned ON and the red LED will be turned OFF.

// Example3:

    switch( weight )
    {
        case 5:
            set_bit( PORTA, red_LED );         // Turn red LED ON
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            break;
        case 10:
            set_bit( PORTA, green_LED );       // Turn green LED ON
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
            break;
        default:
            clear_bit( PORTA, green_LED );     // Turn green LED OFF
            clear_bit( PORTA, red_LED );       // Turn red LED OFF
    }

// If the 'weight' variable has any value other than 5 or 10,
// both the green and red LEDs will be turned OFF.


//  ? :  is an if/else operator. This operator can be used inside an
//       expression to determine if a part of it is true or false.

// Example1:          turn the LED ON              turn the LED OFF
```



```
    (volts > 5) ? set_bit(PORTA, LED_bit) : clear_bit(PORTA, LED_bit);
// If 'volts' has a value of 7 turn the LED ON.

// Example2:          turn the LED ON              turn the LED OFF
    (volts > 5) ? set_bit(PORTA, LED_bit) : clear_bit(PORTA, LED_bit);
// If 'volts' has a value of 3 turn the LED OFF.
```

Program Flow

- while
- do / while
- for
- break
- continue

Program Flow Examples

```
//  while  is a loop control construct. It controls the execution of
//         a block of statements for as long as an expression evaluates
//         to true. The expression is evaluated first and if true,
//         executes the block. If it evaluates to false, stop the
//         execution.

// Example1:

    while( number > 0 )
    {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //  before entering the loop.
    }

// If 'number' has a value of 3 'factorial' will become 6.
// factorial = 3 x 2 x 1 ;

// Example2:

    while( number > 0 )
    {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //  before entering the loop.
    }

// If 'number' has a value of 0, 'factorial' will stay
//  equal to 1 because the loop was never entered.


//  do / while  is a loop control construct. It controls the execution
//              of a block of statements for as long as an expression
//              evaluates to true. The block is executed at least once
//              before the expression is evaluated. If it evaluates to
//              false, stop the execution. If it evaluates to true,
```



```
            continue the execution.

Example1:

  do
  {
    factorial *= number;    // 'factorial' is initialized to 1
    --number;               //  before entering the loop.

  } while( number > 0 );

If 'number' has a value of 4 'factorial' will become 24.
factorial = 4 x 3 x 2 x 1 ;

Example2:

  do
  {
    factorial *= number;    // 'factorial' is initialized to 1
    --number;               //  before entering the loop.

  } while( number > 0 );

If 'number' has a value of 0, 'factorial' will become 0 because
 the loop was entered, before the expression was evaluated.


 for  is a loop control construct. It controls the number of times
      a block of statements is executed. The construct has an
      initial value, a final value, and a loop-count value that is
      incremented each time after the block is executed.

Example1:

  for( volts = 0; volts < 7; volts++ )
  {
    sum += volts;       // 'sum' is initialized to 0
  }                     //  before entering the loop.

Upon exiting the loop 'sum' will have a value of 21.


 break  is an option that can be used to exit out of a for-loop,
        based upon the evaluation of an expression.

Example1:

  for( volts = 0; volts < 7; volts++ )
  {
    if( volts == 5 )
      break;

    sum += volts;       // 'sum' is initialized to 0
  }                     //  before entering the loop.

Upon exiting the loop 'sum' will have a value of only 10.


 continue  is an option used to redirect a for-loop based upon the
           evaluation of an expression. If the expression evaluates
           to true, the block of statements will not be executed.

Example1:
```



```
for( volts = 0; volts < 7; volts++ )
{
  if( volts == 5 )
    continue;

  sum += volts;       // 'sum' is initialized to 0
}                     //  before entering the loop.

on exiting the loop 'sum' will have a value of only 16.
um' will only have the values 0, 1, 2, 3, 4, & 6 added together.
```

e vast majority of programming books, the usage of 'goto' is heavily ecated. This is true for BoostC and PIC C coding as well: it should normally be ed.

: are, anyway, some very specific circumstances where it may still be useful: timize early exit cases within complex nested control structures or to simplify error handling (it can somehow mimic try/catch exception handling syntax).

```
:( ... )

ile(...)

hile(...)
[
 goto exit;
}
```

ne assembly

he asm or _asm operators to embed assembly into C code.

. switching and code page switching code should NOT be added to inline mbly code. The linker will add the appropriate Bank switching and code page hing code.

: will be affected as follows:

Bank switching added automatically.

Code page switching added automatically.

m

: will be affected as follows:

Bank switching added automatically.

Code page switching added automatically.

Other optimizations applied (including dead code removal).

Variable Referencing in asm

To refer to a C variable from inline assembly, simply prefix its name with an underscore '_'. If a C variable name is used with the 'movlw' instruction, the address of this variable is copied into W.
Labels are identified with a trailing semicolon ':' after the label name.

Inline assembly example 1

```
// Example showing use of bit tests and labels in inline assembly
#include <system.h>

void foo()
{
  unsigned char i, b;
  i = 0;
  b = 12;
  asm
  {
    start:
    btfsc _i, 4
    goto  end
    btfss _b, 0
    goto  iter

    iter:
    movlw 0
    movwf _b
    end:
  }
}
```

Inline assembly example 2

```
// Example for PIC18F8720 target showing how to access bytes of
// integer arguments
#include <system.h>

int GetTmr1Val()
{
  int x;
  asm
  {
    movf _tmr1h, W
    movwf _x+1   ; write to high byte of variable x
    movf _tmr1l, W
    movwf _x     ; write to low byte of variable x
  }

  return x;
}
```


Inline assembly example 3

```
// Example of how to access structure members from inline assembly.
//
// Note: This code may not work as expected if the data structure
// is modified causing member count2 to have a different offset.

struct Stats
{
    unsigned int count0; // stored in bytes 0 & 1
    unsigned char count1;// stored in byte 2
    unsigned int count2; // stored in bytes 3 & 4
};

struct Stats myStats;

void AddCount2()
{
  int x;
  asm
  {
    movf _myStats+3, W
    addlw 0x01
    movwf _myStats+3
    btfsc _status, C
    incf _myStats+4,F
  }
}
```

User Data

User data can be placed at the current location using the 'data' assembly instruction followed by comma separated numbers or strings.
Example:

```
// Code below will place bytes 10,11,116,101,115,116,0
// at current code location
asm data 0xA, 0xB, "test"
```

Functions

Inline functions

Functions declared as inline are repeatedly embedded into the code for each occurrence. When a function is defined as inline, its body must be defined before it gets called for the first time.

Though any function can be declared as inline, procedures (functions with no return value and a possibly empty argument list) are best suited to be used as inline. An exception to this rule are inline functions with reference arguments. Such functions will not overload variables passed as arguments but will operate directly on them:

```
inline void foo( char &port )
{
    port = 0xFF; // set all pins of a port
```


n heavy usage of inline functions obviously augments code size.**

›ecial functions

```
›id main(void)
```

›ogram entry point. This function is mandatory for every C program.

```
›id interrupt(void)
```

›terrupt handler function. Is linked to high priority interrupts for PIC18 parts.

```
›id interrupt_low(void)
```

›w priority interrupt handler, can be used only on the PIC18 family.

eneral functions and interrupts

:andard user functions are not *thread-safe*: their local variables are **not** saved hen function execution gets interrupted by an interrupt. This can lead to very ard to trace errors.

› prevent this pitfall, the linker does not allow to call a given function from both ain() and interrupt threads.

you really need to use same function in both threads, you need to duplicate its ›de and assign a different name to the second copy.


```
s function gets called from main thread
 foo()



y of 'foo' that will be called from interrupt thread
 foo_interrupt()



terrupt thread
 interrupt( void )

›_interrupt();



n thread
main( void )

›O;
```

›amic memory management

›mic memory management is used to dynamically create and destroy objects n time.
example, this functionality may be needed when a program needs to keep ral data packets. Memory for this packets can also be allocated at compile , but this way the memory may not be available for other variables even if it's ›sed.
solution is to use dynamic memory allocation. Objects to store data are :ed as soon as they are needed and destroyed after data gets processed.
way all available target data memory is used most efficiently.

amount of possible objects that can be allocated depends on the specific PIC at hand, and on the application.

n the application is built, the linker uses RAM memory left after allocation of al and local variables as a heap. When some memory gets allocated at run by the 'alloc' call, it gets allocated from this heap. The bigger the heap, the › run time objects can exist at any given time.

```
* alloc(unsigned char size)
```

›mically allocate memory 'size' bytes long. Max size is 127 bytes. Returns . if memory can't be allocated.

```
 free(void *ptr)
```

Free memory previously allocated by 'alloc'. Passing any other pointer will lead to unpredictable results.

C language superset

The BoostC compiler has some advanced features "borrowed" from C++ language. These features allow development of more flexible and powerful code, but their use is merely optional.

References as function arguments

Function arguments can be references to other variables.
When such argument changes inside a function the original variable used in function call changes too.
This is a very powerful way to alter the data flow without blowing up the generated code:

```
void foo(char &n)          //'n' is a reference
{
  n = 100;
}
void main(void)
{
  char a = 0;
  foo( a );                //upon return 'a' will have value of 100
  ...
}
```

Notes on using references as function arguments

For general efficiency, the mechanism used to pass a variable by reference is that of taking a copy of the variable data when the function is called, and by copying the data back to the original variable after the function has been exited.

Passing a large structure by reference will generate a large amount of code to copy the data back and fourth. Passing volatile variables (those declared using the volatile type specifier) may result in not the behavior you would expect, despite being a volatile variable its value will only get updated on exit of the function.

General guidelines:

- Don't pass large data structures by reference.
- Don't pass volatile data by reference.

Function overloading

There can be more than one function in the same application having a given name. Such functions must anyway differ by the number and type of their arguments:


```
d foo( void )            //'foo' number 1

d foo( char *ptr )       //'foo' number 2

d foo( char a, char b )  //'foo' number 3

d main( void )

oo();                //'foo' number 1 gets called
oo( "test" );        //'foo' number 2 gets called
oo( 10, 20 );        //'foo' number 3 gets called
..
```

: compiler will generate internal references to the functions so that no biguity is possible (*name mangling*), and will select which function will be oked for each call analyzing how many parameters are passed, as well as their e.

nction templates

ictions can be declared and defined using data type placeholders.
s feature allows writing very general code (for example, linked lists handling) t is not tied to a particular data type and, what may be more important, allows : user to create template libraries contained in header files:

```
plate <class T>
d foo( T *t )

d main( void )

hort s;
oo<char>( "test" );    //'foo( char* )' gets called
oo<short>( &s );       //'foo( short* )' gets called
..
```


netric timing functions

f software based timing functions are strictly dependent on clock speed. s parameter is usually well known at linking time, depending only on are design and implementation, such functions can be dynamically ited, once the clock frequency is correctly assigned with the CLOCK_FREQ a.

functions can be used in the standard way when writing any program for :.

'elay_us(unsigned char t)

ated function) Delays execution for 't' micro seconds. Declared in boostc.h nction gets generated every time a project is linked and is controlled by the _FREQ pragma. In some cases when clock frequency is too low it's not ally possible to generate this function. If that's the case linker will issue a g.

'elay_10us(unsigned char t)

ated function) Delays execution for 't*10' micro seconds. Declared in .h This function gets generated every time a project is linked and is lled by the CLOCK_FREQ pragma. In some cases, when clock frequency is ·, it's not physically possible to generate this function. 's the case, the linker will issue a warning.

lelay_100us(unsigned char t)

'ated function) Delays execution for 't*100' micro seconds. Declared in .h This function gets generated every time a project is linked and is lled by the CLOCK_FREQ pragma. In some cases, when clock frequency is ·, it's not physically possible to generate this function. 's the case, the linker will issue a warning.

lelay_ms(unsigned char t)

'ated function) Delays execution for 't' milli seconds. Declared in boostc.h inction gets generated every time a project is linked and is controlled by the (_FREQ pragma.

lelay_s(unsigned char t)

'ated function) Delays execution for 't' seconds. Declared in boostc.h This in gets generated every time a project is linked and is controlled by the (_FREQ pragma.

: **about delays**: The delays provided are ***at least*** the value specified, the will be longer rather than shorter. The delays produced may be larger than ed if the delay routine is interrupted by an interrupt.

the clock frequency is such that the delay becomes highly inaccurate then :lay overhead, unit delay and delay resolution of the delay are displayed the linking process.

Delay Overhead – The delay created in calling, setting up and returning from the delay function.

Unit Delay – The amount of additional delay generated for a delay value increase of 1.

Delay Resolution – The amount the delay value has to be increased before an actual increase in the delay occurs. A delay resolution of 4 would mean that the delay value may need to be increased by a value of up to 4 in order to see an increase in the delay.

System Libraries

A number of standard functions are included into BoostC installations. The number of such functions isn't static. It increases from release to release as new features are added. Most of these functions are declared in boostc.h (It's not recommended to include boostc.h directly into your code. Instead include system.h which in turn included boostc.h)

General purpose functions

clear_bit(var, num)

(macro) Clears bit 'num' in variable 'var'. Declared in boostc.h

set_bit(var, num)

(macro) Sets bit 'num' in variable 'var'. Declared in boostc.h

test_bit(var, num)

(macro) Tests if bit 'num' in variable 'var' is set. Declared in boostc.h

MAKESHORT(dst, lobyte, hibyte)

(macro) Makes a 16 bit long value (stored in 'dst') from two 8-bit long values (low byte 'lobyte' and high byte 'hibyte'). 'dst' must be a 16-bit long variable. Declared in boostc.h

```
 unsigned short res;
 MAKESHORT( res, adresl, adresh ); //make 16 bit value from adresh:adresl
registers and write it into variable 'res'
```

LOBYTE(dst, src)

(macro) Gets low byte from 'src' and writes it into 'dst'. Declared in boostc.h

HIBYTE(dst, src)

(macro) Gets high byte from 'src' and writes it into 'dst'. 'src' must be a 16-bit long variable. Declared in boostc.h

void nop(void)

(inline function) Generates one 'nop' instruction. Declared in boostc.h

void clear_wdt(void)

(inline function) Generates one 'clrwdt' instruction. Declared in boostc.h

void sleep(void)

(inline function) Generates one 'sleep' instruction. Declared in boostc.h

String and Character Functions

void strcpy(char *dst, const char *src)
void strcpy(char *dst, rom char *src)

void strncpy(char *dst, const char *src, unsigned char len)
void strncpy(char *dst, rom char *src, unsigned char len)

(function) Copies zero terminated string 'src' into destination buffer 'dst'. Destination buffer must be big enough for string to fit. Declared in string.h

unsigned char strlen(const char *src)
unsigned char strlen(rom char *src)

(function) Returns length of a string. Declared in string.h

signed char strcmp(const char *src1, const char *src2)
signed char strcmp(rom char *src1, const char *src2)
signed char strcmp(const char *src1, rom char *src2)
signed char strcmp(rom char *src1, rom char *src2)

signed char stricmp(const char *src1, const char *src2)
signed char stricmp(rom char *src1, const char *src2)
signed char stricmp(const char *src1, rom char *src2)
signed char stricmp(rom char *src1, rom char *src2)

(function) Compares two strings. Returns -1 if string #1 is less than string #2, 1 if string #1 is greater than string #2 or 0 is string #1 is same as string #2. Declared in string.h

signed char strncmp(char *src1, char *src2, unsigned char len)
signed char strncmp(rom char *src1, char *src2, unsigned char len)
signed char strncmp(char *src1, rom char *src2, unsigned char len)
signed char strncmp(rom char *src1, rom char *src2, unsigned char len)

signed char strnicmp(char *src1, char *src2, unsigned char len)
signed char strnicmp(rom char *src1, char *src2, unsigned char len)
signed char strnicmp(char *src1, rom char *src2, unsigned char len)
signed char strnicmp(rom char *src1, rom char *src2, unsigned char len)

(function) Compares first 'len' characters of two strings. Returns -1 if string #1 is less than string #2, 1 if string #1 is greater than string #2 or 0 is string #1 is same as string #2. Declared in string.h

oid strcat(char *dst, const char *src)
oid strcat(char *dst, rom char *src)

oid strncat(char *dst, const char *src, unsigned char len)
oid strncat(char *dst, rom char *src, unsigned char len)

unction) Appends zero terminated string 'src' to destination string 'dst'. estination buffer must be big enough for string to fit. Declared in string.h

har* strpbrk(const char *ptr1, const char *ptr2)
har* strpbrk(const char *src, rom char *src)

nsigned char strcspn(const char *src1, const char *src2)
nsigned char strcspn(rom char *src1, const char *src2)
nsigned char strcspn(const char *src1, rom char *src2)
nsigned char strcspn(rom char *src1, rom char *src2)

unction) Locates the first occurrence of a character in the string that doesn't atch any character in the search string. Declared in string.h

nsigned char strspn(const char *src1, const char *src2)
nsigned char strspn(rom char *src1, const char *src2)
nsigned char strspn(const char *src1, rom char *src2)
nsigned char strspn(rom char *src1, rom char *src2)

unction) Locates the first occurrence of a character in the string. Declared in ring.h

har* strtok(const char *ptr1, const char *ptr2)
har* strtok(const char *src, rom char *src)

unction) Breaks string pointed into a sequence of tokens, each of which is elimited by a character from delimiter string. Declared in string.h

har* strchr(const char *src, char ch)

unction) Locates the first occurrence of a character in the string. Declared in ring.h

har* strrchr(const char *src, char ch)

unction) Locates the last occurrence of a character in the string. Declared in ring.h

har* strstr(const char *ptr1, const char *ptr2)
har* strstr(const char *src, rom char *src)

unction) Locates the first occurrence of a sub-string in the string. Declared in ring.h

ersion Functions

When using conversion functions that store the ASCII result in a buffer, be o provide a buffer of sufficient size or other memory may get overwritten. uffer needs to be enough to store the resulting characters and a null ator.

ned char sprintf(char* buffer, const char *format, unsigned int val)

its a numerical value to a string in the specified format. The buffer must be nough to hold the result. Only one numerical value can be output at a time. red in stdio.h.

it specified in the format string with the following format:

gs][width][radix specifier]

(	Example output	Description
	"-120"	decimal signed integer
	"150"	decimal unsigned integer
	"773"	octal unsigned integer
	"ABF1"	hex unsigned integer
	"101101"	binary unsigned integer

fication	Example output	Description
l"	"231 "	left justified, padded to 8 characters length
.6u"	"0000000000045102"	left justified, padded with zeroes to 16 characters length
b"	" 10"	right justified, padded 8 characters length

lay of	Example output	Description
3d"	"+972 "	left justified, padded 8 characters length, signed always displayed
d"	" 765 "	left justified, padded 8 characters length, positive signed displayed as ' '
ay of sign only applies to signed decimal radix. Radix and field width added o show complete format specification		

Implementation of field width is non standard - If a justification width is specified the width will be padded **or truncated** to match the width provided. The most significant digits and sign maybe truncated. Standard implementations do not truncate the output, which can cause unexpected buffer overrun.

unsigned char sprintf32(char* buffer, const char *format, unsigned long val)

Outputs a numerical value to a string in the specified format. The buffer must be long enough to hold the result. Only one numerical value can be output at a time. Declared in stdio.h.

This function operates as sprintf, but it handles a 32bit value. It also supports the "%l" radix specifier, which is handled the same as "%d".

int strtoi(const char* buffer, char** endPtr, unsigned char radix)

(Function) String to integer. A function that converts the numerical character string supplied into a signed integer (16 bit) value using the radix specified. Radix valid range 2 to 26.

buffer: Pointer to a numerical string.

endPtr: Address of a pointer. This is filled by the function with the address where string scan has ended. Allows determination of where there is the first non-numerical character in the string. Passing a NULL is valid and causes the end scan address not to be saved.

radix: The radix (number base) to use for the conversion, typical values: 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

Return: The converted value.

long strtol(const char* buffer, char** endPtr, unsigned char radix);

(Function) String to long integer. A function that converts the numerical character string supplied into a signed long integer (32 bit) value using the radix specified. Radix valid range 2 to 26.

buffer: Pointer to a numerical string

endPtr: Address of a pointer. This is filled by the function with the address where string scan has ended. Allows determination of where there is the first non-numerical character in the string. Passing a NULL is valid and causes the end scan address not to be saved.

radix: The radix (number base) to use for the conversion, typical values: 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

Return: The converted value.

int atoi(const char* buffer)

(Macro) ASCII to integer. A macro that converts the numerical character string supplied into a signed integer (16 bit) value using a radix of 10.

buffer: Pointer to a numerical string.

Return: The converted value.

Note: Macro implemented as: #define atoi(buffer) strtoi(buffer, NULL, 10)

long atol(const char* buffer)

(Macro) ASCII to long integer. A macro that converts the numerical character string supplied into a signed long integer (32 bit) value using a radix of 10.

buffer: Pointer to a numerical string.

Return: The converted value.

Note: Macro implemented as: #define atoi(buffer) strtoi(buffer, NULL, 10)

char* itoa(int val, char* buffer, unsigned char radix)

(Function) Integer to ASCII. function that converts an integer (16 bit) value into a character string.

char* ltoa(long val, char* buffer, unsigned char radix)

(Function) Long integer to ASCII. function that converts an long integer (32 bit) value into a character string.

Lightweight Conversion Functions

The standard conversion functions offer a lot of flexibility at the cost of ROM, RAM and execution time. For application that are short of RAM and ROM, or require shorter execution time, it maybe desirable to use the following lightweight functions.

void uitoa_hex(char* buffer, unsigned int val, unsigned char digits)

(Function) Unsigned integer to ASCII, hexadecimal representation. This function converts a 16 bit unsigned integer into a hex value with leading zeros. The number of digits is specified using by the *digits* parameter.

void uitoa_bin(char* buffer, unsigned int val, unsigned char digits)

(Function) Unsigned integer to ASCII, binary representation. This function converts a 16 bit unsigned integer into a binary value with leading zeros. The number of digits is specified using by the *digits* parameter.

void uitoa_dec(char* buffer, unsigned int val, unsigned char digits)

(Function) Unsigned integer to ASCII, decimal representation. This function converts and 16 bit unsigned integer into a decimal value with leading zeros. The number of digits is specified using by the *digits* parameter.

unsigned int atoui_hex(const char* buffer)

(Function) ASCII to unsigned integer, hexadecimal representation. This function converts a hexadecimal string value into 16 bit unsigned integer.

ısigned int atoui_bin(const char* buffer)

unction) ASCII to unsigned integer, binary representation. This function converts binary string value into 16 bit unsigned integer.

ısigned int atoui_dec(const char* buffer)

unction) ASCII to unsigned integer, decimal representation. This function ınverts a decimal string value into 16 bit unsigned integer.

haracter

ıar toupper(char ch)

unction) Converts lowercase character to uppercase. Declared in ctype.h

ıar tolower(char ch)

unction) Converts uppercase character to lowercase. Declared in ctype.h

ıar isdigit(char ch)

unction) Checks if character 'ch' is a digit. Returns non zero if this is a digit. eclared in ctype.h

ıar isalpha(char ch)

unction) Checks if character 'ch' is a letter. Returns non zero if this is a letter. eclared in ctype.h

ıar isalnum(char ch)

unction) Checks if character 'ch' is a letter or a digit. Returns non zero if this is a tter or a digit. Declared in ctype.h

ıar isblank(char ch)

unction) Returns a 1 if the argument is a standard blank character. All other puts will return a 0. The following are the standard blank characters: (space) or '\t' (horizontal tab). Declared in ctype.h

ıar iscntrl(char ch)

unction) Returns a 1 if the argument is a valid control character. All other inputs ill return a 0. Declared in ctype.h

ıar isgraph(char ch)

unction) Returns a 1 if the argument is a valid displayable ASCII character. All her inputs will return a 0. Declared in ctype.h

ıar islower(char ch)

unction) Returns a 1 if the argument is a valid lower-case ASCII letter. All other puts will return a 0. Declared in ctype.h

isprint(char ch)

tion) Returns a 1 if the argument is a valid printable ASCII character. All inputs will return a 0. Declared in ctype.h

ispunct(char ch)

tion) Returns a 1 if the argument is a valid punctuation character. All other s will return a 0. The following are the implemented punctuation characters: $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

isspace(char ch)

tion) Returns a 1 if the argument is a standard white-space character. All inputs will return a 0. Declared in ctype.h. The following are the standard -space characters:

acter Description	Character ASCII code	Character Escape sequence
ı	0x20	' '
ontal tab	0x09	'\t'
cal tab	0x0B	'\v'
ine	0x0A	'\n'
age return	0x0D	'\r'
feed	0x0C	'\f'

isupper(char ch)

tion) Returns a 1 if the argument is a valid upper-case ASCII letter. All other s will return a 0. Declared in ctype.h

isxdigit(char ch)

tion) Returns a 1 if the argument is a valid hexadecimal character. All other s will return a 0. Declared in ctype.h

**** memchr(const void *ptr, char ch, unsigned char len)***

tion) Locates the first character in memory. Declared in memory.h

ed char memcmp(const void *ptr1, const void *ptr2, unsigned char

tion) Compares memory. Declared in memory.h

**** memcpy(void *dst, const void *src, unsigned char len)***

tion) Copies memory. Declared in memory.h

**** memmove(void *dst, const void *src, unsigned char len)***

tion) Moves memory. Declared in memory.h

void* memset(void *ptr, char ch, unsigned char len)

(function) sets memory. Declared in memory.h

Miscellaneous Functions

unsigned short rand(void)

(function) Generates pseudo random number. Declared in rand.h Defined in rand.lib

void srand(unsigned short seed)

(function) Sets seed for pseudo random number generator. Declared in rand.h Defined in rand.lib

max(a, b)

(Macro) Returns the value of the argument with the largest value.

min(a, b)

(Macro) Returns the value of the argument with the smallest value.

I2C functions

i2c_init, i2c_start, i2c_restart, i2c_stop, i2c_read, i2c_write
(for more information look into i2c_driver.h and i2c_test.c files)

RS232 functions

uart_init, kbhit, getc, getch, putc, putch
(for more information look into serial_driver.h and serial_test.c files)

LCD functions

lcd_setup, lprintf, lcd_clear, lcd_write, lcd_funcmode, lcd_datamode
(for more information look into lcd_driver.h and lcd.c files)

Flash functions

short flash_read(short addr)

(function) Reads flash content from address 'addr'. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

void flash_loadbuffer(short data)

(function) Stores 'data' in an internal buffer of 4 shorts long. Must be called four times to fill the internal buffer. Data in this buffer is used by flash_write to store data in flash. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

void flash_write(short addr)

(function) Writes data from an internal buffer into flash at address 'addr'. The internal buffer that is 4 shorts long must be filled using 4 calls to flash_loadbuffer. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

EEPROM functions

char eeprom_read(char addr)

(function) Reads eeprom content from address 'addr'. Works with PIC16F87X devices. Declared in eeprom.h Defined in eeprom.pic16.lib

void eeprom_write(char addr, char data)

(function) Writes 'data' into eeprom at address 'addr'. Works with PIC16F87X devices. Declared in eeprom.h Defined in eeprom.pic16.lib

ADC functions

short adc_measure(char ch)

(function) Reads ADC channel 'ch'. ADC must be initialized before using this function. Works with PIC16F devices that have ADC units. Declared in adc.h Defined in adc.pic16.lib

A sample ADC initialization can look like:

```
volatile bit adc_on @ ADCON0 . ADON; //AC activate flag

set_bit(adcon1, ADFM); // AD result needs to be right justified
set_bit(adcon1, PCFG0); // all analog inputs
set_bit(adcon1, PCFG1); // Vref+ = Vdd
set_bit(adcon1, PCFG2); // Vref- = Vss

set_bit(adcon0, ADCS1); // Select Tad = 32 * Tosc (this depends on the X-
tal here 10 MHz, should work up to 20 MHz)
clear_bit(adcon0, CHS0); // Channel 0
clear_bit(adcon0, CHS1); //
clear_bit(adcon0, CHS2); //
adc_on = 1; // Activate AD module
```

One wire bus functions

char oo_busreset()

(function) Resets the one wire bus. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

Here is a typical scenario how to use the one wire library:


```
/ To be able to use the one wire library two global bit variables need to
e declared in the code.
/ These are the variables that control port pin ised for one wire
ommunication. For example
/ if the one wire interface is connected to pin 6 of port 8 the
eclaration will look like this:

define OO_PORT PORTB
define OO_TRIS TRISB
define OO_PIN 6

olatile bit oo_bus @ OO_PORT . OO_PIN;
olatile bit oo_bus_tris @ OO_TRIS . OO_PIN;

..

/ Reset the one wire bus
o_busreset();

/ Start the conversion (non-blocking function)
o_start_conversion();

/ Wait for completion, you could do other stuff here
/ But make sure that this function returns zero before
/ reading the scratchpad
f( oo_wait_for_completion() )

 //handle conversion time out

/ Read the scratchpad
f( oo_read_scratchpad() )

 //handle conversion error

/ And extract the temperature information
hort data = oo_get_data();
```

hort oo_get_data()

unction) Reads data from one wire bus. Declared in oo.h Defined in oo.pic16.lib nd oo.pic18.lib

har oo_read_scratchpad()

unction) Reads scratchpad. Declared in oo.h Defined in oo.pic16.lib and o.pic18.lib

oid oo_start_conversion()

unction) Starts conversion. Declared in oo.h Defined in oo.pic16.lib and o.pic18.lib

har oo_conversion_busy()

unction) Checks if conversion is in progress. Returns 0 if no conversion is active. eclared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

oo_wait_for_completion()

tion) Waits for a conversion to complete. Returns 0 if conversion completed 1 sec. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

PC System Requirements

In order to install and run the Compiler/SourceBoost Integrated Development Environment, a PC with the following specification is required:

Minimum System Specification
Microsoft Windows 95/98/ME/NT/2000/XP,
Adobe Reader and a web browser (to allow access to help files and manuals).
Pentium Processor or equivalent,
128MB of RAM,
CD ROM Drive,
80MB of disk space,
16Bit Color display Adapter at 800x600 Resolution.

Recommended System Specification
As the Minimum System Specification, plus:
2.0GHz (or faster) Processor,
512MByte (or more) RAM,
16Bit Color display Adapter at 1024x768 Resolution (or higher).

Technical support

For example projects and updates please refer to our website:
http://www.sourceboost.com

We operate a forum where technical and license issue problems can be posted. This should be the first place to visit:
http://forum.sourceboost.com

BoostC Support Subscription

By buying a support subscription you will receive priority technical support via email. This ensures that your query or problem will be at the front of the queue and receive the highest priority attention.

BoostC Support Subscriptions are here:
http://www.sourceboost.com/Products/BoostC/BuyLicense/SupportSubscription.html

Licensing Issues

If you have licensing issues, then please send a mail to:
support@sourceboost.com

General Support

For general support issues, please use our support forum:
http://forum.sourceboost.com

We are always pleased to hear your comments, this helps us to satisfy your needs. Post your comments on the SourceBoost Forum or send an email to:
support@sourceboost.com

al Information