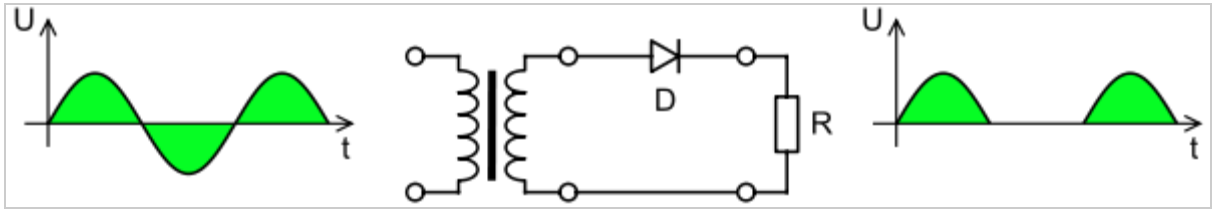
1) Need for constant voltage power source, achieved by-

a) Rectifier-

A **rectifier** is an electrical device that converts alternating current (AC), which periodically reverses direction, to direct current (DC), which flows in only one direction. The process is known as **rectification.**

b)Half-wave rectification

In half wave rectification of a single-phase supply, either the positive or negative half of the AC wave is passed, while the other half is blocked. Because only one half of the input waveform reaches the output, mean voltage is lower. Half-wave rectification requires a single diode in a single-phase supply, or three in a three-phase supply. Rectifiers yield a unidirectional but pulsating direct current; half-wave rectifiers produce far more ripple than full-wave rectifiers, and much more filtering is needed to eliminate harmonics of the AC frequency from the output.



Half-wave rectifier

The no-load output DC voltage of an ideal half wave rectifier for a sinusoidal input voltage is:

$$V_{\mathrm{rms}} = \frac{V_{\mathrm{peak}}}{2}$$

$$V_{\mathrm{dc}} = \frac{V_{\mathrm{peak}}}{\pi}$$

Where:

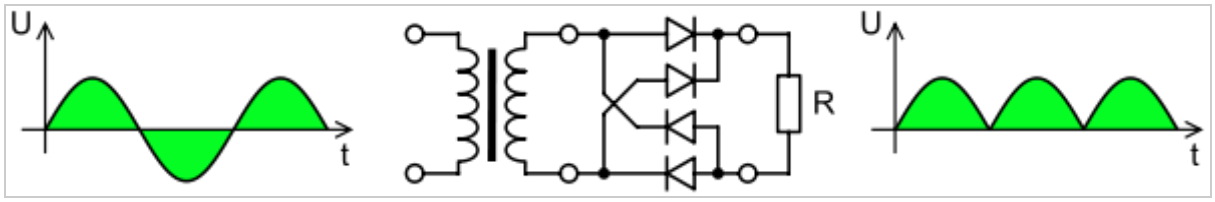
V_{dc}, V_{av} – the DC or average output voltage,

V_{peak}, the peak value of the phase input voltages,

V_{rms}, the root-mean-square value of output voltage.

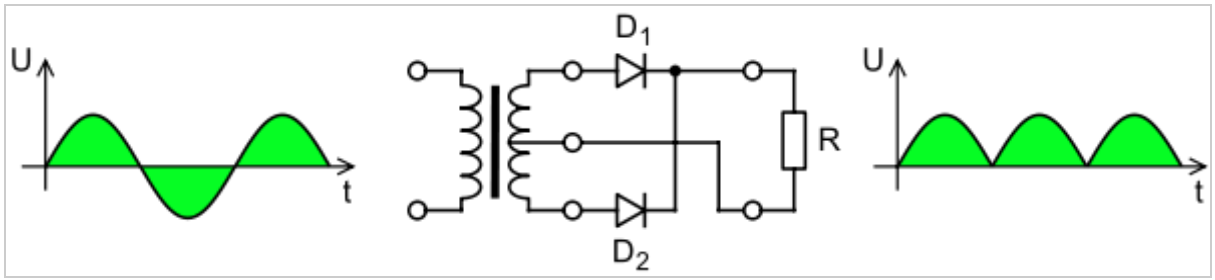
c) Full-wave rectification

A full-wave rectifier converts the whole of the input waveform to one of constant polarity (positive or negative) at its output. Full-wave rectification converts both polarities of the input waveform to pulsating DC (direct current), and yields a higher average output voltage. Two diodes and a center tapped transformer, or four diodes in a bridge configuration and any AC source (including a transformer without center tap), are needed. Single semiconductor diodes, double diodes with common cathode or common anode, and four-diode bridges, are manufactured as single components.

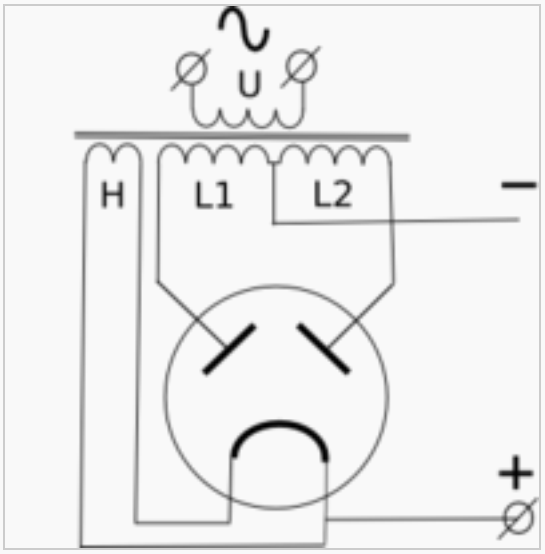


Graetz bridge rectifier: a full-wave rectifier using 4 diodes.

For single-phase AC, if the transformer is center-tapped, then two diodes back-to-back (cathode-to-cathode or anode-to-anode, depending upon output polarity required) can form a full-wave rectifier. Twice as many turns are required on the transformer secondary to obtain the same output voltage than for a bridge rectifier, but the power rating is unchanged.



Full-wave rectifier using a center tap transformer and 2 diodes.



Full-wave rectifier, with vacuum tube having two anodes.

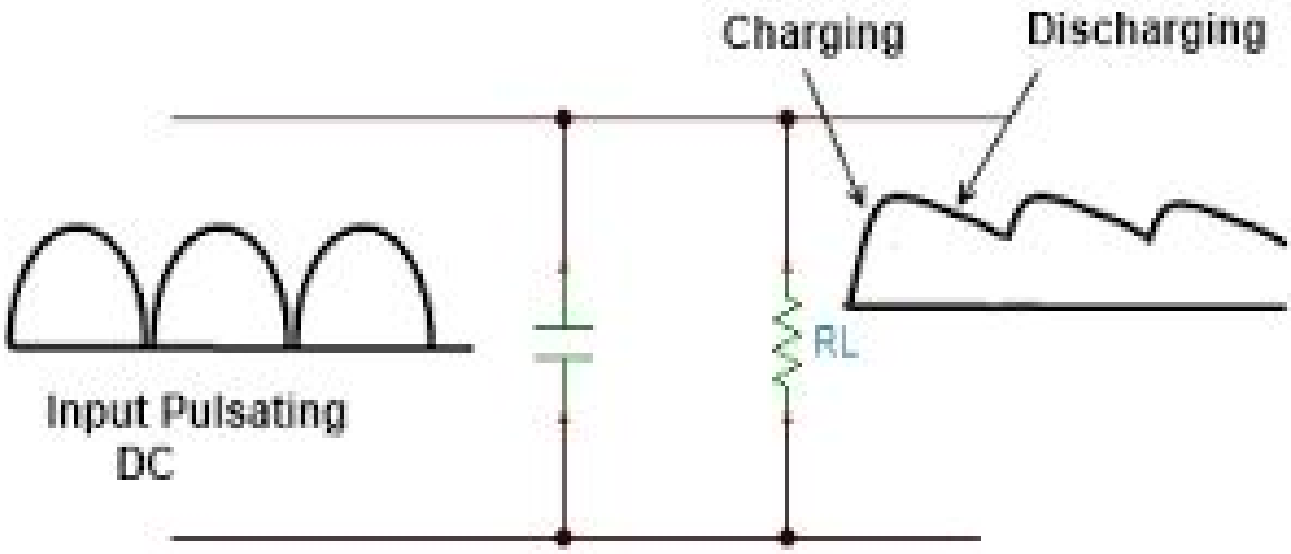
The average and root-mean-square no-load output voltages of an ideal single-phase full-wave rectifier are:

$$V_{\mathrm{dc}} = V_{\mathrm{av}} = \frac{2V_{\mathrm{peak}}}{\pi}$$

$$V_{\mathrm{rms}} = \frac{V_{\mathrm{peak}}}{\sqrt{2}}$$

Very common double-diode rectifier vacuum tubes contained a single common cathode and two anodes inside a single envelope, achieving full-wave rectification with positive output. The 5U4 and 5Y3 were popular examples of this configuration.

d) Capacitive Filter circuit-



2) Electronic Devices -

a. Integrated Circuit -

i) IC78XX -

The **78xx** (sometimes **L78xx**, **LM78xx**, **MC78xx**...) is a family of self-contained fixed linear voltage regulator integrated circuits. The 78xx family is commonly used in electronic circuits requiring a regulated power supply due to their ease-of-use and low cost. For ICs within the family, the *xx* is replaced with two digits, indicating the output voltage (for example, the 7805 has a 5-volt output, while the 7812 produces 12 volts). The 78xx line are positive voltage regulators: they produce a voltage that is positive relative to a common ground. There is a related line of **79xx** devices which are complementary negative voltage regulators. 78xx and 79xx ICs can be used in combination to provide positive and negative supply voltages in the same circuit.

78xx ICs have three terminals and are commonly found in the TO220 form factor, although smaller surface-mount and larger TO3 packages are available. These devices support an input voltage

anywhere from a few volts over the intended output voltage, up to a maximum of 35 to 40 volts depending on the make, and typically provide 1 or 1.5 amperes of current (though smaller or larger packages may have a lower or higher current rating).

ii) IC- LM293D

L293D is a typical Motor driver or Motor Driver IC which is used to drive DC on either direction. It is a 16-pin IC which can control a set of two DC motors simultaneously in any direction. It means that you can control two DC motor with a single L293D IC. Dual H-bridge Motor Driver integrated circuit (IC).The l293d can drive small and quite big motors as well.

Concept

It works on the concept of H-bridge. H-bridge is a circuit which allows the high voltage to be flown in either direction. As you know voltage should change its direction to able to rotate the motor in

clockwise or anticlockwise direction, Hence H-bridge IC are ideal for driving a DC motor.Using micro-controller

In a single l293d IC there two h-Bridge circuit inside the it which can rotate two dc motor independently. Due to its size it is very much used in robotic application for controlling DC motors.

There are two Enable pins on l293d. Pin 1 and pin 9, for being able to drive the motor, the pin 1 and 9 need to be high. For driving the motor with left H-bridge you need to enable pin 1 to high. And for right H-Bridge you need to make the pin 9 to high. If anyone of the either pin1 or pin9 goes low then the motor in the corresponding section will suspend working. It's like a switch.

Working of L293D

The 4 input pins for this l293d, pin 2,7 on the left and pin 15, 10 on the right as shown on the pin diagram. Left input pins will regulate the rotation of motor connected on the left side and right input for motor on the right hand side. The motors are rotated on the basis of the inputs provided at the input pins as LOGIC 1 or LOGIC 0.

In simple you need to provide Logic 0 or 1 across the input pins for rotating the motor.

L293D Logic Table

Lets consider a Motor connected on left side output pins (pin 3,6). For rotating the motor in clockwise direction the input pins has to be given with Logic 1 and Logic 0.

• Pin 2 = Logic 1 and Pin 7 = Logic 0 | Clockwise Direction

• Pin 2 = Logic 0 and Pin 7 = Logic 1 | Anticlockwise Direction

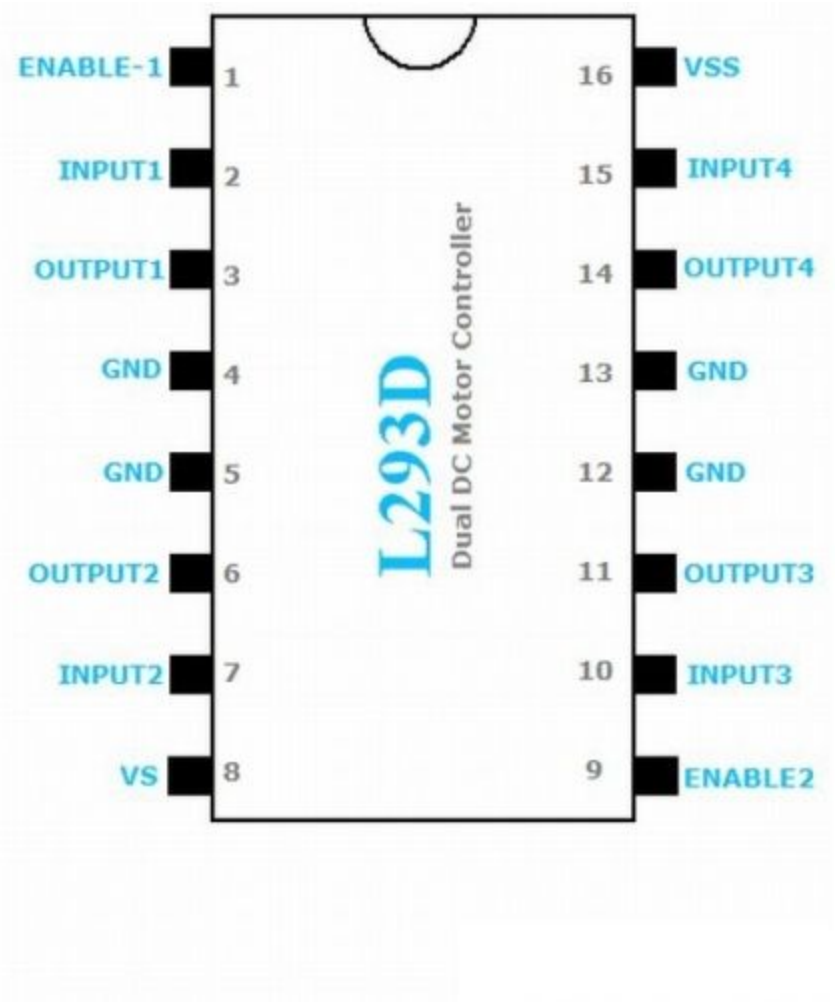
• Pin 2 = Logic 0 and Pin 7 = Logic 0 | Idle [No rotation] [Hi-Impedance state]

• Pin 2 = Logic 1 and Pin 7 = Logic 1 | Idle [No rotation]

In a very similar way the motor can also operated across input pin 15,10 for motor on the right hand side.

Voltage Specifications- The voltage (Vcc) needed to for its own working is 5V but L293d will not use that Voltage to drive DC Motors. That means you should provide that voltage(36V

maximum) and a maximum current of 600mA to drive the motors and maximum resistance is 60



ohms.

iii) OPERATIONAL - AMPLIFIER-

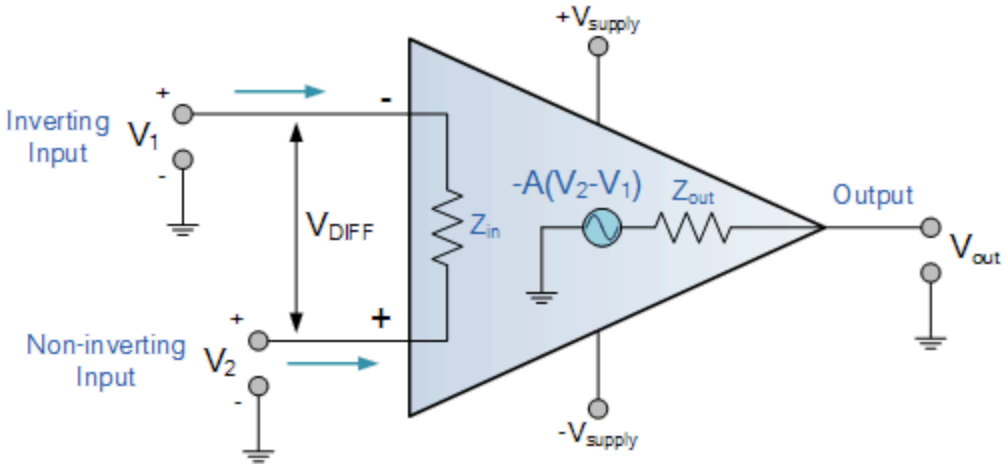
An **operational amplifier** (aka "op-amp" and "OpAmp") is a DC-coupled high-gain electronic voltage amplifier with a differential input and, usually, a single-ended output.In this configuration, an op-amp produces an output potential (relative to circuit ground) that is typically hundreds of thousands of times larger than the potential difference between its input terminals.

Operational amplifiers had their origins in analog computers, where they were used to do mathematical operations in many linear, non-linear and frequency-dependent circuits. The popularity of the op-amp as a building block in analog circuits is due to its versatility. Due to negative feedback, the characteristics of an op-amp circuit, its gain, input and output impedance, bandwidth etc. are

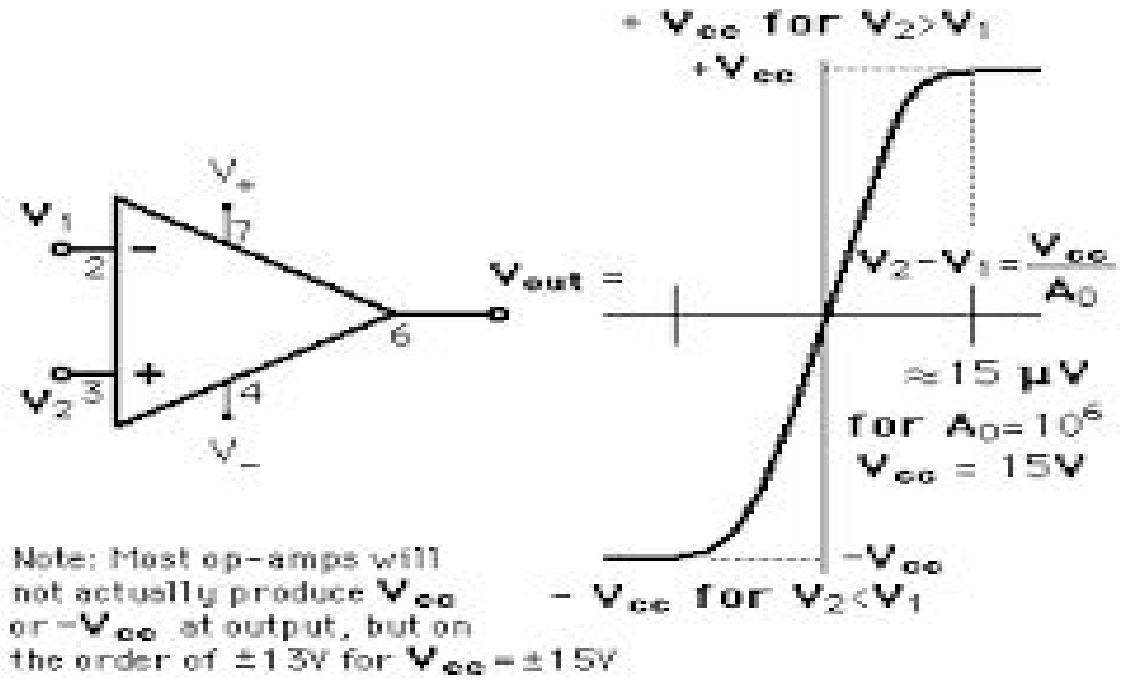
determined by external components and have little dependence on temperature coefficients or manufacturing variations in the op-amp itself.

Op-amps are among the most widely used electronic devices today, being used in a vast array of consumer, industrial, and scientific devices. Many standard IC op-amps cost only a few cents in moderate production volume; however some integrated or hybrid operational amplifiers with special performance specifications may cost over $100 US in small quantities. Op-amps may be packaged as components, or used as elements of more complex integrated circuits.

The op-amp is one type of differential amplifier. Other types of differential amplifier include the fully differential amplifier (similar to the op-amp, but with two outputs), the instrumentation amplifier (usually built from three op-amps), the isolation amplifier (similar to the instrumentation amplifier, but with tolerance to common-mode voltages that would destroy an ordinary op-amp), and negative feedback amplifier (usually built from one or more op-amps and a resistive feedback network).



iv) COMPARATOR -

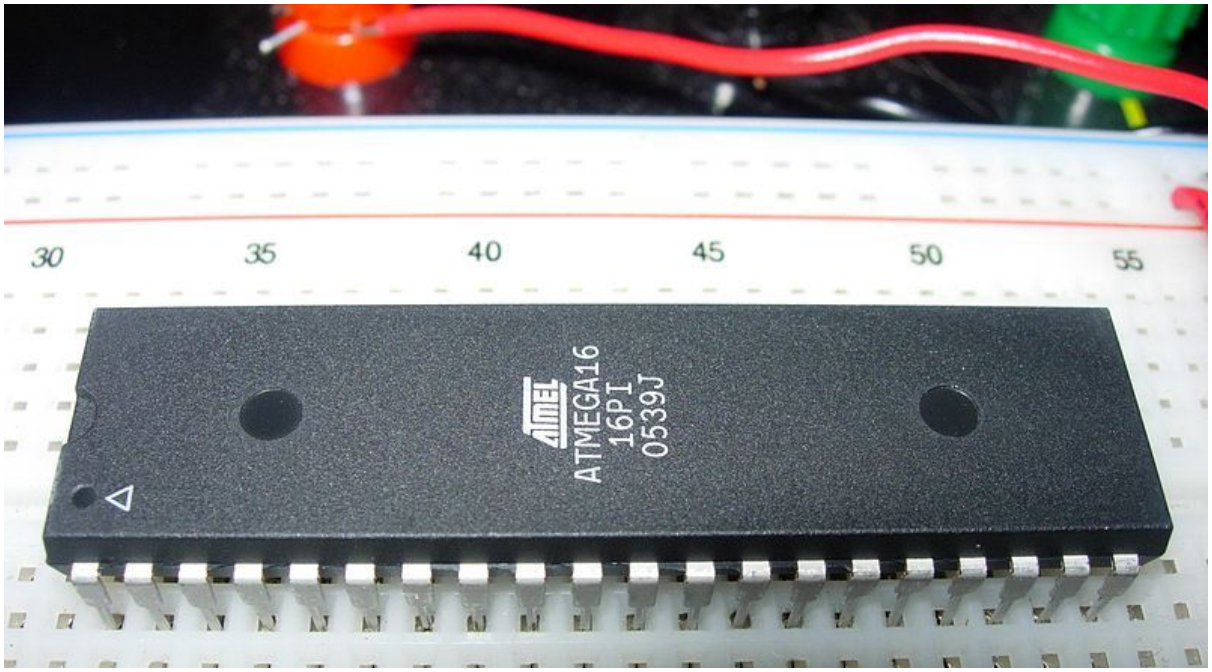


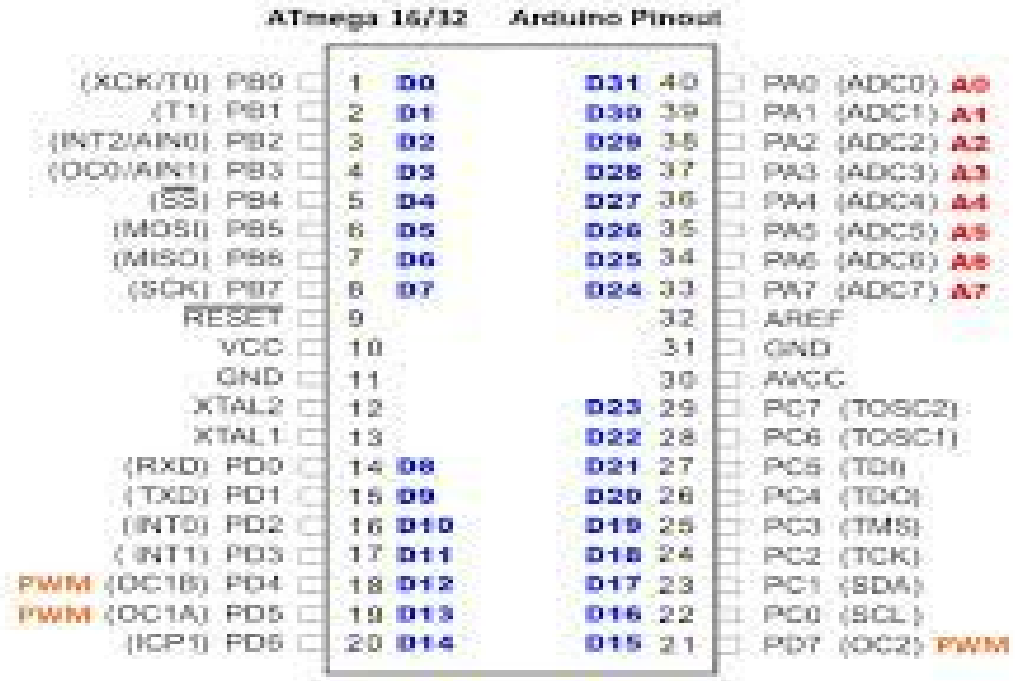
DOCUMENTATION DAY 2-

1) ATmega16-

The **AVR** is a modified Harvard architecture 8-bit RISC single-chip microcontroller, which was developed by Atmel in 1996. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM,EPROM, or EEPROM used by other microcontrollers at the time.

An Atmel ATmega16 microcontroller in a 40 pin DIP package. Mounted in a solderless breadboard.





CODES WRITTEN -

1) TO GLOW AN ARRAY OF LEDS USING ATmega16 -

```
/*
 * GLOWING ARRAY OF LEDS WITHOUT FOR
```

```
 *
 * Created: 12/2/2015 8:11:55 AM
 *  Author: ABhinav Jha
 */


#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
   DDRB= 0b11111111;
         while(1)
   {
   PORTB= (1<<PB0);
         _delay_ms(1000);
         PORTB= (1<<PB1);
         _delay_ms(1000);
         PORTB= (1<<PB2);
         _delay_ms(1000);
         PORTB= (1<<PB3);
         _delay_ms(1000);
         PORTB= (1<<PB4);
         _delay_ms(1000);
         PORTB= (1<<PB5);
         _delay_ms(1000);
         PORTB= (1<<PB6);
         _delay_ms(1000);
         PORTB= (1<<PB7);
         _delay_ms(1000);
         }
         return 0;
}
```

2) GLOWING ARRAY OF LEDS WITH FOR -

```
/*
 * GLOWING ARRAY OF LEDS WITH FOR
 *
 * Created: 12/2/2015 8:33:48 AM
 *  Author: Abhinav Jha
 */
```

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

int main(void)
{
	DDRB = 0b11111111;
	PORTB= 0b00000000;
	int n;

   for(int i=0;i>=0;i++)
	{n=i%8;
		PORTB|=(1<<n);
		_delay_ms(1000);
		PORTB= 0b00000000;
		if(i>9)
		i=0;
	}

	return 0;
}
```

DOCUMENTATION DAY 3 -

1) ADC (Analog to Digital Convertor) -

An **analog-to-digital converter** (**ADC**, **A/D**, or **A to D**) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude.

The conversion involves quantization of the input, so it necessarily introduces a small amount of error. Furthermore, instead of continuously performing the conversion, an ADC does the conversion periodically, sampling the input. The result is a sequence of digital values that have been converted from a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete-amplitude digital signal.

An ADC is defined by its bandwidth (the range of frequencies it can measure) and its signal to noise ratio (how accurately it can measure a signal relative to the noise it introduces). The actual bandwidth of an ADC is characterized primarily by its sampling rate, and to a lesser extent by how it handles errors such as aliasing. The dynamic range of an ADC is influenced by many factors, including the resolution (the number of output levels it can quantize a signal to), linearity and accuracy (how well the quantization levels match the true analog signal) and jitter (small timing errors that introduce additional noise). The dynamic range of an ADC is often summarized in terms of its effective number of bits (ENOB), the number of bits of each measure it returns that are on average not noise. An ideal ADC has an ENOB equal to its resolution. ADCs are chosen to match the bandwidth and required signal to noise ratio of the signal to be quantized. If an ADC operates at a sampling rate greater than twice the bandwidth of the signal, then perfect reconstruction is possible given an ideal ADC and neglecting quantization error. The presence of quantization error limits the dynamic range of even an ideal ADC, however, if the dynamic range of the ADC exceeds that of the input signal, its effects may be neglected resulting in an essentially perfect digital representation of the input signal.

An ADC may also provide an isolated measurement such as an electronic device that converts an input analog voltage or current to a digital number proportional to the magnitude of the voltage or current. However, some non-electronic or only partially electronic devices, such as rotary encoders, can also be considered ADCs. The digital output may use different coding schemes. Typically the digital output will be a two's complement binary number that is proportional to the input, but there are other possibilities. An encoder, for example, might output a Gray code.

2) PULSE WIDTH MODULATION-

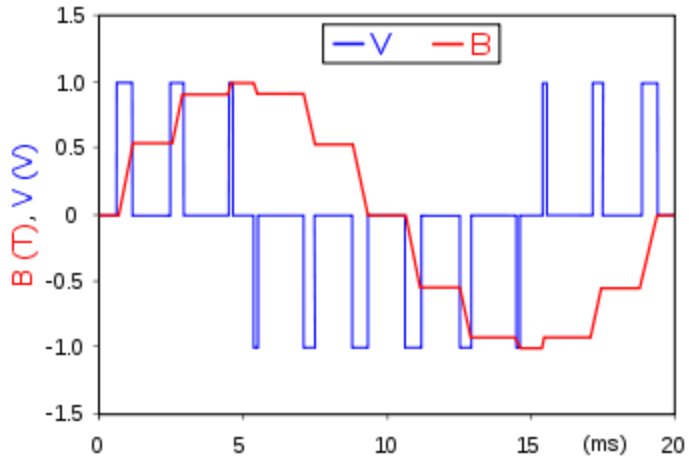
Pulse-width modulation (**PWM**), or **pulse-duration modulation** (**PDM**), is a modulation technique used to encode a message into a pulsing signal. Although this modulation technique can be used to encode information for transmission, its main use is to allow the control of the power supplied to electrical devices, especially to inertial loads such as motors. In addition, PWM is one of the two principal algorithms used in photovoltaic solar battery chargers,[1] the other being MPPT.

The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power supplied to the load.

The PWM switching frequency has to be much higher than what would affect the load (the device that uses the power), which is to say that the resultant waveform perceived by the load must be as smooth as possible. Typically switching has to be done several times a minute in an electric stove, 120 Hz in a lamp dimmer, from few kilohertz (kHz) to tens of kHz for a motor drive and well into the tens or hundreds of kHz in audio amplifiers and computer power supplies.

The term *duty cycle* describes the proportion of 'on' time to the regular interval or 'period' of time; a low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on.

The main advantage of PWM is that power loss in the switching devices is very low. When a switch is off there is practically no current, and when it is on and power is being transferred to the load, there is almost no voltage drop across the switch. Power loss, being the product of voltage and current, is thus in both cases close to zero. PWM also works well with digital controls, which, because of their on/off nature, can easily set the needed duty cycle



(An example of PWM in an idealized inductor driven by a voltage source: the voltage source (blue) is modulated as a series of pulses that results in a sine-like current/flux (red) in the inductor. The blue rectangular pulses nonetheless result in a smoother and smoother red sine wave as the **switching frequency** increases. Note that the red waveform is the (definite) integral of the blue waveform.)

3) TIMERS-

Most applications for microcontrollers have to refer to some kind of time on one way or the other. The typical AVR controllers have three timers called Timer0, Timer1, and Timer2. Newer controllers sometimes have even more timers. The timers Timer0 and Timer2 are 8-bit wide timers and Timer1 is a 16-bit timer.

4) INTERRUPTS-

Interrupts are basically events that require immediate attention by the microcontroller. When an interrupt event occurs the microcontroller pause its current task and attend to the interrupt by executing an **Interrupt Service Routine (ISR)** at the end of the ISR the microcontroller returns to the task it had pause and continue its normal operations.
In order for the microcontroller to respond to an interrupt event the interrupt feature of the microcontroller must be enabled along with the specific interrupt. This is done by setting the **Global Interrupt Enable** bit and the **Interrupt Enable** bit of the specific interrupt.

Interrupt Service Routine or Interrupt Handler

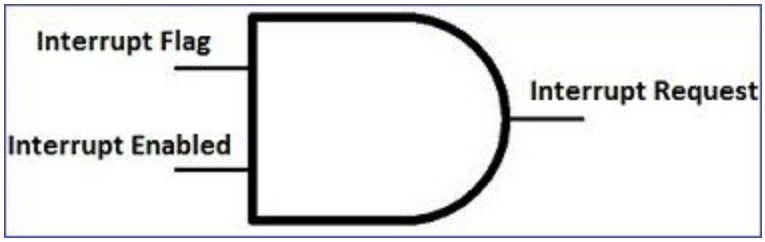
An **Interrupt Service Routine (ISR)** or **Interrupt Handler** is a piece of code that should be execute when an interrupt is triggered. Usually each enabled interrupt has its own ISR. In AVR assembly language each ISR **MUST** end with the **RETI** instruction which indicates the end of the ISR.

Interrupt Flags and Enabled bits

Each interrupt is associated with two (2) bits, an **Interrupt Flag Bit** and an **Interrupt Enabled Bit**. These bits are located in the I/O registers associated with the specific interrupt:

- The **interrupt flag** bit is set whenever the interrupt event occur, whether or not the interrupt is enabled.
- The **interrupt enabled** bit is used to enable or disable a specific interrupt. Basically is tells the microcontroller whether or not it should respond to the interrupt if it is triggered.

In summary basically both the **Interrupt Flag** and the **Interrupt Enabled** are required for an interrupt request to be generated as shown in the figure below.



Global Interrupt Enabled Bit

Apart from the enabled bits for the specific interrupts the global interrupt enable bit **MUST** be enabled for interrupts to be activated in the microcontroller.

For the AVR 8-bits microcontroller this bit is located in the **Status I/O Register (SREG)**. The Global Interrupt Enabled is bit 7, the **I** bit, in the SREG.

CODES WRITTEN -

1) TO USE ADC TO CONVERT POTENTIOMETER INPUT TO GLOW DIFFERENT LEDS-

```
/*
 * TO USE ADC TO CONVERT POTENTIOMETER INPUT TO GLOW DIFFERENT LEDS
 *
 * Created: 12/2/2015 11:29:22 AM
 *  Author: Abhinav Jha
 */


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

```
int main(void)
{
  DDRA= 0b00000000;
  DDRB=0b11111111;
  ADCSRA= ADCSRA|(1<<ADEN)|(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2);
  ADMUX=
ADMUX|(1<<ADLAR)|(0<<MUX0)|(0<<MUX1)|(0<<MUX2)|(1<<REFS0)|(0<<REFS1);
  while(1)
  {
        ADCSRA|=(1<<ADSC);
        while(ADCSRA&(1<<ADSC));
        int a= ADCH;

        if (a>=0 && a<=32)
{PORTB|=(1<<0);
        _delay_ms(1000);
        PORTB=0b00000000;
}
        else if (a>32 && a<=64)
{PORTB|=(1<<1);
        _delay_ms(1000);
        PORTB=0b00000000;
}
        else if (a>64 && a<=96)
{       PORTB|=(1<<2);
        _delay_ms(1000);
        PORTB=0b00000000;
}
        else if (a>96 && a<=128)
        {PORTB|=(1<<3);
        _delay_ms(1000);
        PORTB=0b00000000;
   }
        else if (a>128 && a<=160)
        {PORTB|=(1<<4);
        _delay_ms(1000);
        PORTB=0b00000000;
    }
        else if (a>160 && a<=192)
        {PORTB|=(1<<5);
        _delay_ms(1000);
        PORTB=0b00000000;
   }
```

```
        else if (a>192 && a<=224)
        {PORTB|=(1<<6);
        _delay_ms(1000);
        PORTB=0b00000000;
    }
        else if (a>224 && a<=255)
        {PORTB|=(1<<7);
        _delay_ms(1000);
        PORTB=0b00000000;
        }



   }

   return 0;

}
```

2) TO USE TIMERS TO CREATE A DELAY FUNCTION -

```
/*
 * TO USE TIMERS TO CREATE A DELAY FUNCTION
 *
 * Created: 12/2/2015 4:47:43 PM
 *  Author: AbhinavJha
 */

#define F_CPU 16000000UL
#include <avr/io.h>

void timer(int n)
{
        TCCR0 |= (1 << CS00)|( 1<< CS02);
        TCNT0 = 0x00;

        long int i=0;

        while(1)
        {

                if (TCNT0==255)
                {
```

```
                    i=i+1;
                    TCNT0=0;
                    }
                    if(i>=(61*n))
                    return;

          }
}

int main()
{         DDRB=0xff;

          while (1)
          {PORTB=0x00;
           timer(1);
           PORTB=0xff;
           timer(1);
          }


}
```

3)TO GENERATE PULSE WIDTH MODULATION(PWM) -

```
/*
 * PWN_GENERATION.c
 *
 * Created: 12/2/2015 7:21:28 PM
 *  Author: user
 */


#include <avr/io.h>
#include <util/delay.h>
int main(void)
{ TCCR0=0b01101101;
          DDRB=0b11111111;
   TCNT0=0;
          OCR0=0;

          while(1)
   {
     _delay_ms(1000);
            OCR0+=20;
```

```
            if (OCR0>250)
            OCR0=0;

   }
```

4) MAKING USE OF INTERRUPT TO TOGGLE OUTPUT -

```
/*
 * Interrupt.c
 *
 * Created: 12/2/2015 10:30:23 PM
 *  Author: Abhinav Jha
 */


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

ISR(INT0_vect)
{
        PORTB=0x00;
        _delay_ms(1000);
}
ISR(INT1_vect)
{
 PORTB=0x00;
_delay_ms(1000);

}

int main(void)
{   sei();
        DDRB=0xff;
    DDRD=0x00;
        PORTD=0xff;
        GICR=GICR|(1<<INT0);
        MCUCR=MCUCR||(1<<1);
    while(1)
    {

        PORTB=0xff;
```

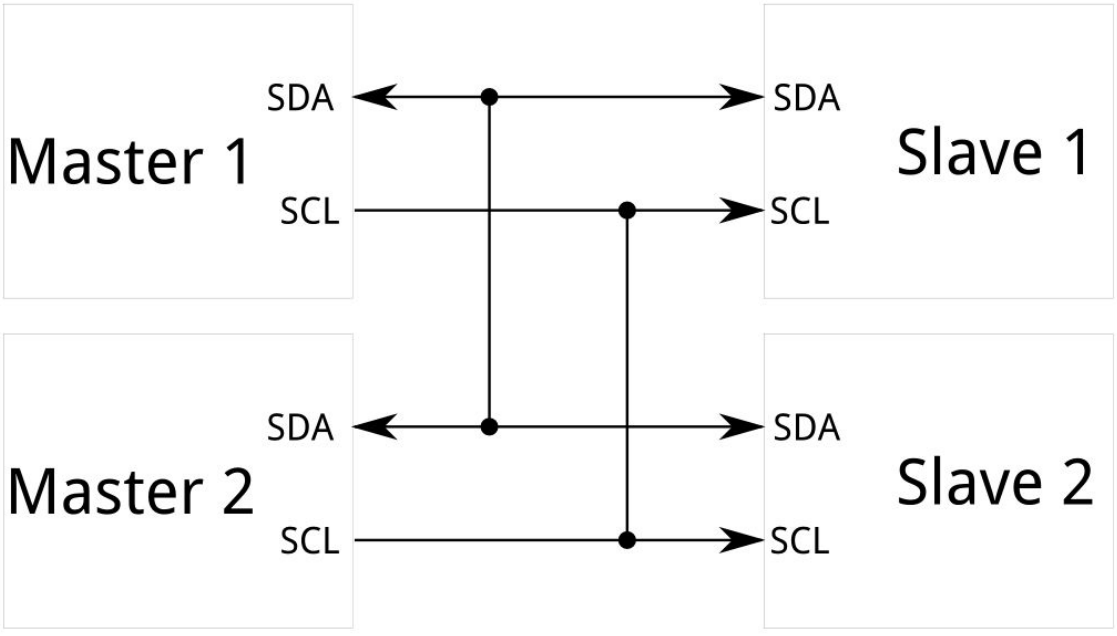
```
        _delay_ms(1000);


    }
cli();
}
```

DOCUMENTATION DAY 4 -

I2C PROTOCOL-

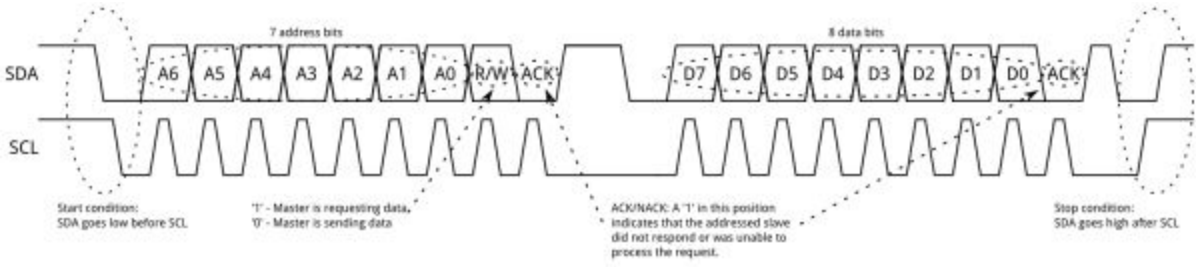
The Inter-integrated Circuit (I2C) Protocol is a protocol intended to allow multiple “slave” digital integrated circuits (“chips”) to communicate with one or more “master” chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.



I2C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 slave devices. Also, unlike SPI, I2C can support a multi-master system, allowing more than one master to communicate with all devices on the bus (although the master devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I2C devices can communicate at 100kHz or 400kHz. There is some overhead with I2C; for every 8 bits of data to be sent, one extra bit of meta data (the "ACK/NACK" bit, which we'll discuss later) must be transmitted.

The hardware required to implement I2C is more complex than SPI, but less than asynchronous serial. It can be fairly trivially implemented in software.



Messages are broken up into two types of frame: an address frame, where the master indicates the slave to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to slave or vice versa. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.

Start Condition

To initiate the address frame, the master device leaves SCL high and pulls SDA low. This puts all slave devices on notice that a transmission is about to start. If two master devices wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. It is possible to issue repeated starts, initiating a new communication sequence without relinquishing control of the bus to other masters; we'll talk about that later.

Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.

The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address). Once the first 8 bits of the frame are sent, the receiving device is given control over SDA. If the receiving

device does not pull the SDA line low before the 9th clock pulse, it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the master of the system to decide how to proceed.

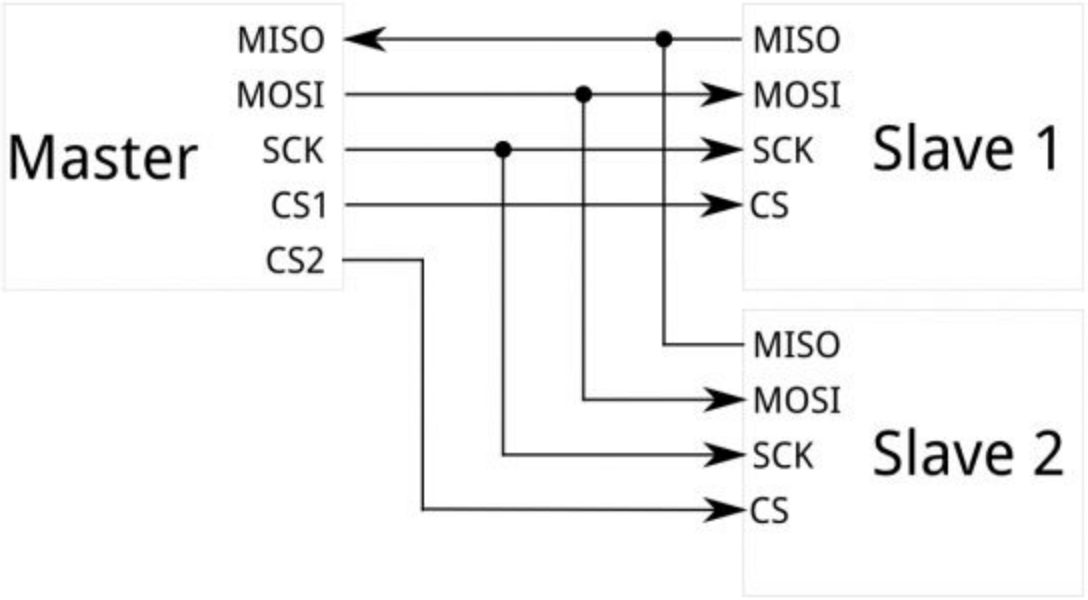
Data Frames

After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation. The number of data frames is arbitrary, and most slave devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.

Stop condition

Once all the data frames have been sent, the master will generate a stop condition. Stop conditions are defined by a 0->1 (low to high) transition on SDA *after* a 0->1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should not change when SCL is high, to avoid false stop conditions.

SPI PROTOCOL -



Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to. Many microcontrollers have built-in SPI peripherals that handle all the details of sending and receiving data, and can do so at very high speeds. The SPI protocol is also simple enough that you (yes, you!) can write your own routines to manipulate the I/O lines in the proper sequence to transfer data.

If you're using an Arduino, there are two ways you can communicate with SPI devices:

1. You can use the shiftIn() and shiftOut() commands. These are software-based commands that will work on any group of pins, but will be somewhat slow.
2. Or you can use the SPI Library, which takes advantage of the SPI hardware built into the microcontroller. This is vastly faster than the above commands, but it will only work on certain pins.

You will need to select some options when setting up your interface. These options must match those of the device you're talking to; check the device's datasheet to see what it requires.

- The interface can send data with the most-significant bit (MSB) first, or least-significant bit (LSB) first. In the Arduino SPI library, this is controlled by the setBitOrder() function.
- The slave will read the data on either the rising edge or the falling edge of the clock pulse. Additionally, the clock can be considered "idle" when it is high or low. In the Arduino SPI library, both of these options are controlled by the setDataMode() function.
- SPI can operate at extremely high speeds (millions of bytes per second), which may be too fast for some devices. To accommodate such devices, you can adjust the data rate. In the Arduino SPI library, the speed is set by the setClockDivider() function, which divides the master clock (16MHz on most Arduinos) down to a frequency between 8MHz (/2) and 125kHz (/128).
- If you're using the SPI Library, you must use the provided SCK, MOSI and MISO pins, as the hardware is hardwired to those pins. There is also a dedicated SS pin that you can use (which must, at least, be set to an output in order for the SPI hardware to function), but note that you can use any other available output pin(s) for SS to your slave device(s) as well.

TIMER1-

Methodology – Using prescaler and interrupt

Okay, so before proceeding further, let me jot down the formula first.

$$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$

Given that we have a CPU Clock Frequency of 16 MHz. At this frequency, and using a 16-bit timer (MAX = 65535), the maximum delay is 4.096 ms. It's quite low. Upon using a prescaler of 8, the timer frequency reduces to 2 MHz, thus giving a maximum delay of 32.768 ms. Now we need a delay of 2 s. Thus, 2 s ÷ 32.768 ms = 61.035 ≈ 61. This means that the timer should overflow 61 times to give a delay of approximately 2 s.

Now it's time for you to get introduced to the TIMER1 registers (ATMEGA16/32). We will discuss only those registers and bits which are required as of now. More will be discussed as and when necessary.

TCCR1B Register

The **Timer/Counter1 Control Register B**– TCCR1B Register is as follows.

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B Register

Right now, only the highlighted bits concern us. The **bit 2:0 – CS12:10** are the **Clock Select Bits** of TIMER1. Their selection is as follows.

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Clock Select Bits Description

Since we need a prescaler of 8, we choose the third option (010).

TCNT1 Register

The **Timer/Counter1** – TCNT1 Register is as follows. It is 16 bits wide since the TIMER1 is a 16-bit register. **TCNT1H** represents the HIGH byte whereas**TCNT1L** represents the LOW byte. The timer/counter value is stored in these bytes.

Bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H
	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCNT1 Register

TIMSK Register

The **Timer/Counter Interrupt Mask Register** – TIMSK Register is as follows.

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TIMSK Register

As we have discussed earlier, this is a *common* register for all the timers. The bits associated with other timers are greyed out. **Bits 5:2** correspond to TIMER1. Right now, we are interested in the yellow bit only. Other bits are related to CTC mode which we will discuss later. **Bit 2 – TOIE1 – Timer/Counter1 Overflow Interrupt Enable** bit enables the overflow interrupt of TIMER1. We enable the overflow interrupt as we are making the timer overflow 61 times (refer to the methodology section above).

TIFR Register

The **Timer/Counter Interrupt Flag Register** – TIFR is as follows.

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TIFR Register

Once again, just like TIMSK, TIFR is also a register *common* to all the timers. The greyed out bits correspond to different timers. Only **Bits 5:2** are related to TIMER1. Of these, we are interested in **Bit 2 – TOV1 – Timer/Counter1 Overflow Flag**. This bit is set to '1' whenever the timer overflows. It is cleared (to zero) automatically as soon as the corresponding Interrupt Service Routine (ISR) is executed. Alternatively, if there is no ISR to execute, we can clear it by writing '1' to it.

CODES WRITTEN -

1)TO ROTATE SERVO MOTOR BY 180 DEGREES USING PWM -

```
/*
 * SERVO_MOTOR.c
 *
 * Created: 12/3/2015 2:48:43 PM
 *  Author: Abhinav Jha
 */
```

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{ DDRD=0xff;
        PORTD=0xff;
        TCCR1A= TCCR1A|(1<<WGM11)|(0<<WGM10)|(1<<COM1A1)|(0<<COM1A0);
        TCCR1B= TCCR1B|(0<<CS12)|(1<<CS11)|(1<<CS10)|(1<<WGM13)|(1<<WGM12);
        ICR1=5000;
        OCR1A= 137.5;
        while(1)
        { float n;
                n=(600-137.5)/180;
        OCR1A= n+OCR1A;
        _delay_ms(450);
        if(OCR1A>=600)
        OCR1A=137.5;

        }


        return;


}
```

DOCUMENTATION DAY - 5 -

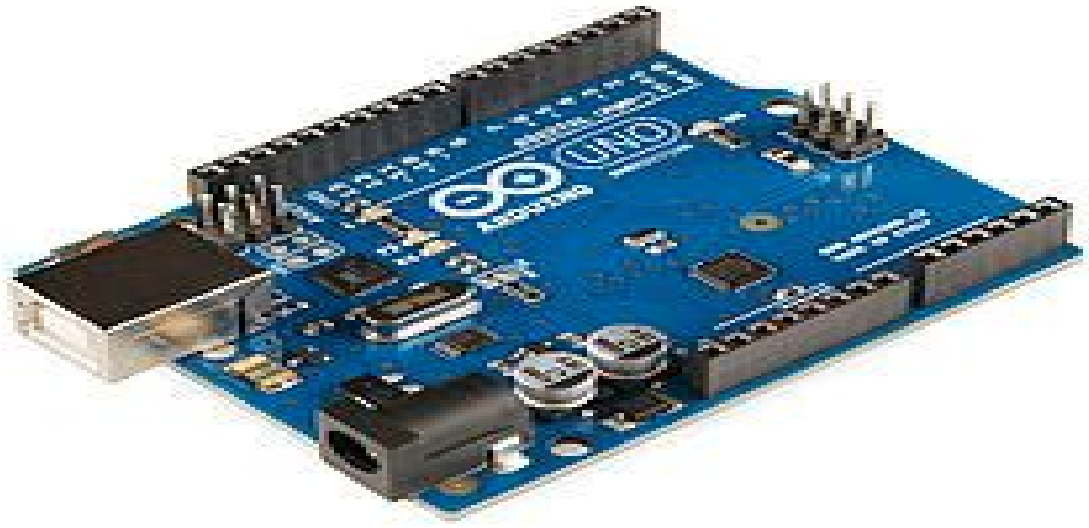
ARDUINO -

Arduino is an open-source computer hardware and software company, project and user community that designs and manufactures microcontroller-based kits for building digital devices and interactive objects that can sense and control objects in the physical world.

The project is based on microcontroller board designs, manufactured by several vendors, using various microcontrollers. These systems provide sets of digital and analog I/O pins that can be interfaced to various expansion boards ("shields") and other circuits. The boards feature serial communications interfaces, including USB on some models, for loading programs from personal computers. For programming the microcontrollers, the Arduino project provides an integrated development environment (IDE) based on the Processing project, which includes support for the C and C++ programming languages.

The first Arduino was introduced in 2005, aiming to provide an inexpensive and easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. Common examples of such devices intended for beginner hobbyists include simple robots, thermostats, and motion detectors.

Arduino boards are available commercially in preassembled form, or as do-it-yourself kits. The hardware design specifications are openly available, allowing the Arduino boards to be manufactured by anyone. Adafruit Industries estimated in mid-2011 that over 300,000 official Arduinos had been commercially produced, and in 2013 that 700,000 official boards were in users' hands.



CODES WRITTEN -

1) LIGHT DETECTOR SENSOR -

```
int sensePin= 0;
int ledPin=13;

void setup()
{
  pinMode(ledPin,OUTPUT);

}

void loop()
{
  int val= analogRead(sensePin);
  val= constrain(val,750,900);
  int ledlevel= map(val,750,900,255,0);
  analogWrite(ledPin,ledlevel);
}
```

2) RUNNING SERVO MOTOR WITH PWM -

```
#include <Servo.h>
 Servo myServo;

 int servoPin=9;
 int distpin=0;

 void setup()
 {
   myServo.attach(servoPin);

 }

 void loop()
 {
   int dist = analogRead(distpin);
   int pos= map(dist,0,1023,0,180);
   myServo.write(pos);
```

```
 }
```

3) PWM CONTROLLED LED BRIGHTNESS -

```
int switchPin= 8;
int ledPin=11;
boolean lastButton= LOW;
boolean currentButton= LOW;
int ledlevel= 0;

void setup()
{
  pinMode(switchPin, INPUT);
  pinMode(ledPin,OUTPUT);

}

boolean debounce(boolean last)
{
  boolean current= digitalRead(switchPin);
  if( last != current)
  {
    delay(5);
    current=digitalRead(switchPin);
  }
  return current;
}

void loop()
{
  currentButton= debounce(lastButton);
  if (lastButton== LOW && currentButton == HIGH)
   ledlevel= ledlevel+51;

  lastButton= currentButton;
  if (ledlevel>255) ledlevel=0;

  analogWrite(ledPin,ledlevel);

}
```

4) RUNNING MOTORS TAKING INPUT FROM SENSORS -

```
#include<math.h>
#include<Wire.h>
int outm1=1;
```

```
int outm2=2;
int outm3=3;
int outm4=4;
const int MPU=0x68;

int ena1=5;
int ena2=6;



void setup()
{

  pinMode(ena1,OUTPUT);
  pinMode(ena2,OUTPUT);
  pinMode (outm1,OUTPUT);
    pinMode (outm2,OUTPUT);
     pinMode (outm3,OUTPUT);
      pinMode (outm4,OUTPUT);


  //pinMode (11,OUTPUT);
  //digitalWrite(11,HIGH);
  Serial.begin(38400);           //data send by IMU is too fast
  Wire.begin();   //start with I2C transmission
  Wire.beginTransmission(MPU);  //transmission with this address
  Wire.write(0x6B);  //first specifies power management address of MCU to be given command
  Wire.write(0);   //awakes MCU by sending 0 to above register address
 Wire.endTransmission(true);
 }
void loop(){
  Wire.beginTransmission(MPU); //starting the communication again
  Wire.write(0x3B);  //start with this register address (its the first data register
  Wire.endTransmission(false);  //continue to read data
  Wire.requestFrom(MPU,14,true);  //request the slave to send the 14 byte data
  int acc_x=Wire.read()<<8|Wire.read();  //acc_x is 16 bit data .the data is automatically read
sequently from 0x3B
  int acc_y=Wire.read()<<8|Wire.read();  //all the data is sequently stored in registers in
IMU....hence we can read it sequently only by specifying the starting address .
  int acc_z=Wire.read()<<8|Wire.read();
  int tmp=Wire.read()<<8|Wire.read();  //each quantity has 16 bit data..however the wire.read
reads only 8 bit at a time.first H register and then L register
  int gyr_x=Wire.read()<<8|Wire.read();
```

```
 int gyr_y=Wire.read()<<8|Wire.read();
 int gyr_z=Wire.read()<<8|Wire.read();
 Wire.endTransmission();  //end the transmission
 float ACCY=atan2((double)acc_x,(double)acc_z)*180/PI;
 if(ACCY<0)
 float ACCY1=ACCY;
 float pos_duty_cycle = map(ACCY,15,90,155,255);
 float neg_duty_cycle = map(ACCY1,-15,-90,155,255);
 if (pos_duty_cycle>150 )
{ digitalWrite(outm1,HIGH);
 digitalWrite(outm3,HIGH);
 digitalWrite(outm2,0);
 digitalWrite(outm4,0);
 analogWrite(ena1,pos_duty_cycle);
  analogWrite(ena2,pos_duty_cycle);
}

 if (neg_duty_cycle>150)
{ digitalWrite(outm1,0);
 digitalWrite(outm3,0);
 digitalWrite(outm2,HIGH);
 digitalWrite(outm4,HIGH);
  analogWrite(ena1,neg_duty_cycle);
   analogWrite(ena2,neg_duty_cycle);
}

}
```

DOCUMENTATION - DAY -6 -

CODE TO TAKE INPUT FROM MAGNETOMETER -

```
#include <Wire.h>
/*****************************************************************************************************/
int j,x,y,z;
double angle;
#define HMC5883_WriteAddress 0x1E     //  i.e 0x3C >> 1
```

```
#define HMC5883_ModeRegisterAddress 0x02
#define HMC5883_ContinuousModeCommand 0x00
#define HMC5883_DataOutputXMSBAddress  0x03

int regb=0x01;
int regbdata=0x40;
int outputData[6];
/*****************************************************************************************************/

void setup_HMC()
{
   Wire.beginTransmission(HMC5883_WriteAddress);
   Wire.write(regb);
   Wire.write(regbdata);
   Wire.endTransmission();

}

/*****************************************************************************/
float HMC_data()
{
   Wire.beginTransmission(HMC5883_WriteAddress); //Initiate a transmission with HMC5883
(Write address).
   Wire.write(HMC5883_ModeRegisterAddress);       //Place the Mode Register Address in
send-buffer.
   Wire.write(HMC5883_ContinuousModeCommand);     //Place the command for Continuous
operation Mode in send-buffer.
   Wire.endTransmission();                        //Send the send-buffer to HMC5883 and end the I2C
transmission.
   delay(50);

   Wire.beginTransmission(HMC5883_WriteAddress);  //Initiate a transmission with HMC5883
(Write address).
   Wire.requestFrom(HMC5883_WriteAddress,6);      //Request 6 bytes of data from the address
specified.
   delay(50);
                                       //Read the value of magnetic components X,Y and Z
   if(6 <= Wire.available()) // If the number of bytes available for reading be <=6.
   {
     for(j=0;j<6;j++)
       {
            outputData[j]=Wire.read();  //Store the data in outputData buffer
       }
```

```
  }

    x=outputData[0] << 8 | outputData[1]; //Combine MSB and LSB of X
    z=outputData[2] << 8 | outputData[3]; //Combine MSB and LSB of Z
    y=outputData[4] << 8 | outputData[5]; //Combine MSB and LSB of Y
    angle= atan2((double)y,(double)x) * (180 / 3.14159265) + 180; // angle in degrees

    Serial.print("|| Deg:");
    Serial.println(angle,2);
    return angle;
}

/*********************************************************************/
void setup(void)
{
 Serial.begin(9600);
 setup_HMC();

}
/*********************************************************************/

void loop(void)
{


 HMC_data();
 delay(100);

}

}
```

PID CONTROL -

A **proportional–integral–derivative controller** (**PID controller**) is a control loop feedback mechanism (controller) commonly used in industrial control systems. A PID controller continuously calculates an *error value* as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the error over time by adjustment of a *control variable*,

such as the position of a control valve, a damper, or the power supplied to a heating element, to a new value determined by a weighted sum:

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

where K_p, K_i, and K_d, all non-negative, denote the coefficients for the proportional, integral, and derivative terms, respectively (sometimes denoted *P, I,* and *D*). In this model,

- *P* accounts for present values of the error (e.g. if the error is large and positive, the control variable will be large and negative),
- *I* accounts for past values of the error (e.g. if the output is not sufficient to reduce the size of the error, the control variable will accumulate over time, causing the controller to apply a stronger action), and
- *D* accounts for possible future values of the error, based on its current rate of change.

As a PID controller relies only on the measured process variable, not on knowledge of the underlying process, it is broadly applicable. By tuning the three parameters of the model, a PID controller can deal with specific process requirements. The response of the controller can be described in terms of its responsiveness to an error, the degree to which the system overshoots a setpoint, and the degree of any system oscillation. The use of the PID algorithm does not guarantee optimal control of the system or even itsstability.

Some applications may require using only one or two terms to provide the appropriate system control. This is achieved by setting the other parameters to zero. A PID controller will be called a PI, PD, P or I controller in the absence of the respective control actions. PI controllers are fairly common, since derivative action is sensitive to measurement noise, whereas the absence of an integral term may prevent the system from reaching its target value.

For discrete time systems, the term PSD, for **proportional-summation-difference**, is often used.

FUZZY LOGIC -

Fuzzy logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1. By contrast, in Boolean logic, the truth values of variables may only be 0

or 1. Fuzzy logic has been extended to handle the concept of partial truth, where the truth value may range between completely true and completely false.Furthermore, when linguistic variables are used, these degrees may be managed by specific functions.

The term *fuzzy logic* was introduced with the 1965 proposal of fuzzy set theory by Lotfi A. Zadeh. Fuzzy logic has been applied to many fields, from control theory to artificial intelligence. Fuzzy logic had however been studied since the 1920s, as infinite-valued logic—notably by Łukasiewicz and Tarski.

DOCUMENTATION DAY -7 -

THE ROBOT -

```
#include <Wire.h>
#include<SoftwareSerial.h>
#define M_PI 3.14
#define EN1 9
#define IN1 6
#define IN2 7
#define EN2 8
#define IN3 5
#define IN4 4
SoftwareSerial BT(10, 11);
float head_target;
float head_init;
float walk_dist = 0;
/*****************************************************************************************/
#include <Adafruit_Sensor.h>
#include <Adafruit_HMC5883_U.h>

/* Assign a unique ID to this sensor at the same time */
Adafruit_HMC5883_Unified mag = Adafruit_HMC5883_Unified(12345);
float heading; float headingsum = 0;
float headingDegrees;
float declinationangle;


/*****************************************************************************************/
int j, x, y, z;
```

```
double angle;
#define HMC5883_WriteAddress 0x1E     //  i.e 0x3C >> 1
#define HMC5883_ModeRegisterAddress 0x02
#define HMC5883_ContinuousModeCommand 0x00
#define HMC5883_DataOutputXMSBAddress  0x03

int regb = 0x01;
int regbdata = 0x40;
int outputData[6];
/*****************************************************************************************************/

/*****************************************************************************************************/
void setup_HMC()
{
  Wire.beginTransmission(HMC5883_WriteAddress);
  Wire.write(regb);
  Wire.write(regbdata);
  Wire.endTransmission();
  //  sensor_t sensor;
  //  mag.getSensor(&sensor);
  //  Serial.println("------------------------------------");
  //  Serial.print  ("Sensor:       "); Serial.println(sensor.name);
  //  Serial.print  ("Driver Ver:   "); Serial.println(sensor.version);
  //  Serial.print  ("Unique ID:    "); Serial.println(sensor.sensor_id);
  //  Serial.print  ("Max Value:    "); Serial.print(sensor.max_value); Serial.println(" uT");
  //  Serial.print  ("Min Value:    "); Serial.print(sensor.min_value); Serial.println(" uT");
  //  Serial.print  ("Resolution:   "); Serial.print(sensor.resolution); Serial.println(" uT");
  //  Serial.println("------------------------------------");
  // Serial.println("");


}
/*****************************************************************************************************/
float HMC_data()
{
  Wire.beginTransmission(HMC5883_WriteAddress); //Initiate a transmission with HMC5883
(Write address).
  Wire.write(HMC5883_ModeRegisterAddress);       //Place the Mode Register Address in
send-buffer.
  Wire.write(HMC5883_ContinuousModeCommand);     //Place the command for Continuous
operation Mode in send-buffer.
  Wire.endTransmission();                        //Send the send-buffer to HMC5883 and end the I2C
transmission.
```

```
  delay(10);

  Wire.beginTransmission(HMC5883_WriteAddress);  //Initiate a transmission with HMC5883
(Write address).
  Wire.requestFrom(HMC5883_WriteAddress, 6);     //Request 6 bytes of data from the address
specified.
  delay(10);
  //Read the value of magnetic components X,Y and Z
  if (6 <= Wire.available()) // If the number of bytes available for reading be <=6.
  {
    for (j = 0; j < 6; j++)
    {
      outputData[j] = Wire.read(); //Store the data in outputData buffer
    }
  }

  x = outputData[0] << 8 | outputData[1]; //Combine MSB and LSB of X
  z = outputData[2] << 8 | outputData[3]; //Combine MSB and LSB of Z
  y = outputData[4] << 8 | outputData[5]; //Combine MSB and LSB of Y
  angle = atan2((double)y, (double)x) * (180 / 3.14159265) + 180; // angle in degrees

  Serial.print("| Deg:");
  Serial.println(angle, 2);
  return angle;
}

float HMC_Idata()
{
  sensors_event_t event;
  mag.getEvent(&event);

  heading = atan2(event.magnetic.y, event.magnetic.x);
  heading = heading * 180 / M_PI + 180;
  Serial.print("Bot_deg"); Serial.println(heading);
  return heading;
}
void parse_string()
{
  char a[40], l[10], t[10];
  int i, j, k;
  i = j = k = 0;

  while (1)
```

```
  {

    a[i++] = BT.read();
    while (!BT.available());
    //delayMicroseconds(250);
    if (a[i - 1] == '\n')
    {
      break;
    }
  }

  a[i] = '\0';
 // BT.flush();
  // Serial.print("A : ");
  // Serial.println(a);

  for (j = 0; a[j + 2] != 'T'; j++)
  {
    l[j] = a[j + 2];
  }
  l[j] = '\0';
  for (k = 0; a[j + 4] != '\0'; j++, k++)
  {
    t[k] = a[j + 4];
  }
  t[k] = '\0';

  walk_dist = atof(l);
  head_target = atof(t);
  Serial.print(" WALK_DIST =");
  //Serial.print(l);
  Serial.print(walk_dist);
  Serial.print(" HEAD_TARGET = ");
  //Serial.println(t);
  Serial.println(head_target);
}
void turn_zero_radius()
{
  while (fabs(head_target - head_init) >= 12)
  {
    head_init = HMC_data();
    if (head_target - head_init <= 0)   //right
    {
```

```
      digitalWrite(IN1, HIGH);
      digitalWrite(IN2, LOW);
      digitalWrite(IN3, LOW);
      digitalWrite(IN4, HIGH);
      analogWrite(EN1, 140);
      analogWrite(EN2, 140);
    }
    else if (head_target - head_init >= 0)  //left
    {
      digitalWrite(IN1, LOW);
      digitalWrite(IN2, HIGH);
      digitalWrite(IN3, HIGH);
      digitalWrite(IN4, LOW);
      analogWrite(EN1, 140);
      analogWrite(EN2, 140);
    }
    Serial.print(head_target);
    Serial.print("   |  ");
    Serial.println("Zero_radius");
  }

}
void turn_walk()
{
  while (fabs(head_target - head_init) >= 12)
  {
    head_init = HMC_data();
    if (head_target - head_init <= 0)   //right
    {
      digitalWrite(IN1, HIGH);
      digitalWrite(IN2, LOW);
      digitalWrite(IN3, HIGH);
      digitalWrite(IN4, HIGH);
      analogWrite(EN1, 140);
      analogWrite(EN2, 140);
    }
    else if (head_target - head_init >= 0)  //left
    {
      digitalWrite(IN1, HIGH);
      digitalWrite(IN2, HIGH);
      digitalWrite(IN3, HIGH);
      digitalWrite(IN4, LOW);
      analogWrite(EN1, 140);
```

```
      analogWrite(EN2, 140);
    }
  }
}
void go_forward()
{
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  analogWrite(EN1, 140);
  analogWrite(EN2, 140);
  Serial.println("Go_forward");
  //delay(25);

  //stall();
}
void stall()
{
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, HIGH);
  analogWrite(EN1, 175);
  analogWrite(EN2, 175);
  Serial.println("stall");
}
/*-----------------------------------------------------------------------------------------------------------------------
---------------------------------------*/
void setup(void)
{
  Serial.begin(9600);
  BT.begin(9600);

  /* Initialise the sensor */

  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  digitalWrite(IN1, HIGH);
```

```
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, HIGH);
  analogWrite(EN1, 0);
  analogWrite(EN2, 0);
  Serial.print("here");
//  if (!mag.begin())
//  {
//    /* There was a problem detecting the HMC5883 ... check your connections */
//    Serial.println("Ooops, no HMC5883 detected ... Check your wiring!");
//
//  }


  setup_HMC();
  /* Display some basic information on this sensor */
}
/*-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------*/
void loop(void)
{

  while (BT.available() > 0)
  {
    head_init = HMC_data();

    parse_string();

    if (fabs(head_target - head_init) >= 12 && walk_dist == 0)
    {
      turn_zero_radius();
    }
//    else if (fabs(head_target - head_init) >= 2 && walk_dist == 1)
//    {
//      turn_walk();
//    }
    else if (fabs(head_target - head_init) <= 12 && walk_dist == 1)
    {
      go_forward();
    }
    else if (fabs(head_target - head_init) <= 12 && walk_dist == 0)
    {
      stall();
```

```
    }
    else
    {
    }
    //BT.flush();
    // delay(50);
  }
}
```