# Chapter 3

## Basic Programming Robot

On successful completion of this course, students will be able to:

- Explain about robot's actuators.
- Program the sensors and motors for robot.

## Introduction

Robot becomes a new trend of students and engineers, especially with a main event and a robotics Olympiad each year. Programming the robot using microcontroller is the basic principle of controlling the robot, where the orientation of the microcontroller is to control the application of an information system based on the inputs received, and processed by a microcontroller, and the action performed on the output corresponding predetermined program.

## Robot's Actuators

Actuators are an important part of the robot that functions as an activator of the command given by the controller. Usually, an electromechanical actuator device produces movement. Actuator consists of two types:

- Electric Actuators.
- Pneumatic and Hydraulic Actuators.

In this sub-section will discuss the electric actuator which is often used as a producer of such rotational motion of the motor.

## DC Motor

A DC Motor in simple words is a device that converts direct current (electrical energy) into mechanical energy. It's of vital importance for the industry today, and is equally important for engineers to look into the working principle of DC motor in details. The very basic construction of a DC motor contains a current carrying armature which is connected to the supply end through commutator segments and brushes and placed within the north south poles of a permanent or an electro-magnet as shown in the figure below:

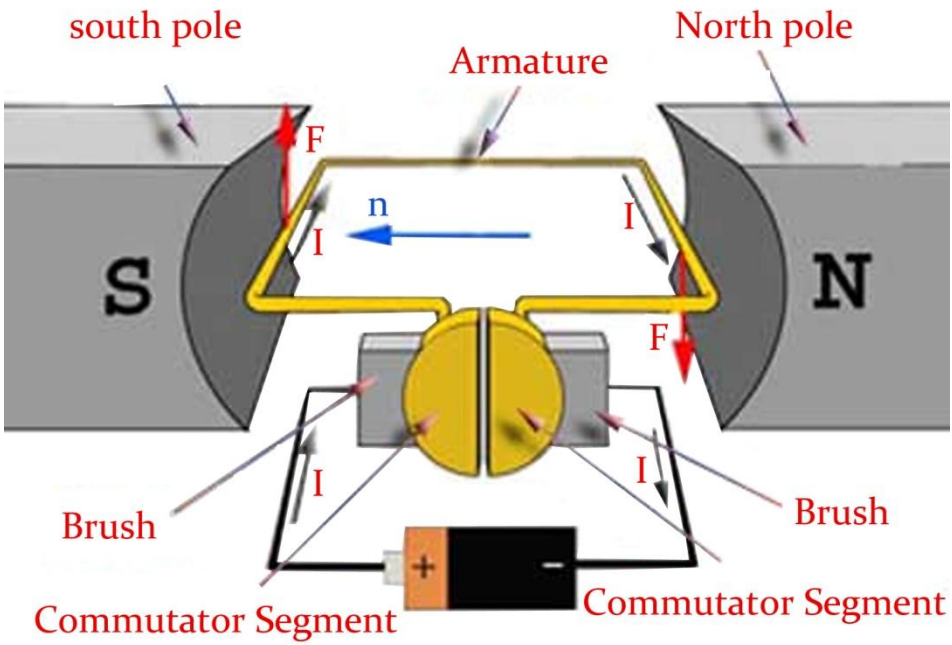


***Figure 3.1*** *DC Motor diagram.*

To understand the operating Principle of DC motor, it is important that we have a clear understanding of Fleming's left hand rule to determine the direction of force acting on the armature conductors of dc motor. Fleming's left hand rule says that if we extend the index finger, middle finger and thumb of our left hand in such a way that the current carrying conductor is placed in a magnetic field (represented by the index finger) is perpendicular to the direction of current (represented by the middle finger), then the conductor experiences a force in the direction (represented by the thumb) mutually perpendicular to both the direction of field and the current in the conductor.

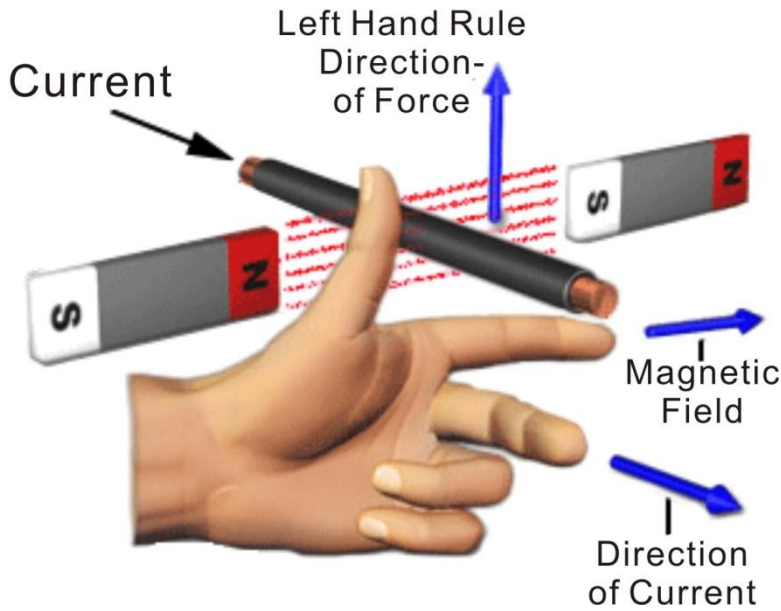


***Figure 3.2*** *Fleming's left hand rule.*

Figure below displays a DC motor with gearbox used on the robot to improve torque:

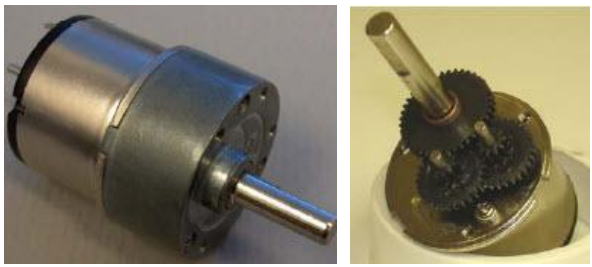

***Figure 3.3*** *An example of DC Motor with gearbox 7.2V 310RPM.*

## Servo Motor

Another important actuators are servo motors, which can work the wheel or as a robot arm or gripper. Servo motors are often used is continuous Servo Parallax, Parallax standard servo, GWS-S03, Hitec HS-805BB and HS-725BB. Some of the grippers are often used in the lab. Robot gripper usually based on

aluminum, lynxmotion robotic gripper hand and fingers are very popular as follows:

(a) (b)

***Figure 3.4*** *Lynxmotion robot hand RH1 with 2 servos (a) and gripper finger using 5 servos to 14 joint (b).*

Author recommends that you conduct experiments and make system-based visual servoing robotic arm that can pick up an object using a robotic arm based stereo camera. The robot arm is best used Dagu 6 degress of freedom and AX18FCM5 Smart Robotic arm that uses the CM-5 controller, Full feedback for position, speed, load, voltage and temperature, full control over position (300 degrees), uses servo AX-18F and is compatible with MATLAB and other common microcontroller systems.

(a) (b)

***Figure 3.5*** *Dagu 6 degree freedom arm robotic system using aluminum Dagu gripper (a) and AX18FCM5 Smart Robotic arm using CM-5 controller (b)[1].*

## Programming Motors of Robot

DC motors are usually driven by an H-Bridge since such a circuit can reverse the polarity of the motor connected to it. The DC brushed motors included in this kit are driven by the L6205 H-Bridge on the Propeller Robot Control Board. Understanding how to control this H-Bridge is the key to controlling the direction, speed and duration that the motors are on or off. Parallax has released a Propeller object called, "PWM_32" which makes it easy to drive servos as well as control motors using pulse width modulation. This object can be used with the Propeller Robot Control Board to drive the on-board H-Bridge, which in turn drives the DC motors.

The L6205 inputs are connected to P24 through P27 on the Propeller chip. When the power switch on the control board is set for POWER ON/MOTORS ON, the L6205 is enabled and the outputs are connected to the motors. The truth table for controlling the L6205 is shown below in Table 3.1. P24 and P25 control the left motor while P26 and P27 control the right motor. This table assumes the motors are connected to the control board as defined in the assembly instructions.

***Table 3.1*** *Motor truth table.*

| P24 | P25 | P26 | P27 | Left Motor | Right Motor |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Brake | Brake |
| 1 | 0 | 0 | 0 | Reverse | Brake |
| 0 | 1 | 0 | 0 | Forward | Brake |
| 1 | 1 | 0 | 0 | Brake | Brake |
| 0 | 0 | 1 | 0 | Brake | Forward |
| 1 | 0 | 1 | 0 | Reverse | Forward |
| 0 | 1 | 1 | 0 | Forward | Forward |
| 1 | 1 | 1 | 0 | Brake | Forward |
| 0 | 0 | 0 | 1 | Brake | Reverse |
| 1 | 0 | 0 | 1 | Reverse | Reverse |
| 0 | 1 | 0 | 1 | Forward | Reverse |
| 1 | 1 | 0 | 1 | Brake | Reverse |
| 0 | 0 | 1 | 1 | Brake | Brake |
| 1 | 0 | 1 | 1 | Reverse | Brake |
| 0 | 1 | 1 | 1 | Forward | Brake |
| 1 | 1 | 1 | 1 | Brake | Brake |

Note that it may be more intuitive to look at the table as two groups consisting of P24/P25 and P26/P27. In this manner you have 4 possible combinations for each motor as shown in Table 3.2.

***Table 3.2*** *The value given to P24 and P25 and P26 and 27 for the motors.*

| P24 | P25 | Left Motor | P26 | P27 | Right Motor |
|---|---|---|---|---|---|
| 0 | 0 | Brake | 0 | 0 | Brake |
| 1 | 0 | Reverse | 1 | 0 | Forward |
| 0 | 1 | Forward | 0 | 1 | Reverse |
| 1 | 1 | Brake | 1 | 1 | Brake |

The program to make the left motor active is shown below:

**File: LeftMotorTest.spin**

```
CON
_xinfreq = 5_000_000
_clkmode =xtal1 + pll16x
PUB Main
Dira[27..24] := %1111           ' Set P24 – P27 to output

    Outa [25] : = 1                 ' Left motor forward
    Waitcnt (clkfreq * 2 + cnt)     ' 2 seconds pause
    Outa [25] :=0                   ' Left motor stop
    Waitcnt (clkfreq * 2 + cnt)
    Outa[24] :=1                    ' Left motor reverse
    Waitcnt (clkfreq * 2 + cnt)
    Outa[24] :=0
    repeat
```

To control the speed of a DC motor can use PWM (Pulse Width Modulation), with the following example:

**File : PWMx8.spin**

```
CON
  resolution   = 256           'The number of steps in the pulse
widths. Must be an integer multiple of 4.
  nlongs        = resolution / 4
```

```
  VAR
   long  fcb[5]
   long  pwmdata[nlongs]
   long  pinmask
   long  previndex[8]
   byte  cogno, basepin
  PUB start(base, mask, freq)
  ' This method is used to setup the PWM driver and start its cog. If a driver had
  ' already been started, it will be stopped first. The arguments are as follows:
  '   base: The base pin of the PWM output block. Must be 0, 8, 16, or 24.
  '   mask: The enable mask for the eight pins in the block:
  '         bit 0 = basepin + 0
  '         bit 1 = basepin + 1
  '         ...
  '         bit 7 = basepin + 7
  '
  '          Set a bit to 1 to enable the corresponding pin for PWM ouput.
  '
  '    freq: The frequency in Hz for the PWM output.
  '
   if (cogno)
      stop
   freq *= resolution
   if (clkfreq =< 4000000 or freq > 20648881 or clkfreq  < freq * 135 / 10 or clkfreq / freq > 40000 or base <> base & %11000 or mask <> mask & $ff or resolution <> resolution & $7ffffffc)
      return false
   basepin := base
   pinmask := mask << base
   longfill(@pwmdata, 0, nlongs)
   longfill(@previndex, 0, 8)
   fcb[0] := nlongs
   fcb[1] := freq
   fcb[2] := constant(1 << 29 | 1 << 28) | base << 6 | mask
```

```
   fcb[3] := pinmask
   fcb[4] := @pwmdata
   if (cogno := cognew(@pwm, @fcb) + 1)
      return true
   else
      return false

  PUB stop

  ' This method is used to stop an already-started PWM driver. It
returns true if
  ' a driver was running; false, otherwise.
   if (cogno)
      cogstop(cogno - 1)
      cogno~
      return true
   else
      return false

  PUB duty(pinno, value) | vindex, pindex, i, mask, unmask
  ' This method defines a pin's duty cycle. It's arguments are:
  ' pinno: The pin number of the PWM output to modify.
  ' value: The new duty cycle (0 = 0% to resolution = 100%)
  ' Returns true on success; false, if pinno or value is invalid.

   if (1 << pinno & pinmask == 0 or value < 0 or value >
resolution)
      return false
   pinno -= basepin
   mask := $01010101 << pinno
   unmask := !mask
   vindex := value >> 2
   pindex := previndex[pinno]
   if (vindex > pindex)
      repeat i from pindex to vindex - 1
        pwmdata[i] |= mask
   elseif (vindex < pindex)
      repeat i from pindex to vindex + 1
        pwmdata[i] &= unmask
```

```
  pwmdata[vindex] := pwmdata[vindex] & unmask | mask &
($ffffffff >> (31 - ((value & 3) << 3)) >> 1)
  previndex[pinno] := vindex
  return true
```

# Sensors for Intelligent Robot

## Ultrasonic Distance Sensor: PING)))™

PING)))™ ultrasonic sensor provides an easy method of distance measurement. This sensor is perfect for any number of applications that require you to perform measurements between moving or stationary objects. Interfacing to a microcontroller is a snap. A single I/O pin is used to trigger an ultrasonic burst (well above human hearing) and then "listen" for the echo return pulse. The sensor measures the time required for the echo return, and returns this value to the microcontroller as a variable-width pulse via the same I/O pin. The PING))) sensor works by transmitting an ultrasonic (well above human hearing range) burst and providing an output pulse that corresponds to the time required for the burst echo to return to the sensor. By measuring the echo pulse width, the distance to target can easily be calculated.

Key Features:

- Provides precise, non-contact distance measurements within a 2 cm to 3 m range for robotics application.
- Ultrasonic measurements work in any lighting condition, making this a good choice to supplement infrared object detectors.
- Simple pulse in/pulse out communication requires just one I/O pin.
- Burst indicator LED shows measurement in progress.
- 3-pin header makes it easy to connect to a development board, directly or with an extension cable, no soldering required.

The PING))) sensor detects objects by emitting a short ultrasonic burst and then "listening" for the echo. Under control of a host microcontroller (trigger pulse), the sensor emits a short 40 kHz (ultrasonic) burst. This burst travels through the air, hits an object and then bounces back to the sensor. The PING))) sensor provides an output pulse to the host that will terminate when the echo is detected, hence the width of this pulse corresponds to the distance to the target.

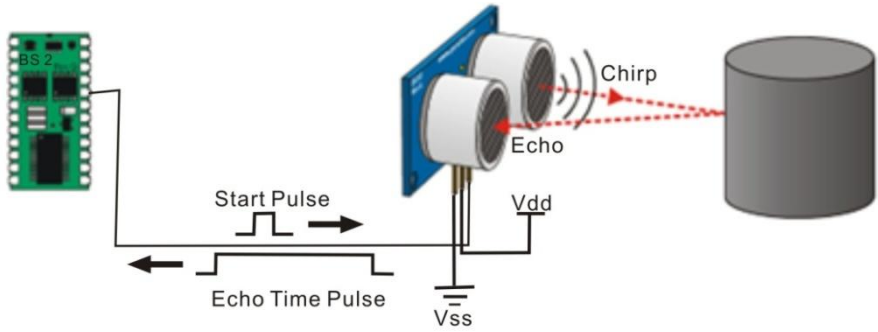


***Figure 3.6*** *The basic principle of ultrasonic distance sensor [2].*

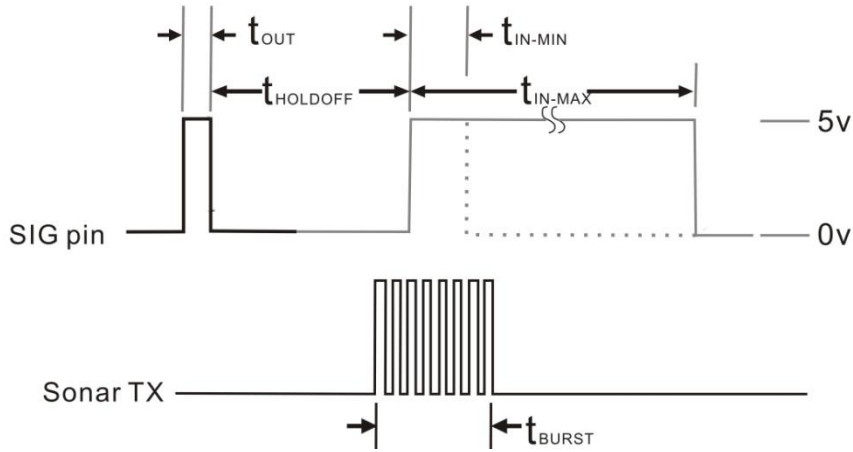


***Figure 3.7*** *Communication protocol of the PING))).*

This circuit allows you to quickly connect your PING))) sensor to a BASIC Stamp/Propeller Board. The PING))) module's GND pin connects to Vss, the 5 V pin connects to Vdd, and the SIG pin connects to I/O pin P15.

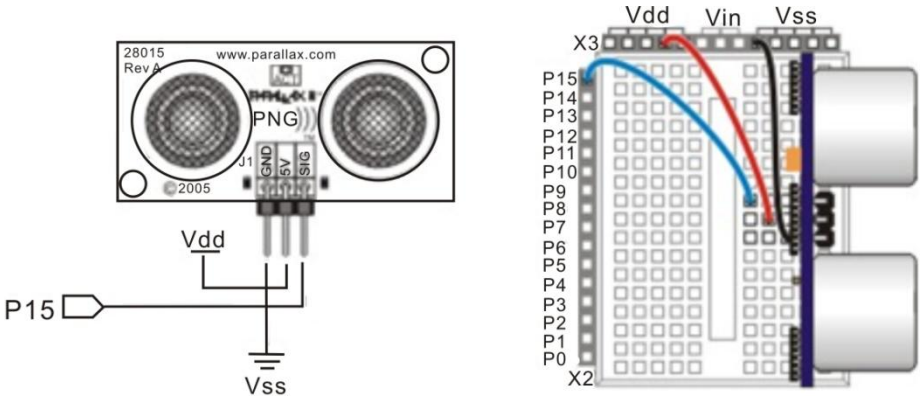


***Figure 3.8*** *PING))) to the board.*

Here is an example of using the Ping sensor shown in Serial LCD 4x20.

**File: Ping_Demo.spin**

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  PING_Pin = 15          ' I/O Pin For PING)))
  LCD_Pin = 1            ' I/O Pin For LCD
  LCD_Baud = 19_200      ' LCD Baud Rate
  LCD_Lines = 4         ' Parallax 4X20 Serial LCD (#27979)

VAR
  long  range
OBJ
  LCD: "debug_lcd"
  ping: "ping"

PUB Start
  LCD.init(LCD_Pin, LCD_Baud, LCD_Lines)     ' Initialize LCD
Object
  LCD.cursor(0)                          ' Turn Off Cursor
  LCD.backlight(true)                    ' Turn On Backlight
  LCD.cls                                ' Clear Display
  LCD.str(string("PING))) Demo", 13, 13, "Inches   -", 13,
"Centimeters -"))
```

```
    repeat                                    ' Repeat Forever
      LCD.gotoxy(15, 2)                      ' Position Cursor
      range := ping.Inches(PING_Pin)         ' Get Range In Inches
      LCD.decx(range, 2)                     ' Print Inches
      LCD.str(string(".0 "))                 ' Pad For Clarity
      LCD.gotoxy(14, 3)                      ' Position Cursor
      range := ping.Millimeters(PING_Pin) ' Get Range In
Millimeters
      LCD.decf(range / 10, 3)                ' Print Whole Part
      LCD.putc(".")                          ' Print Decimal Point
      LCD.decx(range // 10, 1)               ' Print Fractional Part
      waitcnt(clkfreq / 10 + cnt)            ' Pause 1/10 Second
```

Robot avoider is a robot that able to avoid the obstacle at the in front of the robot or at the left or right side of the robot. Here's an example using a PING))) as an avoider robot that only able to detect the obstacle in front of the robot using 1 PING))).

**Serial_LCD_Avoider.spin:**

```
' Copyright Dr. Widodo Budiharto
' www.toko-elektronika.com 2014

CON
 _clkmode = xtal1 + pll16x
 _xinfreq = 5_000_000

 LCD_PIN        = 23
  PING_Pin = 13           ' I/O Pin For PING)))
 LCD_Baud          = 19_200
  LCD_Lines=2
VAR
long range
OBJ

 Serial      : "FullDuplexSerial.spin"
 LCD         : "debug_lcd"
 ping        : "ping"

PUB Main
  Dira[27..24]:= %1111      ' Set P24  P27 to be output
```

```
   LCD.init(LCD_Pin, LCD_Baud, LCD_Lines)     ' Initialize LCD
Object
   LCD.cursor(0)                 ' Turn Off Cursor
   LCD.backlight(true)           ' Turn On Backlight
   LCD.cls
     LCD.gotoxy(3, 0)            ' Clear Display
   LCD.str(string("WIDODO.COM"))
   repeat
    range  :=  ping.Millimeters(PING_Pin)          '  Get  Range  In
Millimeters
    LCD.gotoxy(3, 1)
    LCD.decf(range / 10, 3)        ' Print Whole Part
    LCD.putc(".")                  ' Print Decimal Point
    LCD.decx(range // 10, 1)       ' Print Fractional Part
     LCD.gotoxy(10, 1)
      LCD.str(string("Cm"))
    if range >400
        Outa [24] :=0           ' Left motor stop
        Outa [27] :=0           ' Right motor stop
         waitcnt(clkfreq / 2 + cnt)        '
         Outa[25]:= 1              ' Left motor forward
         Outa[26]:= 1              ' Right motor forward
         waitcnt(clkfreq / 10 + cnt)   ' Pause 1/10 Second
    if range <=400
      'reverse
      Outa[25]:= 0              ' Left motor stop
      Outa[26]:= 0              ' Right motor stop
       waitcnt(clkfreq / 2 + cnt)        ' Pause
      Outa [24] :=1             ' Left motor reverse
      Outa [27] :=1             ' Right motor reverse
       'turn left
      Outa [24] :=1             ' Left motor reverse
      Outa [27] :=0             ' Right motor stop
      waitcnt(clkfreq/5  + cnt)        ' Pause 1/10 Second
      Outa [24] :=0             ' Left motor stop
      Outa [27] :=0             ' Right motor stop
```

Now, if we want an intelligent robot that able to avoid the obstacle using 3 PING))), we can propose the system as shown in figure 3.9.

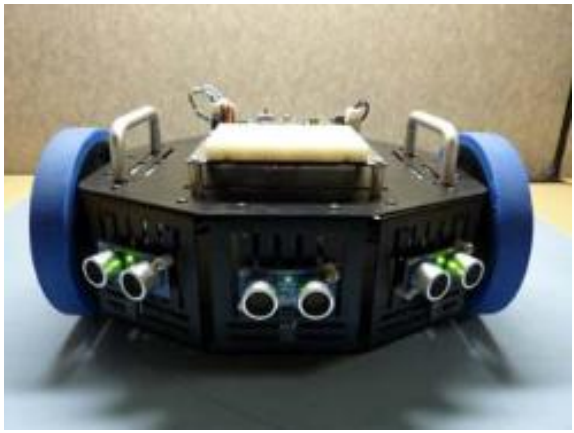

***Figure 3.9*** *Avoider robot using 3 PING))) on the body.*

**Avoider_LCD_3PING.spin**

```
' Avoider Robot, copyright Dr. Widodo Budiharto, 2014
CON
 _clkmode = xtal1 + pll16x
 _xinfreq = 5_000_000

 LCD_PIN        = 23
 PINGRight_Pin=0       ' I/O Pin For PING)))
 PINGFront_Pin = 13
 PINGLeft_Pin=22
 LCD_Baud          = 19_200
  LCD_Lines=2
VAR
long rangeFront
long rangeRight
long rangeLeft
OBJ

 Serial      : "FullDuplexSerial.spin"
 LCD         : "debug_lcd"
 ping        : "ping"
```

```
  PUB Main
   Dira[27..24]:= %1111       ' Set P24  P27 to be output

  LCD.init(LCD_Pin, LCD_Baud, LCD_Lines)  ' Initialize LCD Object
  LCD.cursor(0)                    ' Turn Off Cursor
  LCD.backlight(true)              ' Turn On Backlight
  LCD.cls
  LCD.gotoxy(3, 0)                 ' Clear Display
  LCD.str(string("WIDODO.COM"))
  waitcnt(clkfreq/2  + cnt)        ' Pause 1/10 Second
  repeat

   rangeFront := ping.Millimeters(PINGFront_Pin)    ' Get Range In
Millimeters
   rangeRight := ping.Millimeters(PINGRight_Pin)    ' Get Range In
Millimeters
   rangeLeft  := ping.Millimeters(PINGLeft_Pin)     ' Get Range In
Millimeters
   LCD.gotoxy(0, 1)
   LCD.decf(rangeLeft / 10, 3)         ' Print Whole Part
   LCD.gotoxy(5, 1)
   LCD.decf(rangeFront / 10, 3)        ' Print Whole Part
   LCD.putc(".")                       ' Print Decimal Point
   LCD.decx(rangeFront // 10, 1)     ' Print Fractional Part
   LCD.gotoxy(12, 1)
   LCD.decf(rangeRight / 10, 3)

   if rangeFront >200  and rangeRight>200
     LCD.cls
     LCD.gotoxy(3, 0)          ' Clear Display
     LCD.str(string("FORWARD"))
     Outa [24] :=0             ' Left motor stop
     Outa [27] :=0             ' Right motor stop
     waitcnt(clkfreq / 2 + cnt)        '
     Outa[25]:= 1                    ' right motor forward
     Outa[26]:= 1                    ' left motor forward
     waitcnt(clkfreq / 10 + cnt)     ' Pause 1/10 Second
   if rangeFront <=200
     LCD.cls
```

```
    'reverse
    LCD.gotoxy(3, 0) ' Clear Display
    LCD.str(string("REFERSE"))
    Outa[25]:= 0                  ' left motor stop
    Outa[26]:= 0                  ' right motor stop
    waitcnt(clkfreq / 5 + cnt)    ' Pause
    Outa [24] :=1                 ' Left motor reverse
    Outa [27] :=1                 ' Right motor reverse
    waitcnt(clkfreq  + cnt)       ' Pause
   if  rangeRight<=200
    LCD.cls
    'turn left
    LCD.gotoxy(3, 0)  ' Clear Display
    LCD.str(string("TURN LEFT"))
    Outa [24] :=0                   ' Left motor stop
    Outa [27] :=0                   ' Right motor stop
    waitcnt(clkfreq/10  + cnt)      ' Pause 1/10 Second
    Outa[25]:= 1                    ' left motor forward
    waitcnt(clkfreq/2  + cnt)       ' Pause 1/10 Second
    Outa[25]:= 0                    ' left motor stop
  if  rangeLeft<=200
    LCD.cls
    'turn right
    LCD.gotoxy(3, 0)  ' Clear Display
    LCD.str(string("TURN RIGHT"))
    Outa [24] :=0                   ' Left motor stop
    Outa [27] :=0                   ' Right motor stop
    waitcnt(clkfreq/10  + cnt)      ' Pause 1/10 Second
    Outa[26]:= 1                    ' right motor forward
    waitcnt(clkfreq/2  + cnt)       ' Pause 1/10 Second
    Outa[26]:= 0                    ' right motor stop
```

## Compass Module: 3-Axis HMC5883L

The Compass Module 3-Axis HMC5883L is designed for low-field magnetic sensing with a digital interface. This compact sensor fits into small projects such as UAVs and robot navigation systems. The sensor converts any magnetic

field to a differential voltage output on 3 axes. This voltage shift is the raw digital output value, which can then be used to calculate headings or sense magnetic fields coming from different directions.

Key Features:

- Measures Earth's magnetic fields.
- Precision in-axis sensitivity and linearity.
- Designed for use with a large variety of microcontrollers with different voltage requirements.
- 3-Axis magneto-resistive sensor.
- 1 to 2 degree compass heading accuracy.
- Wide magnetic field range (+/-8 gauss).
- Fast 160 Hz maximum output rate.
- Measures Earth's magnetic field, from milli-gauss to 8 gauss.

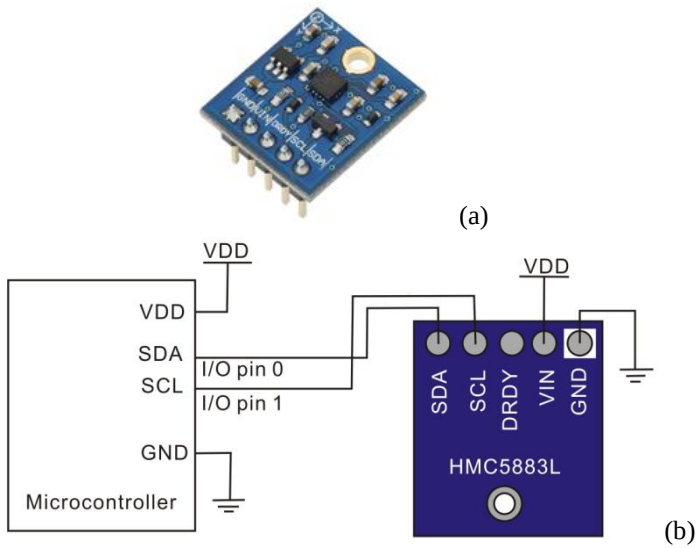


***Figure 3.10*** *Compass module (a) and the schematic (b).*

Here is an example code for using Compass module:

**DemoCompass.spin:**

```
OBJ
 pst  :  "FullDuplexSerial"  ' Comes with Propeller Tool
CON

 _clkmode      = xtal1 + pll16x
 _clkfreq      = 80_000_000

 datapin  = 1          ' SDA of compass to pin P1
 clockPin = 0          ' SCL of compass to pin P0

 WRITE_DATA     = $3C ' Requests Write operation
 READ_DATA      = $3D ' Requests Read operation
 MODE           = $02 ' Mode setting register
 OUTPUT_X_MSB   = $03 ' X MSB data output register

VAR

 long x
 long y
 long z

PUB Main

 waitcnt(clkfreq/100_000 + cnt)      ' Power up delay
 pst.start(31, 30, 0, 115200)
 SetCont
 repeat
  SetPointer(OUTPUT_X_MSB)
  getRaw                ' Gather raw data from compass
  pst.tx(1)
  ShowVals

PUB SetCont
 ' Sets compass to continuous output mode

 start
 send(WRITE_DATA)
 send(MODE)
 send($00)
 stop

PUB SetPointer(Register)
```

```
 ' Start pointer at user specified register (OUT_X_MSB)

 start
 send(WRITE_DATA)
 send(Register)
 stop

PUB GetRaw
 ' Get raw data from continuous output

 start
 send(READ_DATA)
 x := ((receive(true) << 8) | receive(true))
 z := ((receive(true) << 8) | receive(true))
 y := ((receive(true) << 8) | receive(false))
 stop
 ~~x
 ~~z
 ~~y
 x := x
 z := z
 y := y

PUB ShowVals
' Display XYZ compass values

 pst.str(string("X="))
 pst.dec(x)
 pst.str(string(", Y="))
 pst.dec(y)
 pst.str(string(", Z="))
 pst.dec(z)
 pst.str(string("    "))

PRI send(value)

 value := ((!value) >< 8)

 repeat 8
  dira[dataPin]  := value
  dira[clockPin] := false
```

```
   dira[clockPin] := true
   value >>= 1

  dira[dataPin]  := false
  dira[clockPin] := false
  result         := !(ina[dataPin])
  dira[clockPin] := true
  dira[dataPin]  := true

PRI receive(aknowledge)

  dira[dataPin] := false

  repeat 8
   result <<= 1
   dira[clockPin] := false
   result         |= ina[dataPin]
   dira[clockPin] := true

  dira[dataPin]  := aknowledge
  dira[clockPin] := false
  dira[clockPin] := true
  dira[dataPin]  := true

PRI start

  outa[dataPin]  := false
  outa[clockPin] := false
  dira[dataPin]  := true
  dira[clockPin] := true

PRI stop

  dira[clockPin] := false
  dira[dataPin]  := false
```

## Gyroscope Module 3-Axis L3G4200D

The Gyroscope Module is a low power 3-Axis angular rate sensor with temperature data for UAV, IMU Systems, robotics and gaming. The gyroscope shows the rate of change in rotation on its X, Y and Z axes. Raw angular rate and temperature data measurements are accessed from the selectable digital I2C or SPI interface. The small package design and SIP interface accompanied by

the mounting hole make the sensor easy to integrate into your projects. Designed to be used with a variety of microcontrollers, the module has a large operating voltage window.

Key Features:

- 3-axis angular rate sensor (yaw, pitch & roll) make it great for model aircraft navigation systems.
- Supports both I2C and SPI for whichever method of communication you desire.
- Three selectable scales: 250/500/2000 degrees/sec (dps).
- Embedded power down and sleep mode to minimize current draw.
- 16 bit-rate value data output.

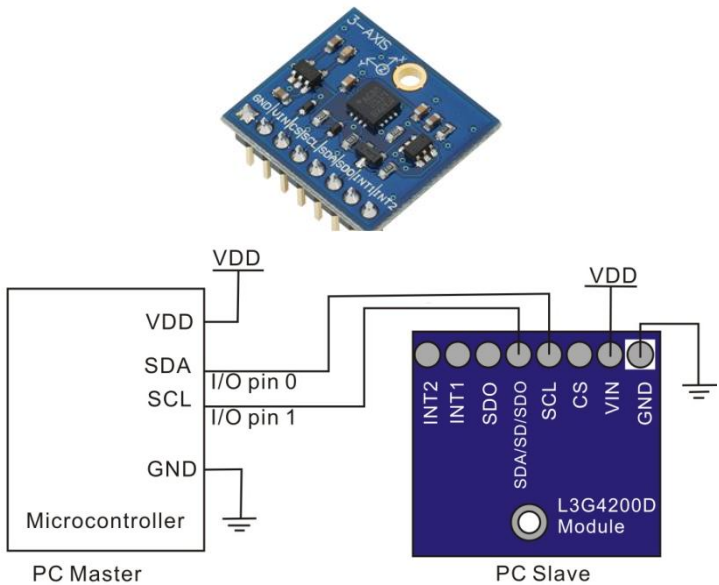


***Figure 3.11*** *Gyroscope Module 3-Axis L3G4200D (a) and general schematic (b).*

Program below demonstrates X, Y, Z output to a serial terminal and uses default (I C) interface on the Gyroscope mo dule.

**Gyro_Demo.spin**

```
CON
```

```
  _clkmode      = xtal1 + pll16x
  _clkfreq      = 80_000_000

  SCLpin         = 2
  SDApin         = 4

  '****Registers****

  WRITE         = $D2
  READ          = $D3

  CTRL_REG1     = $20    'SUB $A0
  CTRL_REG3     = $22
  CTRL_REG4     = $23
  STATUS_REG    = $27
  OUT_X_INC     = $A8

  x_idx = 0
  y_idx = 1
  z_idx = 2

 VAR

  long x
  long y
  long z

  long cx
  long cy
  long cz

  long ff_x
  long ff_y
  long ff_z

  long multiBYTE[3]

 OBJ
   Term      : "FullDuplexSerial"

 PUB Main | last_ticks

  ''Main routine for example program - Shows RAW X,Y,Z data and
example of calculated data for degrees
```

```
  term.start(31, 30, 0, 115200)       'start a terminal Object
(rxpin, txpin, mode, baud rate)

  Wrt_1B(CTRL_REG3, $08)         'set up data ready signal
  Wrt_1B(CTRL_REG4, $80)         'set up "block data update" mode
(to avoid bad reads when the values would get updated while we
are reading)
  Wrt_1B(CTRL_REG1, $1F)         'write a byte to control
register one (enable all axis, 100Hz update rate)

  Calibrate

  last_ticks := cnt

  repeat             'Repeat indefinitely

   term.tx(1)        'Set Terminal data at top of screen

   WaitForDataReady
   Read_MultiB(OUT_X_INC)       'Read out multiple bytes starting
at "output X low byte"

   x := x - cx                  'subtract calibration out
   y := y - cy
   z := z - cz

   ' at 250 dps setting, 1 unit = 0.00875 degrees,
   ' that means about 114.28 units = 1 degree
   ' this gets us close
   x := x / 114
   y := y / 114
   z := z / 114

   RawXYZ           'Print the Raw data output of X,Y and Z

  PUB RawXYZ

  ''Display Raw X,Y,Z data
  term.str(string("RAW X ",11))
  term.dec(x)
  term.str(string(13, "RAW Y ",11))
  term.dec(y)
  term.str(string(13, "RAW Z ",11))
  term.dec(z)
```

```
  PUB Calibrate
   cx := 0
   cy := 0
   cz := 0

   repeat 25
    WaitForDataReady
    Read_MultiB(OUT_X_INC)      ' read the 3 axis values and
accumulate
    cx += x
    cy += y
    cz += z

   cx /= 25                     ' calculate the average
   cy /= 25
   cz /= 25

  PUB WaitForDataReady | status
   repeat
    status := Read_1B(STATUS_REG)          ' read the ZYXZDA bit
of the status register (looping until the bit is on)
    if (status & $08) == $08
     quit

  PUB Wrt_1B(SUB1, data)

   ''Write single byte to Gyroscope.

    start
    send(WRITE)                    'device address as write
command
     'slave ACK
    send(SUB1)                     'SUB address = Register MSB 1 =
reg address auto increment
     'slave ACK
    send(data)                     'data you want to send
     'slave ACK
    stop

  PUB Wrt_MultiB(SUB2, data, data2)

  ''Write multiple bytes to Gyroscope.
```

```
    start
    send(WRITE)          'device address as write command
     'slave ACK
    send(SUB2)         'SUB address = Register MSB 1 = reg address
auto increment
     'slave ACK
    send(data)                     'data you want to send
  'slave ACK
  send(data2)                    'data you want to send
  'slave ACK
  stop

  PUB Read_1B(SUB3) | rxd

  ''Read single byte from Gyroscope

    start
    send(WRITE)            'device address as write command
    'slave ACK
    send(SUB3)             'SUB address = Register MSB 1 = reg
address auto increment
    'slave ACK
    stop
    start                  'SR condition
    send(READ)             'device address as read command
     'slave ACK
    rxd := receive(false)          'recieve the byte and put in
variable rxd
    stop
   result := rxd

  PUB Read_MultiB(SUB3)
  ''Read multiple bytes from Gyroscope
    start
    send(WRITE)          'device address as write command
    'slave ACK
    send(SUB3)           'SUB address = Register MSB 1 = reg
address auto increment
    'slave ACK
    stop
```

```
    start                'SR condition
    send(READ)           'device address as read command
    'slave ACK
    multiBYTE[x_idx] := (receive(true)) | (receive(true)) << 8
'Receives high and low bytes of Raw data
    multiBYTE[y_idx] := (receive(true)) | (receive(true)) << 8
    multiBYTE[z_idx] := (receive(true)) | (receive(false)) << 8
    stop

    x := ~~multiBYTE[x_idx]
    y := ~~multiBYTE[y_idx]
    z := ~~multiBYTE[z_idx]

  PRI send(value)  ' I²C Send data - 4 Stack Longs

   value := ((!value) >< 8)

   repeat 8
    dira[SDApin]      := value
    dira[SCLpin]      := false
    dira[SCLpin]      := true
    value >>= 1
   dira[SDApin]      := false
   dira[SCLpin]      := false
   result            := not(ina[SDApin])
   dira[SCLpin]      := true
   dira[SDApin]      := true

  PRI receive(aknowledge) ' I²C receive data - 4 Stack Longs
   dira[SDApin]     := false

   repeat 8
   result <<= 1
   dira[SCLpin]     := false
   result            |= ina[SDApin]
   dira[SCLpin]      := true
   dira[SDApin]      := (aknowledge)
   dira[SCLpin]      := false
   dira[SCLpin]      := true
   dira[SDApin]      := true
```

```
PRI start ' 3 Stack Longs
 outa[SDApin]       := false
 outa[SCLpin]       := false
 dira[SDApin]       := true
 dira[SCLpin]       := true
PRI stop ' 3 Stack Longs
 dira[SCLpin]       := false
 dira[SDApin]       := false
```

## PID Controller for the Robot

A PID controller is used to make a quantity (like position) reach a target value (a target position). The first thing a PID controller does is to calculate the error *e(t).* The PID controller algorithm involves three separate constant parameters, and is accordingly sometimes called three-term control: the proportional, the integral and derivative values, denoted *P*, *I*, and *D.* Simply put, these values can be interpreted in terms of time: *P* depends on the *present* error, *I* on the accumulation of *past* errors, and *D* is a prediction of *future* errors, based on current rate of change. The weighted sum of these three actions is used to adjust the process via a control element such as the position of a motor. The controller attempts to minimize the error by adjusting (an Output). The model of PID Controller shown in fig. 3.11:

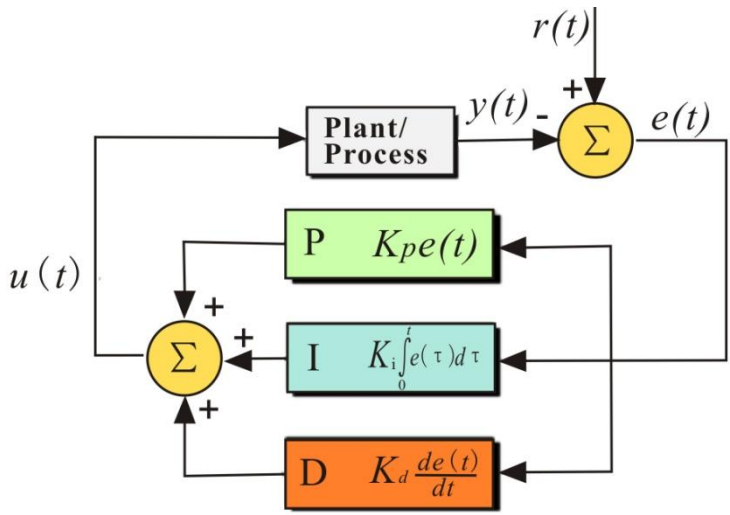


***Figure 3.12*** *General PID Controller.*

The output of a PID controller, equal to the control input to the system, in the time-domain is as follows:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt} \quad (3.1)$$

In Propeller microcontroller, we can use Propeller Object Exchange named A quadrature encoder and PID controller driver that runs in one cog. The code has been fully optimized with a super simple spin interface for maximum speed and is also fully commented. It provides full support for getting the quadrature encoder's current position and position delta in ticks and setting the quadrature encoders current speed in ticks per second through PID control through a standard DC motor.

## Exercises

1) Write a motor speed controller using PID.
2) Write a program for fire fighter robot using flame sensor, distance sensor and compass to follow the side of the wall.

## References

[1] Crustcrawler.com.

[2] www.parallax.com.