LINE FOLLOWING ROBOT WITH MAZE SOLVING APPLICATION

S V National Institute of Technology, Surat

AUTONOMOUS ROBOT

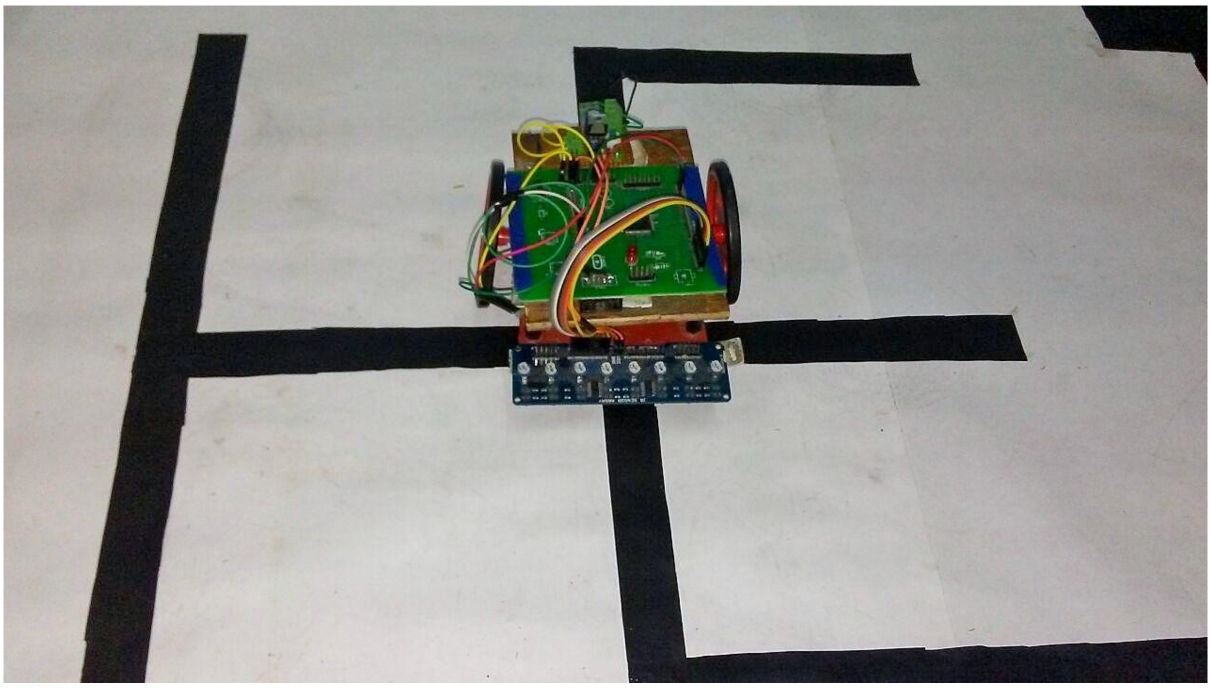
An autonomous robot is one with the ability to behave by taking decisions autonomously in accordance with its environment, with little or no human intervention.

An autonomous robot is able to gain information about its environment and work for an extended period without human assistance.

LINE FOLLOWER ROBOT

A line follower is a very basic autonomous robot which can be programmed to follow a black line on a white surface or a white line on a black surface.

It is basically a robot which follows a particular path or trajectory and decides its own course of action which interacts with obstacle.



BASIC METHOD OF LINE FOLLOWING

In order to detect a line, let’s say a black line on a white surface or vice versa, we use infrared or IR sensors.

IR sensors work by using a specific light sensor to detect a selected light wavelength in the invisible IR spectrum.

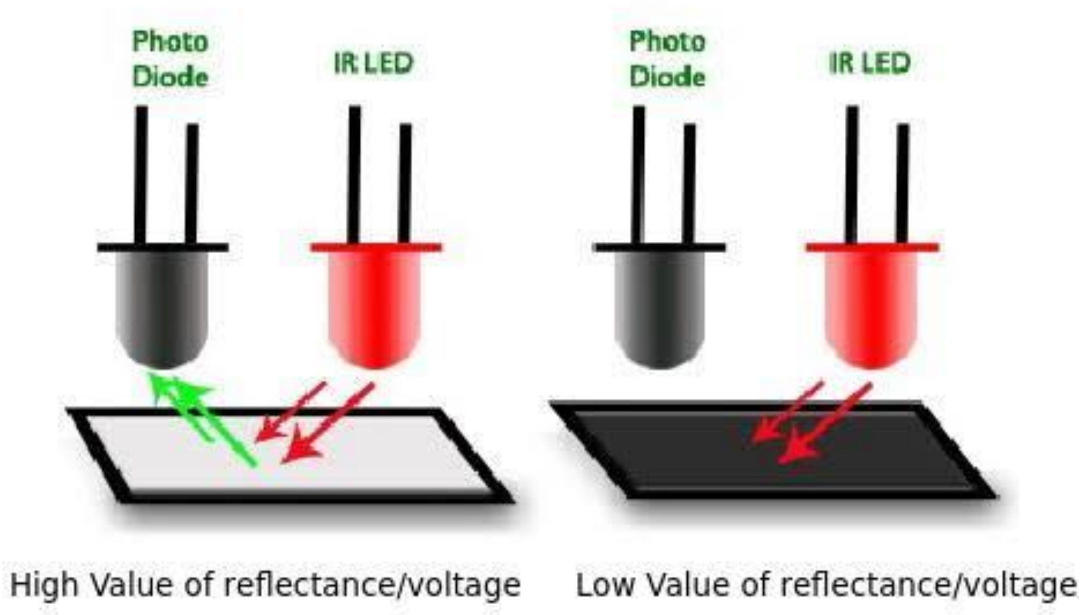


Photo Diode
IR LED
Photo Diode
IR LED
High Value of reflectance/voltage
Low Value of reflectance/voltage

An IR sensor consists of two diodes- an Infrared Light Emitting Diode (IR LED) and a Photo Diode.

An IR LED emits infrared rays. These rays are almost completely absorbed by a black surface (low reflectance) and almost completely reflected by a white surface (high reflectance).

A Photo Diode conducts electricity when IR rays fall on it, otherwise no current will pass through it.

When working in pair, these can used to identify a black or white surface. When there is black surface the IR rays will be absorbed and current through photodiode will be zero (the sensor will return 0 logic value).

When there is white surface the IR rays will be reflected and current through photodiode will be non-zero (the sensor will return 1 logic value).

Hence by using an IR sensor array, we can detect a black line on a white surface. For example, if an 8 IR sensor array returns values, S1=0, S2=0, S3=0, S4=1, S5=1, S6=0, S7=0, S8=0, then the black line is going through the middle of the sensor array (Sn represents pins connected to each sensor and returning a logic value 0 or 1). This value can be represented as an 8 digit binary number 11100111.

Hence combination of different values can be used to detect the position of the black line on the white surface and the bot can be accordingly programmed to achieve line following.

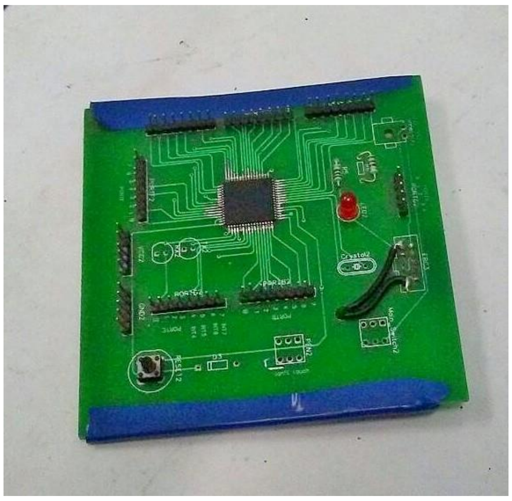
The IR sensors used have a potentiometer attached with the circuit. This potentiometer can set the range of distance for which the sensor has to work. Hence sensors can be calibrated for different positions.

There is also an advanced auto-calibrating and more accurate line sensor array available (e.g. model LSA08), which converts Boolean logic values like 11100111 into their corresponding decimal value, here 231, which is given as output and can be used as input to microcontroller.

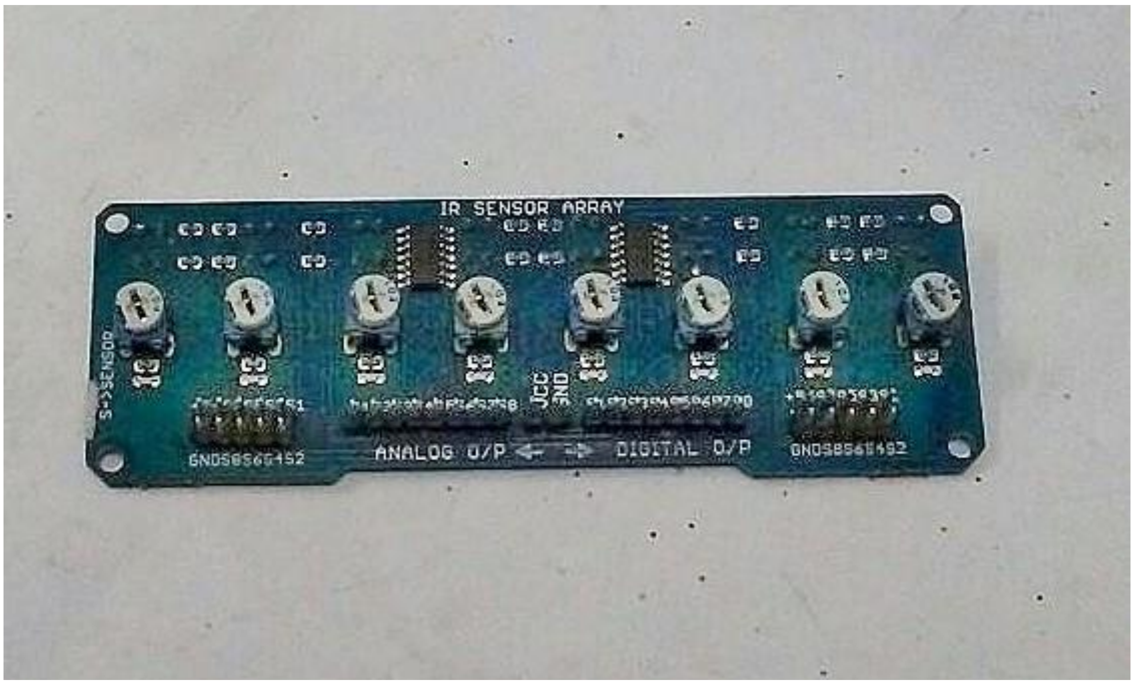
PARTS USED BY US

- A two floored chassis with two wheels
- Two toy motors having 1000 rpm
- ATmega 128 microcontroller
- Cytron motor driver circuit
- 12 V Li-ion battery
- 8 IR sensor array
- A voltage divider circuit
- Jumper wires to make connections
- Double-sided and electric insulation tapes

Two floored chassis with toy motors and wheels mounted on it


ATmega128 microcontroller

Cytron motor driver circuit



8 IR line sensor array

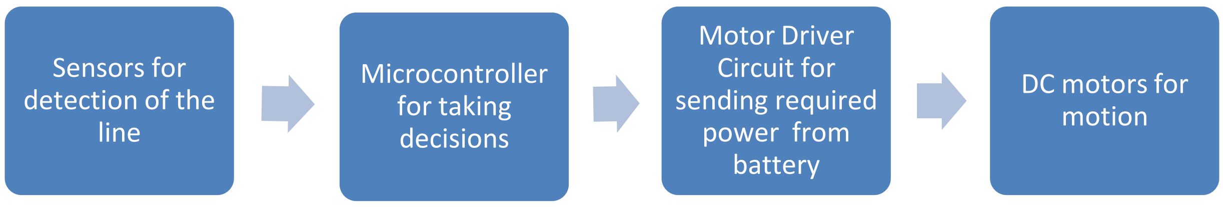
Voltage divider circuit

12V Li-ion battery

Jumper wires

BASIC DESIGN OF THE BOT

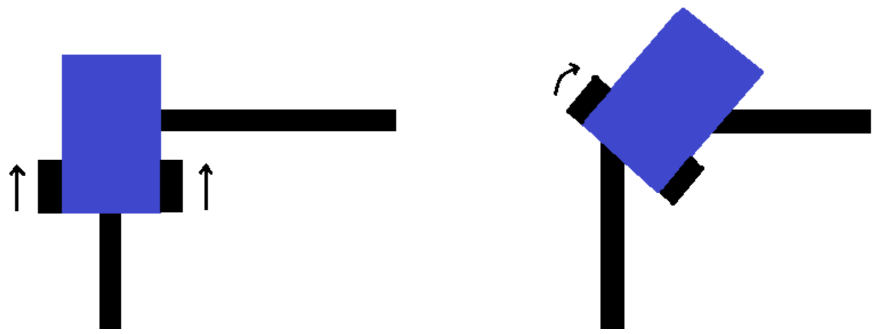
The line following robot used in this project can be considered to be made of four major blocks.


Sensors for detection of the line
Microcontroller for taking decisions
Motor Driver Circuit for sending required power from battery
DC motors for motion

- The robot is powered using a 12v rechargeable Li-ion battery. A voltage divider circuit is made using the LM7805 IC. This circuit receives 12v supply from the battery and gives 5v supply to the microcontroller and 12v supply to the motor driver circuit.
- The microcontroller receives input from the sensors and gives digital signals as output, ranging between 0 and 5v, to the motor driver. The motor driver sets this signal on a scale of 0 to 12v and provides running power from the battery to the motors.
- The motors do rotation accordingly and give motion to the robot. The motor driver used here, Cytron also provides direction pins.

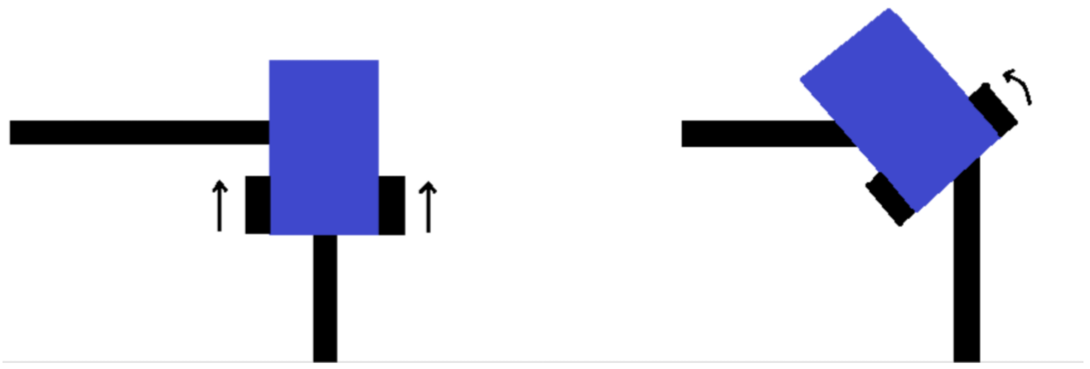
  The pins given on the driver are dir1, pwm1, dir2, pwm2 and gnd. The pwm pins are for receiving voltage levels from the microcontroller. The dir pin when set high gives a motor rotation in a specific direction and when set low, in the opposite direction. The gnd pin is for grounding purposes.

TURNING MECHANISM OF THE BOT



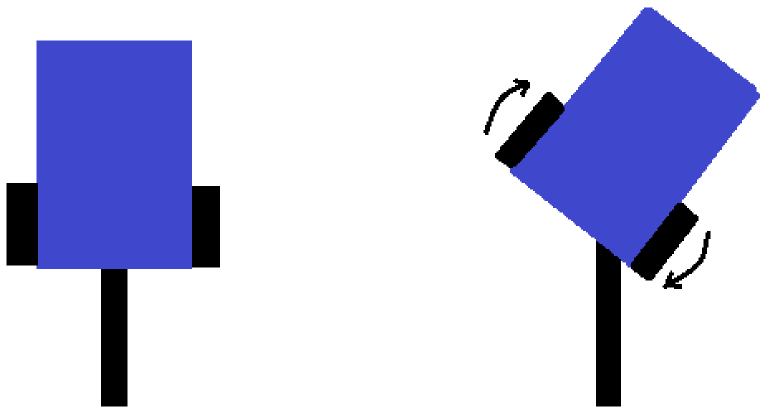
right turning of the bot

To make the bot turn right, the right wheel is stopped and the left wheel continues to rotate till the bot again comes back on straight path, after which both wheels start rotating with same speed.



left turning of the bot

To make the bot turn left, the left wheel is stopped and the right wheel continues to rotate till the bot again comes back on straight path, after which both wheels start rotating with same speed.



reverse turning at dead end

At a dead end in the path, the bot needs to do a reverse turn. For achieving this, both the motors are made to rotate in reverse direction, till the bot comes back on straight path. After this, the bot moves with both motors in forward direction.

In each case, the turning of the wheels has to be tuned for time intervals using delay functions in the program of the bot.

BASIC CODE FOR LINE FOLLOWING

A basic code in C, implementing these techniques is given. The port D is being used as the input port, D0 connected to leftmost sensor and D7 connected to rightmost sensor (there are 8 sensors). Pins B5 and B6 are used for providing output to the motors, 5 for the left motor and 6 for the right motor. Pins A1 and A2 are being used as direction pins, A1 for left motor and A2 for right motor.

```
#include <avr/io.h>
#include <util/delay.h>

int readSensors();                         //function for reading sensor output

int a,i, s[8];

int main(void)
{
  DDRD=0x00;              //declaring D port as input port
  DDRB|=0xFF;             //declaring B port as output port
  DDRA|=0xFF;             //declaring A port as output port

  while(1)
  {
    a = readSensors();
    PORTA = 0b01100000;

    if ((a==231) || (a==243) || (a==207))       //11100111,11110011,11001111 (straight path)
    {
      PORTB = 0b00000110;
    }
    else if ((a==224 ) || (a==240) || (a==248))  //11100000,11110000,11111000 (right turn)
    {
      PORTB = 0b00000100;
      _delay_ms(500);
    }
    else if ((a==7 ) || (a==15) || (a==31))           //00000111,00001111,00011111 (left turn)
    {
      PORTB = 0b00000010;
      _delay_ms(500);
    }
    else if (a==255)               //11111111 (reverse turn)
    {
      PORTA = 0b01000000;
      PORTB = 0b00000110;
      _delay_ms(500);
    }
  }
}

int readSensor()
{
    s[8] = 0;
    int  k=0;
    for(i=0;i<8;i++)
    {
      if(bit_is_set(PIND,i)    {s[i] = 1;}
      else                      {s[i] = 0;}
    }
    k = s[7] + s[6]*2 + s[5]*4 + s[4]*8 + s[3]*16 + s[2]*32 + s[1]*64 + s[0]*128;
                                                   // leftmost to s[0], rightmost to s[7]
    return(k);
  }
```

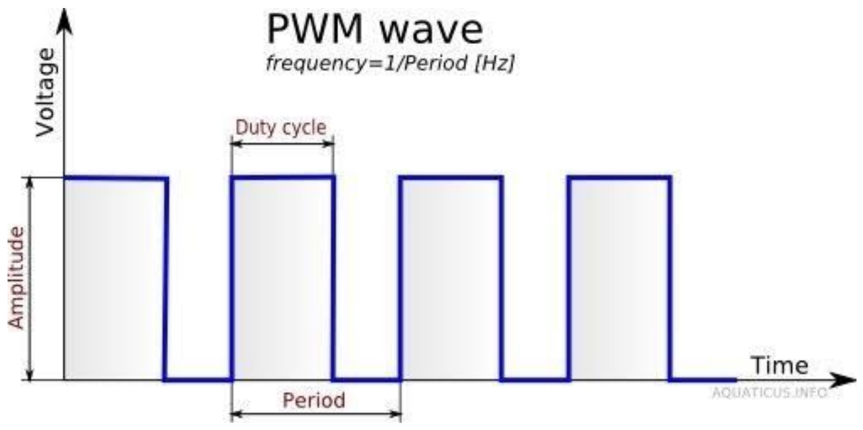
In the above system, the pins PB5 and PB6 are connected to pwm1 and pwm2 of motor driver and hence to the left motor and the right motor respectively, for giving voltage levels. The pins PA1 and PA2 are connected to dir1 and dir2 on the motor driver respectively and are responsible for giving direction of rotation and grounding gnd is provided to the driver from the MCU.

The delay values to be given to the _delay_ms() function has to be tuned experimentally for different turnings. The delay values decreases with increase in the rpm of motors used.

The above code is very basic and won't give a very stable bot. For that purpose we need to apply pulse width modulation and PID control algorithm techniques.

PULSE WIDTH MODULATION (PWM)

A digital device like a microcontroller can give only two voltage values, 5V or 0V (5V for logic 1 and 0V for logic 0). This 0-5V voltage range can be magnified to a 0-12V voltage range with the help of an external power source and a motor driver circuit (higher voltage ranges can be obtained with other batteries and other motor drivers). So if a MCU returns these two values only, it can either run a motor at full rpm or 0 rpm. If we need to obtain a speed less than full, we need to obtain a voltage less than 5V from the MCU. This can be achieved through pulse width modulation (PWM) technique.



Pulse-width modulation uses a rectangular pulse wave whose pulse width is modulated resulting in the variation of the average value of the waveform. Hence analog voltage levels between 0 to 5 volts can be obtained with the help of this technique.

For a time period T of the wave, let the voltage be high for time T_{on} and be low for time $T_{off.}$ Then duty cycle of the wave is defined as, Duty cycle = $\frac{\text{Ton}}{\text{Ton+Toff}} \times 100\%$ = $\frac{\text{Ton}}{\text{T}} \times 100\%$

The voltage obtained as output is averaged over the entire waveform.

Output Voltage = (Duty Cycle %) × maximum voltage.

Maximum voltage in the case of an atmega microcontroller is 5 volts.

Since this technique doesn’t give any real analog voltage but the output voltage appears to be at an analog level, the speed of the PWM signal should be very fast to achieve this goal. Hence PWM is possible only with high frequency devices.

TIMER1 AND PWM GENERATION

Microcontrollers have dedicated circuits for specific applications. Such circuits are called peripherals. Timers are important peripherals.

A timer basically counts from 0 to a TOP value and then overflows back to 0. When it overflows it signals the processor for some action, if some action has to take place.

Here we will use a 16 bit timer, timer1 of the ATmega128 chip for generation of Fast PWM wave. For a 16 bit timer the maximum limit of the TOP value is 2^{16} i.e. 65536.

The 16-bit timer, Timer 1 on ATmega128 can be accessed through pins PB5 and PB6, referred as two channels of Timer1, channel A (PB5) and channel B (PB6).

Specific values have to be set in the Timer/Counter Control Registers (TCCR1A and TCCR1B) for Fast PWM generation.

Timer/Counter1 Control Register A (TCCR1A)

Bit	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10
Initial Value	0	0	0	0	0	0	0	0

All the bits in the TCCR1A register are read/write type.

Timer/Counter1 Control Register B (TCCR1B)

Bit	7	6	5	4	3	2	1	0
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Initial Value	0	0	0	0	0	0	0	0

All the bits in the above register are read/write type except register 5.

The clock speed to be used in the program is defined in the beginning of the program in the following manner,

```
#define F_CPU 8000000UL
```

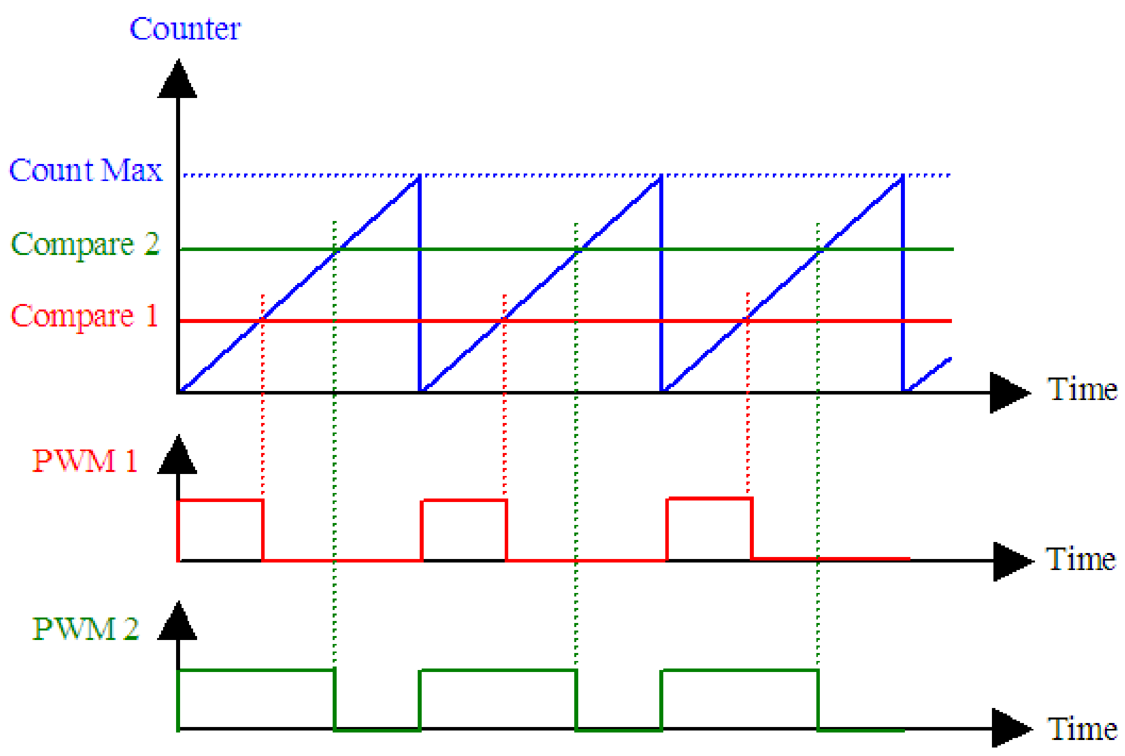
This defines a processing speed of 8MHz for the program, referred as clock source output Clk$_{I/O}$. The clock source for the timer can be selected using the Prescalar.

The prescalar is an electronic counting circuit used to reduce a high frequency electrical signal to a low frequency by integer division. The prescalar can be accessed using the CS1n (clock select) bits in TCCR1B. A fraction of the frequency of F_CPU can be used for timer using prescalar.

CS12	CS11	CS10	
0	0	0	no clock (timer stopped)
0	0	1	Clk$_{I/O}$/1 (no prescaling)
0	1	0	Clk$_{I/O}$/8 (from prescalar)
0	1	1	Clk$_{I/O}$/64 (from prescalar)
1	0	0	Clk$_{I/O}$/256 (from prescalar)
1	0	1	Clk$_{I/O}$/1024 (from prescalar)
1	1	0	external clock source
1	1	1	external clock source

The clock selected from above list would be used for timer1 functioning.

The main concept behind Fast PWM is explained below.



Generation of Fast PWM wave using Timer1

The timer keeps counting from 0 to a TOP value (Count Max) provided by the program. For 16 bit timer1, its maximum value can be 65536.

The program also provides two compare values, compare 1 and 2, lower than the TOP value, for the two channels of timer 1, A and B respectively.

Now in one cycle of counting, i.e. from 0 to TOP, the controller gives a high voltage in channel 1 till the counter value becomes equal to compare 1, after which it gives 0v for the rest part of the cycle and again repeats the same in the next cycle. Hence, pulse width modulation is achieved.

The output voltage is proportional to the pulse width, which in turn is proportional to the ratio of compare and TOP values.

Hence,

$$\text{Output Voltage} = \frac{\text{compare value}}{\text{TOP value}} \times maximum\ voltage$$

The maximum voltage in the case of a microcontroller is 5v. It can be set on a scale of higher voltage, like 12v, by using proper batteries and driver circuits.

The registers that are being used here for storing the TOP and compare values are,

- ICR1 (Input Capture Register 1) – it will store the TOP value. Its maximum limit is 2^{16} (65536).
- OCR1A (Output Compare Register 1A) – it will store the compare value for channel A (output voltage at PB5).
- OCR1B (Output Compare Register 1B) – it will store the compare value for channel B (output voltage at PB6).

BASIC CODE FOR FAST PWM GENERATION

```
#define F_CPU 8000000UL

void pwm_init();

void main()
{
  DDRB |= 0xFF;
  pwm_init();

  while (1)
  {
    OCR1A = 800;          //one-fifth of maximum output
    OCR1B = 2000;       //half of maximum output
  }
}

void pwm_init()
{
    TCCR1A |= (1<<COM1A1) | (1<<COM1B1) | (1<<WGM11);
    TCCR1B |= (1<<WGM12) | (1<<WGM13) | (1<<CS10); //no prescaling
    ICR1 = 4000;        //top value for maximum output
}
```

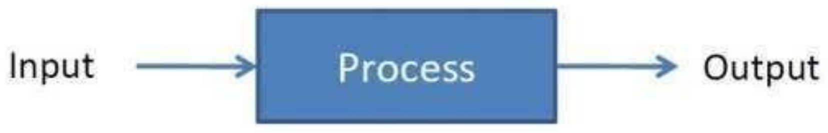
In above code, ICR1 is 4000 which sets the upper limit. OCR1A is 800, so one-fifth of maximum output through channel A (PB5) and OCR1B is 2000, so half of the maximum output through channel B (PB6). Hence Fast PWM output is achieved.

CONTROL SYSTEMS BASICS

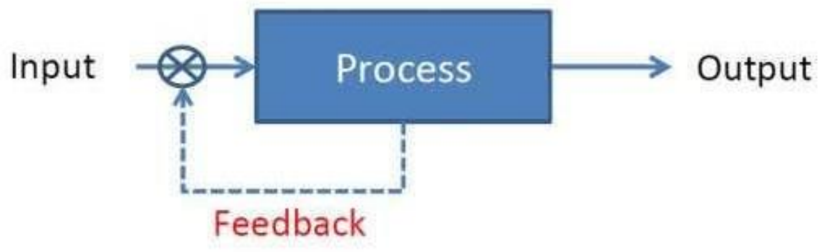
When a number of elements are combined together to form a system which receives some input and produces desired output then the system is referred as control system.

There are two types of control systems:-

1. **Open loop control systems** (non-feedback controller): In this system, the control action from the controller is independent of the process output, which is the process variable (PV) that is being controlled. A common example is electric drier; hot air comes out from it till a hand is present, irrespective of how much the hand has dried.



2. **Closed loop control systems** (feedback controller): In this system, the control action from the controller is dependent on the feedback from the process in the form of the value of the process variable (PV). A common example is that of an air conditioner, whose functioning depends upon the temperature of the room.

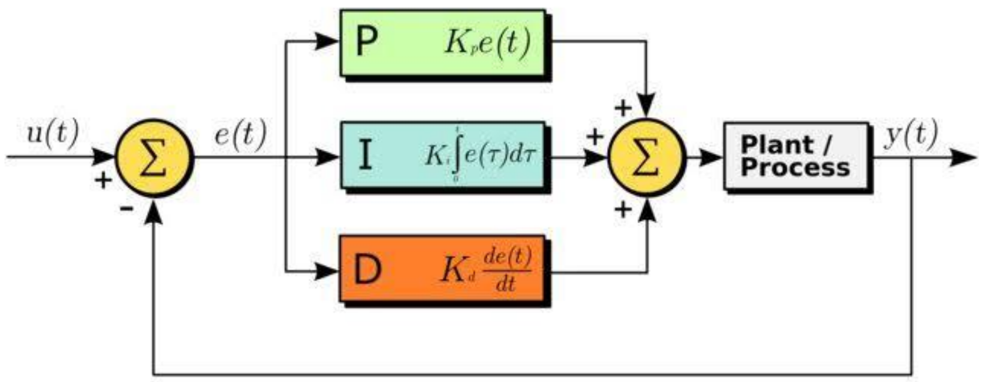


PROPORTIONAL-INTEGRAL-DERIVATIVE CONTROLLER (PID)

A proportional-integral-derivative controller (PID controller) is a closed loop control mechanism used to achieve error reduction in various systems.

A PID controller takes the feedback input of the process variable (PV) to be controlled. The Set Point (SP) is the desired value of PV which is to be achieved. The controller calculates error value *e(t)*, as the difference between the set point and the process variable and applies correction.

Hence error value *e(t)* = SP – PV(t), where SP is set point and PV is the process variable.



The correction value *v(t)* which is to be added to the process variable to reduce the error is given by the equation,

$$v(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

This correction term is added to the base value y_o of output variable *y(t)* to reduce error. Hence,

$y(t) = y_0 + v(t)$ is the output of the controller.

The way a PID controller corrects can be understood by considering all the three parameters separately. The controller tries to reduce the difference between set point and process variable, i.e. error value to zero.

- Proportional control (P): The most basic idea is, larger the error, larger should be the correction i.e. the correction term is proportional to the error produced, positive or negative. It accounts for the present value of error.
- Integral control (I): This accounts for the past values of error. This term increases corrective action not only in relation with error but also in relation with the time for which it has persisted. A pure I controller can bring a system to zero error, but it would be extremely slow. Since the integration of error over some time cannot change sign along with the error value, sometimes the I control prompts overshooting which has to be accounted for. The I control is of great help when the system has to work against some external physical force.
- Derivative control (D): This control doesn't consider the error, but the rate of change of error, and tries to bring this rate to zero. This accounts for the possible future values of error. Sometimes the error might not be very large at the particular moment of time but may be having a significant rate of change with respect to time, if not corrected; it will cause a large

error in the future. That is why a correction proportional to this rate is required. This is tackled by the derivative controller.

APPLICATION OF PID IN DISCRETE SYSTEMS

When applied to discrete time systems, the analog integration and differentiation can be replaced by summation and difference, hence using a PSD (Proportional-Sum-Difference) controller.

The correction term can be rewritten for discrete systems as,

$$v(t) = K_{\mathrm{p}}e(t) + K_{\mathrm{i}}\textstyle\sum_0^t e(t)\Delta t + K_{\mathrm{d}}\frac{\Delta e(t)}{\Delta t}$$

The Δt term can be defined in the program as a small time interval for one cycle of computation, like 1 millisecond. Since this term is constant in the computation process it can be included with the constant values K_p and K_d,

i.e. let $k_i = K_i\Delta t$ and $k_d = K_d / \Delta t$.

Hence, the correction term for a proportional-sum-difference (PSD) controller can be expressed as,

$$v(t) = k_{\mathrm{p}}e(t) + k_{\mathrm{i}}\textstyle\sum_0^t e(t) + k_{\mathrm{d}}\Delta e(t)$$

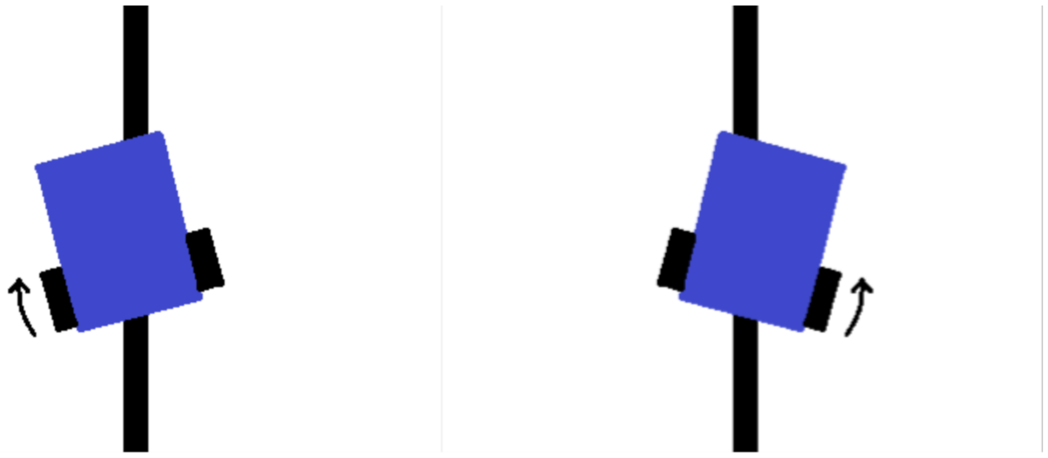
and the output *y(t)* of the controller, having base value y_o, will be expressed as,

$y(t) = y_{\mathrm{o}} + v(t)$, which will result in control action.

APPLICATION OF PID CONTROLLER IN LINE FOLLOWER BOT

The goal to be achieved in the case of a line follower is that the centre of the robot should always be on the black line in white background. Deviation of the bot's centre from the line is the error function *e(t)* in this case.

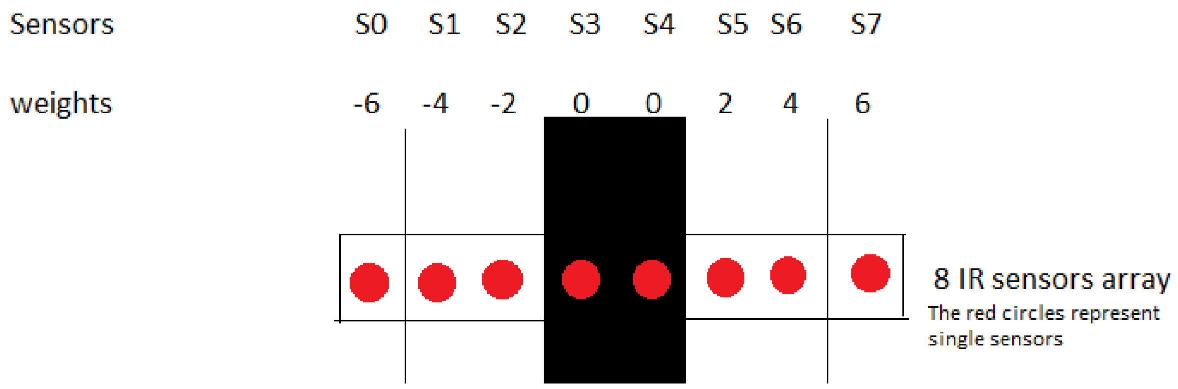
Here we will refer deviation to the left from the line as negative deviation and deviation to the right from the line as positive deviation.



negative deviation (left) positive deviation (right)

The distance between two IR sensors should be so adjusted that only two IR sensors are there on the line at a point of time. For example, if the width if the line is 3 cm, then distance between two IR sensors should be 1.5 cm.

To calculate the deviation value, we are going to mathematically model the situation by giving different weightings to all the sensors. Here we are using 8 IR sensors, so starting from left; we will give weights like -6, -4, -2, 0, 0, 2, 4, and 6. The sum of the binary value returned by each sensor multiplied by its respective weight will give us the deviation value.



So if we use an array to store sensor values, like s[i], i being the index number varying from 0 to 7, we can store binary values for each number, 0 for white and 1 for black.

So deviation value,

*Deviation = (-6)*s[0] + (-4)*s[1] + (-2)*s[2] + (0)*s[3] + (0)*s[4] + (2)*s[5] + (4)*s[6] + (6)*s[7]*

In zero error condition, when bot is exactly on the line, the array s will be,

S = {1, 1, 1, 0, 0, 1, 1, 1},

So, deviation = (-6)*1 + (-4)*1 + (-2)*1 + (0)*0 + (0)*0 + (2)*1 + (4)*1 + (6)*1
= 0 (zero error deviation is zero).

The zero error deviation i.e. zero is our set point. The process variable is the current deviation at any point of time. Their difference is our error value.

Error value = desired deviation – current deviation
= 0 – current deviation

Hence error value is positive for left deviation and negative for right deviation.

Also the position value can be calculated using the array s,

Position value K = s[0](128) + s[1]*(64) + s[2]*(32) + s[3]*(16) + s[4]*(8) + s[5]*(4) + s[6]*(2) + s[7]*(1)*

This position value is just the decimal conversion of the binary value returned by the sensors, for example, 231 for 11100111.

PSEUDO CODE FOR PID CONTROL

Here is a simple format for programming PID controller.

```
Start:
desired_deviation = 0
current_deviation = sensor_input
error = desired_deviation – current_deviation
error_sum = error_sum + error
error_difference = error – old_error
pid_output = $K_p$*(error) + $K_i$*(erro_sum) + $K_d$*(error_difference)
old_error = error
goto Start
```

For tuning the values of K_p, K_i, and K_d, the following points should be considered,

- Starting with Kp, Ki and Kd equals to zero, we shall start tuning Kp. We can start the with keeping Kp = 1, if the bot is slow and cannot turn quickly, Kp should be increased gradually or else if the bot overshoots, the value should be increased. The basic thing should be that the bot is somewhat able to follow line.

- Leaving Ki for later, we should first tune Kd. Kd should be increased in smaller values of 0.25 or 0.5. The value should be increased till the amount of wobbling is reduced.
- Now when a fairly stable line following is achieved with particular values of Kp and Kd, we should assign Ki a value between 0.5 and 1. Higher values of Ki will cause jerking and very low values won't give any perceivable results, so Ki needs to be precisely adjusted.
- The values of the constants also depend a lot on the speed of the bot and need to be adjusted accordingly.