DAY 1

The first day of the workshop dealt with " The Theory Of Everything". 4 major topics were discussed.

1. Physics

The principle of the Self Balancing Bot, i.e. Inverted Pendulum. The Vertical position is the position of Unstable equilibrium. We intend to use accelerometer and gyroscope to interface and measure the angle made by the bot with the vertical. Gyroscope to be used as it gives a precise reading while accelerometer gives an acurate but less precise reading. So a complementary filter has to be used to get the actual value of "theta" at all times.

Complementary Filter :

$$angle = 0.98 * (angle + gyrData * dt) + 0.02 * (accData)$$

2. Bitwise Operators

The discussed Bitwise Operators were "^","|","&","~",">>","<<"

"^" - Bitwise XOR Operator .

Truth Table :

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

"|" - Bitwise OR Operator

Truth Table:

A	B	X = A+B
0	0	0
0	1	1
1	0	1
1	1	1

“&” - Bitwise AND Operator
Truth Table:

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

“~” - Bitwise NOT Operator
Truth Table:

Input	Output
0	1
1	0

“<<”, “>>” - Left Shift/Right Shift Operators
Operators which can shift the whole binary thread left or right appending or proceeding it by zeroes by the given parameters.

3. Basic Coding
   Constants, Variables, Loops & Condition Statements were discussed.

4. AVR Coding
   Basic Etiquettes of AVR coding were discussed. So were common common practices. Then Registers “PORTX”, “DDRX”,”PINX” were discussed.

“DDRX” - Stores the pinMode of each pin in the port ‘X’, 1 for output and 0 for input.
“PORTX” - Write only Register, 1 if output is high else 0, can be set for pins in output mode only.

“PINX” - Read Only Register, 1 if input is high, 0 if low, can be read for pins in input mode only in port “X”.

5. Setting, Clearing, Checking, Toggling a bit
   Setting : a|=(1<<x)
   Clearing: a&=~(1<<x)
   Checking: a&=(1<<x)
   Toggling: a^=(1<<x)
   If ‘X’ is the position of the bit to be toggled.

Programs for basic I/O functions on ATmega16 (LED)

1. Switching LED On:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
int main(void)
{
	DDRB=setit;
	PORTB=setit;
	while(1)
	 {
		//TODO:: Please write your Application code
	}
}
```

2. Blinking LED’s

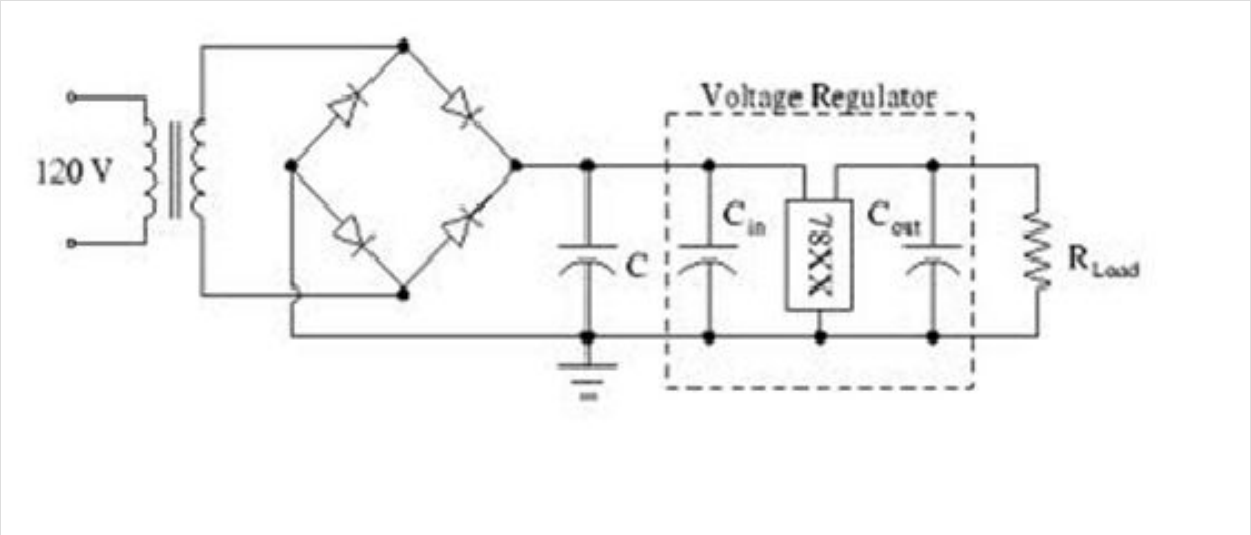
```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
int main(void)
{
	DDRB=setit;
	PORTB=setit;
	while(1)
	 {
		 //TODO:: Please write your Application code
			_delay_ms(500);
			PORTB=(~PORTB);
	}
}
```

3. Pattern LED’s

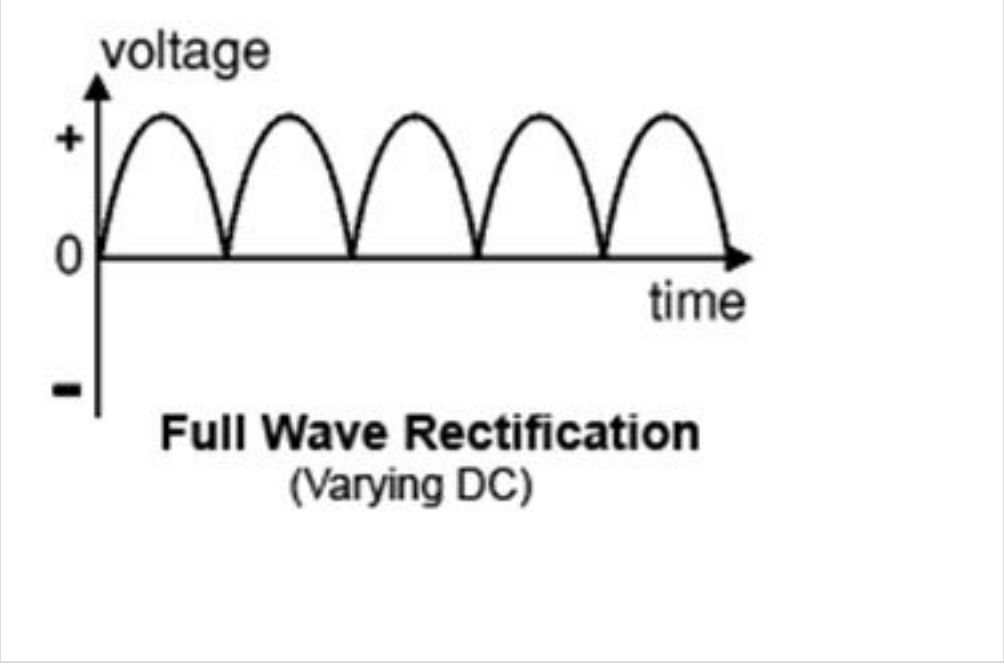
```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
#define setit1 0
int main(void)
{
	DDRB=setit;
	PORTB=setit1;
	for (int i=0; i<8; i++)
	{
		for (int j=i; j<8; j++)
		{
			PORTB|=(1<<j);
			_delay_ms(200);
			PORTB^=(1<<j);
		}
		for (int k=7; k>=i; k--)
		{
			PORTB|=(1<<k);
			_delay_ms(200);
			PORTB^=(1<<k);
		}
		PORTB|=(1<<i);
		_delay_ms(200);
	}
}
```

DAY 2

Rectifier Circuit :
They convert A.C. voltage into regulated D.C voltage. A rectifier uses the polar property of a diode i.e. a diode allows only unidirectional current to pass through it which happens when the P- terminal is at higher voltage then the N- terminal of the diode. This configuration is known as forward bias of the diode. The following shows a basic circuit for the rectifier circuit.



The diodes at the right top and left bottom are in forward bias in the positive half cycle and the diodes at the left top and the bottom right are in forward bias in the negative half cycle of the voltage. Hence current always flows in one direction in the circuit. The first capacitor is known as filter which converts the varying signals (as shown below) to ripple voltage.



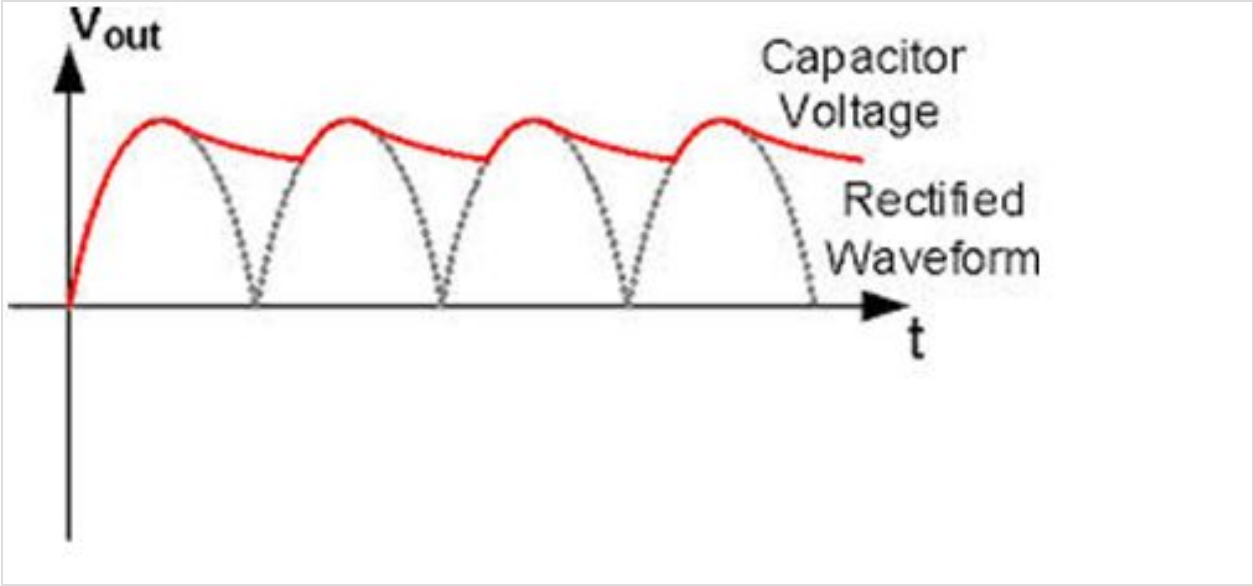
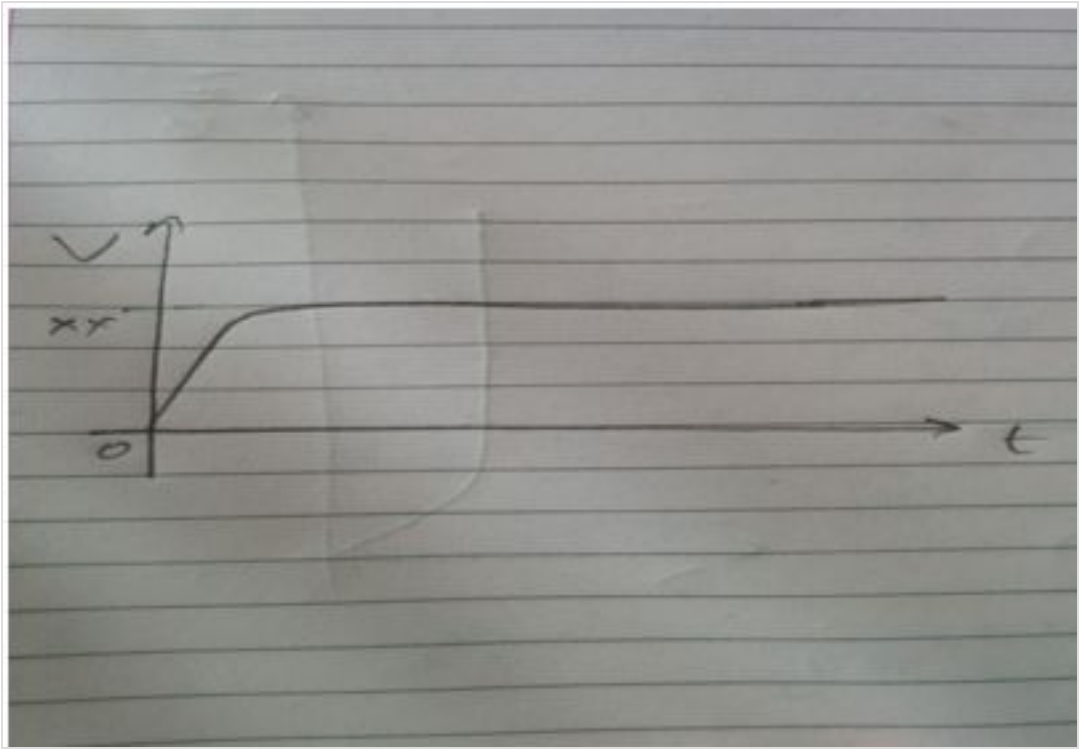


Fig 1 shows voltage after only rectifier and fig 2 shows voltage after filtering the voltage using a capacitor. The filtering works on the property of capacitor to store charge when voltage across terminals is increasing and give it away when the reverse happens i.e. the voltage across its terminals decreases.

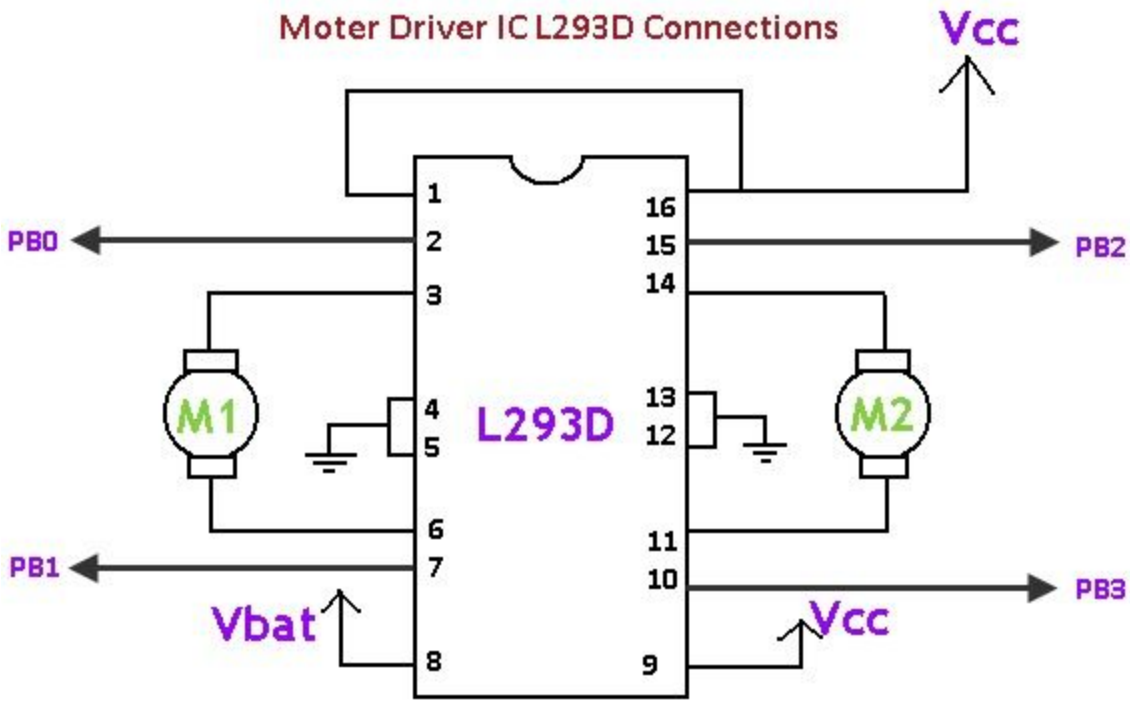
This is followed by a 78xx I.C. which is voltage regulator, where xx denotes its output voltage. It removes the ripples from the voltage and the voltage looks like:



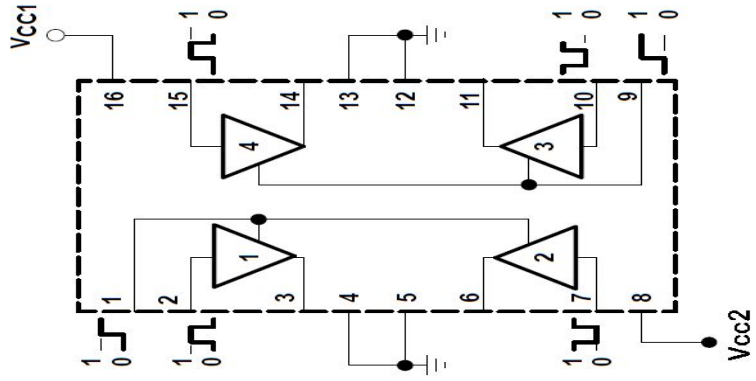
Hence we arrive at a regulated voltage as we wished to. The capacitor after this I.C. is used to take away whatever little ripples are left.

Motor Driver IC's (L293D) :

The motor Driver Circuit L293D is a quad comparator IC which is used to drive motors forward and backward as required. Internally it contains 4 H bridges which are used to drive motors. Block Diargram is shown Below :



The job of the Comparator is to compare the input voltage with a reference voltage and give a logical high if the Input voltage is greater than the reference voltage and a logical low otherwise. The comparator H bridge is shown below.



The Two motors as shown in the above figure are connected to the output pins and the direction of the motor is decided by the voltage that is supplied to the pins I1, I2, I3, I4.
I1, I2 - For motor 1
I3, I4 - For motor 2.

This motor driver IC can be interfaced with out ATmega16 Development Board and a code can be written as follows to run the motors as required.

Note : A PWM across the motors can be generated using Enable pins on this IC.

Code :

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
#define setit1 0
void W()
{
        PORTC=(1<<PC0)|(1<<PC4);

}
void A()
{
        PORTC=(1<<PC0)|(1<<PC5);

}
void S()
{
        PORTC=(1<<PC1)|(1<<PC5);
}
void D()
{
        PORTC=(1<<PC1)|(1<<PC4);
}
int main(void)
{
        DDRC=setit;
        DDRD=setit;
        PORTC=setit1;
        PORTD=(1<<PD4)|(1<<PD5);
   while(1)
   {
                    W();
                   _delay_ms(500);
                   A();
```

```
                _delay_ms(500);
                S();
                _delay_ms(500);
                D();
                _delay_ms(500);
                //TODO:: Please write your application code
    }
}
```

ADC (Analog To Digital Conversion) :

The ATmega16 has the capability to take analog input from some of its pins and some of the pins have been reserved for this purpose. These pins are PA0, PA1, PA2, PA3, PA4, PA5, PA6, PA 7.
These pins can take analog inputs and convert it into a 10 bit digital output i.e. a potential difference of 0 - 5 V can be converted into a data packet which has readings from 0 to 1023.

Specific Registers are set in the ATmega16 chip which carry out this conversion and store the result . These registers are explained below.

ADMUX – ADC Multiplexer Selection Register

The ADMUX register is as follows.

Bit	7	6	5	4	3	2	1	0	
	REFS1	**REFS0**	**ADLAR**	**MUX4**	**MUX3**	**MUX2**	**MUX1**	**MUX0**	**ADMUX**
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	

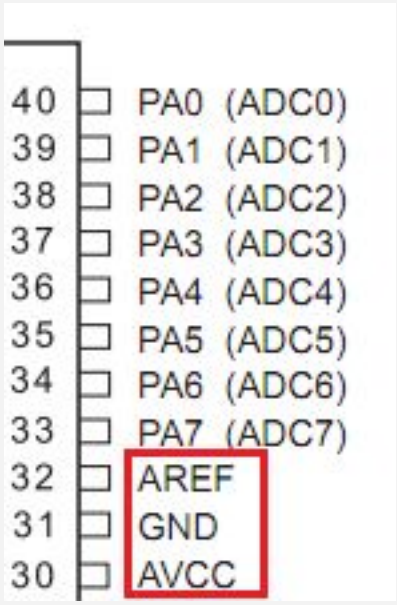
ADMUX Register

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bits 7:6 ï REFS1:0 ï Reference Selection Bits** – These bits are used to choose the reference voltage. The following combinations are used.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Reference Voltage Selection



ADC Voltage Reference Pins

The ADC needs a reference voltage to work upon. For this we have a three pins AREF, AVCC and GND. We can supply our own reference voltage across AREF and GND. For this, **choose the first option**. Apart from this case, you can either connect a capacitor across AREF pin and ground it to prevent from noise, or you may choose to leave it unconnected. If you want to use the VCC (+5V), **choose the second option**. Or else, **choose the last option** for internal Vref. Let's choose the second option for Vcc = 5V.

- **Bit 5 ï ADLAR ï ADC Left Adjust Result** – Make it '1' to Left Adjust the ADC Result. We will discuss about this a bit later.
- **Bits 4:0 ï MUX4:0 ï Analog Channel and Gain Selection Bits** – There are 8 ADC channels (PA0…PA7). You can choose one specific channel by setting these bits. Since

there are 5 bits, it consists of 2^5 = 32 different conditions. However, we are concerned only with the first 8 conditions. Initially, all the bits are set to zero.

ADCSRA – ADC Control and Status Register A

The ADCSRA register is as follows.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

ADCSRA Register

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bit 7 ï ADEN ï ADC Enable** – As the name says, it enables the ADC feature. Unless this is enabled, ADC operations cannot take place across PORTA i.e. PORTA will behave as GPIO pins.
- **Bit 6 ï ADSC ï ADC Start Conversion** – Write this to '1' before starting any conversion. This 1 is written as long as the conversion is in progress, after which it returns to zero. Normally it takes 13 ADC clock pulses for this operation. But when you call it for the first time, it takes 25 as it performs the initialization together with it.
- **Bit 5 ï ADATE ï ADC Auto Trigger Enable** – Setting it to '1' enables auto-triggering of ADC. ADC is triggered automatically at every rising edge of clock pulse. View the SFIOR register for more details.
- **Bit 4 ï ADIF ï ADC Interrupt Flag** – Whenever a conversion is finished and the registers are updated, this bit is set to '1' automatically. Thus, this is used to check whether the conversion is complete or not.
- **Bit 3 ï ADIE ï ADC Interrupt Enable** – When this bit is set to '1', the ADC interrupt is enabled. This is used in the case of interrupt-driven ADC.
- **Bits 2:0 ï ADPS2:0 ï ADC Prescaler Select Bits** – The prescaler (division factor between XTAL frequency and the ADC clock frequency) is determined by selecting the proper combination from the following.

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADC Prescaler Selections

Assuming XTAL frequency of 16MHz and the frequency range of 50kHz-200kHz, we choose a prescaler of 128.
Thus, F_ADC = 16M/128 = 125kHz.

And the Last but very Important register THE ADC Register;

ADCL and ADCH – ADC Data Registers

The result of the ADC conversion is stored here. Since the ADC has a resolution of 10 bits, it requires 10 bits to store the result. Hence one single 8 bit register is not sufficient. We need two registers – ADCL and ADCH (ADC Low byte and ADC High byte) as follows. The two can be called together as ADC.

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	**ADCH**
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 0

ADC Data Registers (ADLAR = 0)

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	*ADLAR = 1*

ADC Data Registers (ADLAR = 1)

You can very well see the the effect of ADLAR bit (in ADMUX register). Upon setting ADLAR = 1, the conversion result is left adjusted.

Note : AS SOON AS THE ADCH REGISTER IS READ BY THE MICROCONTROLLER THE VALUES OF ADCH AND ADCL ARE DUMPED. Hence while using ADLAR =1 mode we can get an accuracy of 2^8 only.

A sample Code That Uses ADC is given Below.

```
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/io.h>
#define setit ~0
#define setit1 0
int main(void)
{
	DDRB = setit;
	DDRA = setit1;
	PORTB = setit1;
	ADCSRA = (1<<ADEN)|(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2);
	ADMUX = (1<<REFS0);
	while(1)
	{
		//TODO:: Please write your application code
		ADCSRA |=(1<<ADSC);
		int a = ADC;
		if (a>820)
		{
			PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3)|(1<<PB4);
		}
		else if (a>615)
```

```
                {
                        PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3);
                }
                else if (a>410)
                {
                        PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2);
                }
                else if (a>205)
                {
                        PORTB=(1<<PB0)|(1<<PB0);
                }
                else
                {
                        PORTB=(1<<PB0);
                }
        }
}
```

This code utilises the data given by the POTENTIOMETER AT PA0 and lights up status LED's.

Timer0/ 8 - Bit Timer with PWM.

- Timers are made up of registers, whose value automatically increases/decreases. Thus, the terms timer/counter are used interchangeably.
- In AVR, there are three types of timers – TIMER0, TIMER1 and TIMER2. Of these, TIMER1 is a 16-bit timer whereas others are 8-bit timers.
- Prescalers are used to trade duration with resolution.
- As we want less precision but higher values we choose our prescalers to bigh high in the range of 256 to 1024.

TCNT0 Register

The **Timer/Counter Register** – TCNT0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
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	

TCNT0 Register

This is where the uint 8-bit counter of the timer resides. The value of the counter is stored here and increases/decreases automatically. Data can be both read/written from this register.

TCCR0 Register

The **Timer/Counter Control Register** – TCCR0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	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	

TCCR0 Register

Right now, we will concentrate on the highlighted bits. The other bits will be discussed as and when necessary. By selecting these three **Clock Select Bits**, **CS02:00**, we set the timer up by choosing proper prescaler. The possible combinations are shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(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 T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Clock Select Bit Description

Bit 6,3 ï WGM01:0 ï Waveform Generation Mode – These bits can be set to either “00” or “01” depending upon the type of PWM you want to generate. Here’s the look up table.

DAY 1

The first day of the workshop dealt with “ The Theory Of Everything”. 4 major topics were discussed.

1. Physics

The principle of the Self Balancing Bot, i.e. Inverted Pendulum. The Vertical position is the position of Unstable equilibrium. We intend to use accelerometer and gyroscope to interface and measure the angle made by the bot with the vertical. Gyroscope to be used as it gives a

precise reading while accelerometer gives an acurate but less precise reading. So a complementary filter has to be used to get the actual value of “theta” at all times.

Complementary Filter :

$$angle = 0.98 * (angle + gyrData * dt) + 0.02 * (accData)$$

2. Bitwise Operators

The discussed Bitwise Operators were “^”,”|”,”&”,”~”,”>>”,”<<”

“^” - Bitwise XOR Operator .

Truth Table :

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

“|” - Bitwise OR Operator

Truth Table:

A	B	X = A+B
0	0	0
0	1	1
1	0	1
1	1	1

“&” - Bitwise AND Operator

Truth Table:

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

“~” - Bitwise NOT Operator
Truth Table:

Input	Output
0	1
1	0

“<<”, “>>” - Left Shift/Right Shift Operators
Operators which can shift the whole binary thread left or right appending or proceeding it by zeroes by the given parameters.

3. Basic Coding

   Constants, Variables, Loops & Condition Statements were discussed.

4. AVR Coding

   Basic Etiquettes of AVR coding were discussed. So were common common practices. Then Registers “PORTX”, “DDRX”,”PINX” were discussed.

“DDRX” - Stores the pinMode of each pin in the port ‘X’, 1 for output and 0 for input.
“PORTX” - Write only Register, 1 if output is high else 0, can be set for pins in output mode only.
“PINX” - Read Only Register, 1 if input is high, 0 if low, can be read for pins in input mode only in port “X”.

5. Setting, Clearing, Checking, Toggling a bit

   Setting : a|=(1<<x)

   Clearing: a&=~(1<<x)

   Checking: a&=(1<<x)

   Toggling: a^=(1<<x)

   If 'X' is the position of the bit to be toggled.

Programs for basic I/O functions on ATmega16 (LED)

1. Switching LED On:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
int main(void)
{
	DDRB=setit;
	PORTB=setit;
	while(1)
	 {
		//TODO:: Please write your Application code
	}
}
```

2. Blinking LED's

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
int main(void)
{
	DDRB=setit;
	PORTB=setit;
	while(1)
	 {
		 //TODO:: Please write your Application code
			_delay_ms(500);
			PORTB=(~PORTB);
	}
}
```

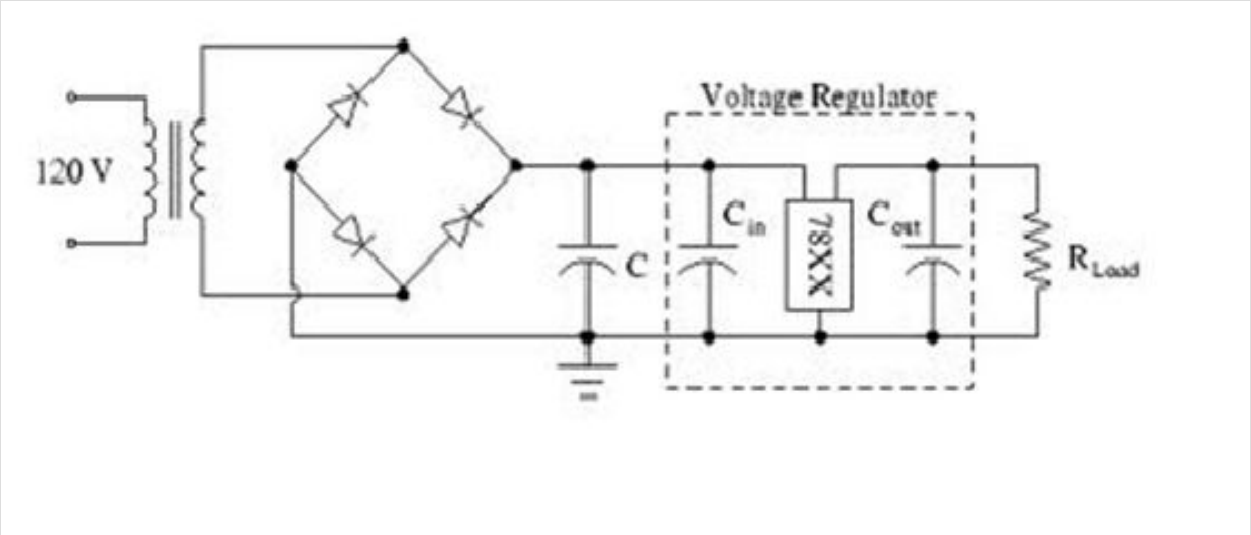
3. Pattern LED's

```
#define F_CPU 16000000UL
```

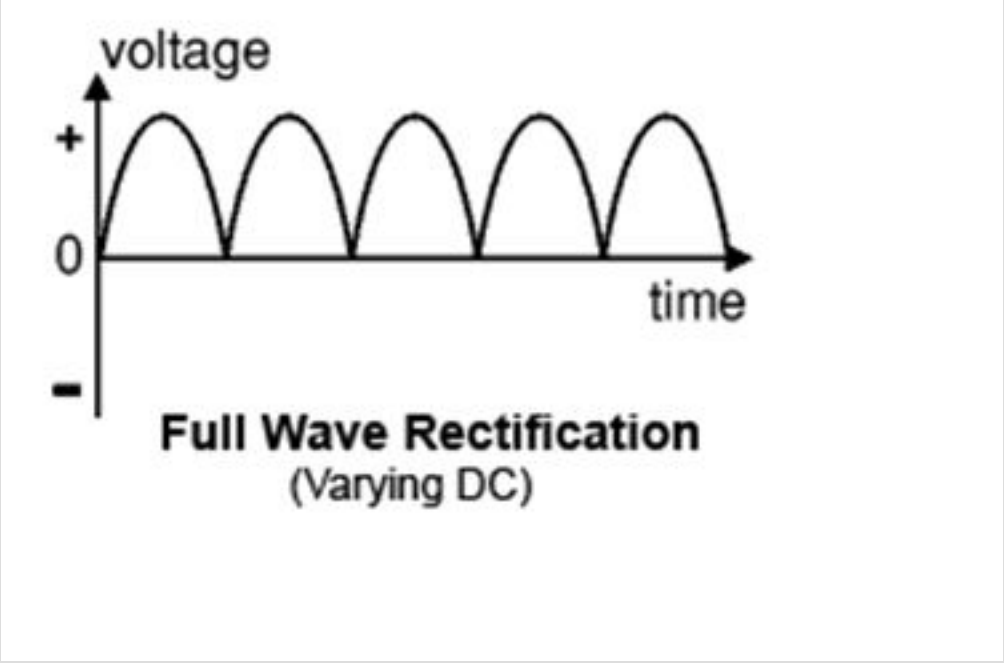
```
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
#define setit1 0
int main(void)
{
	DDRB=setit;
	PORTB=setit1;
	for (int i=0; i<8; i++)
	{
		for (int j=i; j<8; j++)
		{
			PORTB|=(1<<j);
			_delay_ms(200);
			PORTB^=(1<<j);
		}
		for (int k=7; k>=i; k--)
		{
			PORTB|=(1<<k);
			_delay_ms(200);
			PORTB^=(1<<k);
		}
		PORTB|=(1<<i);
		_delay_ms(200);
	}
}
```

DAY 2 & 3

Rectifier Circuit :
They convert A.C. voltage into regulated D.C voltage. A rectifier uses the polar property of a diode i.e. a diode allows only unidirectional current to pass through it which happens when the P- terminal is at higher voltage then the N- terminal of the diode. This configuration is known as forward bias of the diode. The following shows a basic circuit for the rectifier circuit.



The diodes at the right top and left bottom are in forward bias in the positive half cycle and the diodes at the left top and the bottom right are in forward bias in the negative half cycle of the voltage. Hence current always flows in one direction in the circuit. The first capacitor is known as filter which converts the varying signals (as shown below) to ripple voltage.



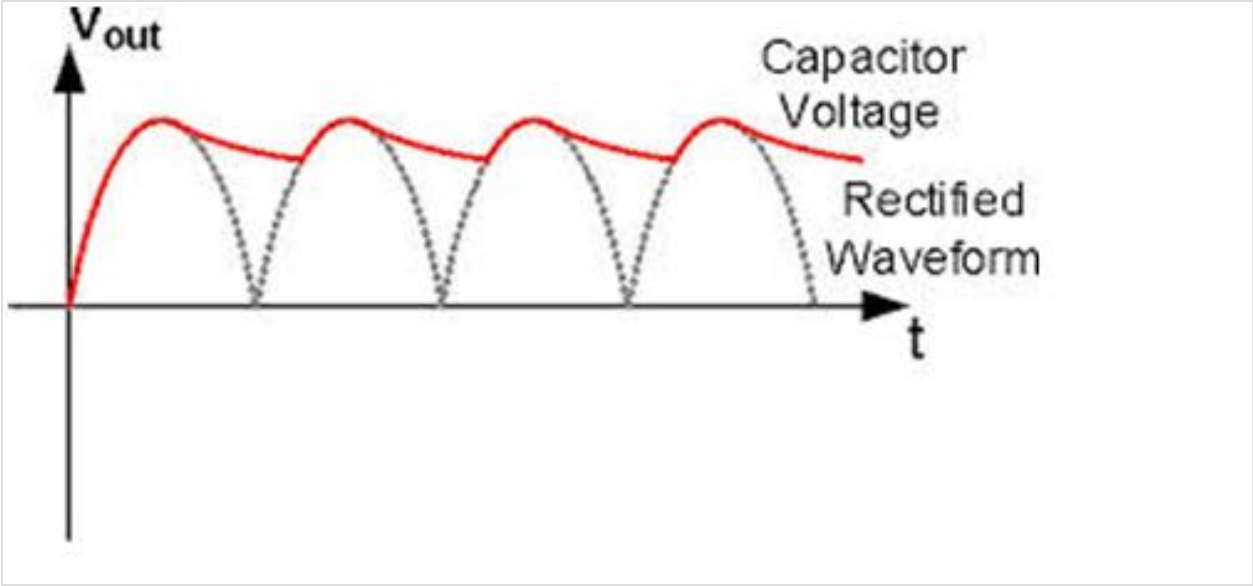
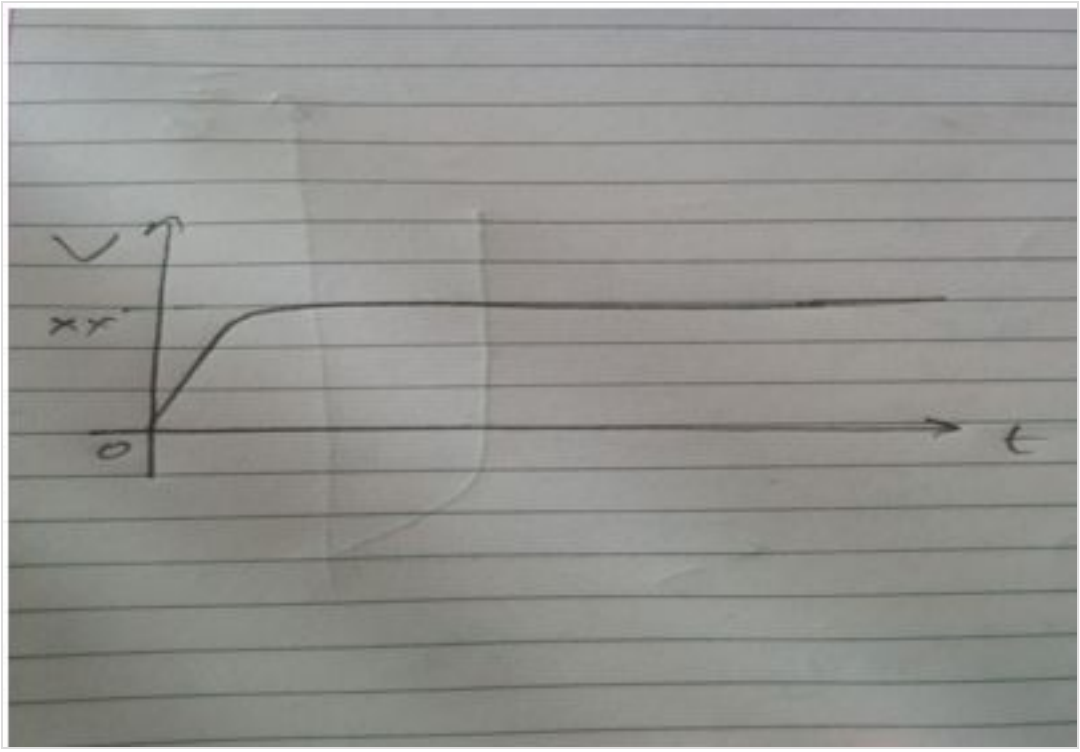


Fig 1 shows voltage after only rectifier and fig 2 shows voltage after filtering the voltage using a capacitor. The filtering works on the property of capacitor to store charge when voltage across terminals is increasing and give it away when the reverse happens i.e. the voltage across its terminals decreases.

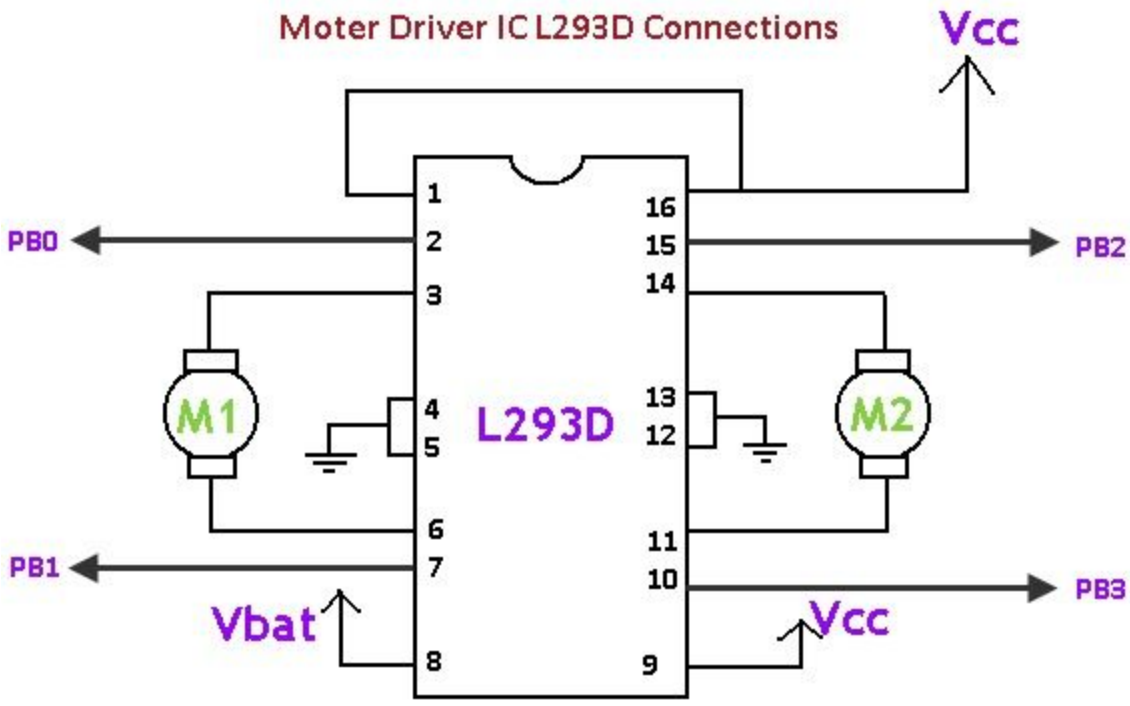
This is followed by a 78xx I.C. which is voltage regulator, where xx denotes its output voltage. It removes the ripples from the voltage and the voltage looks like:



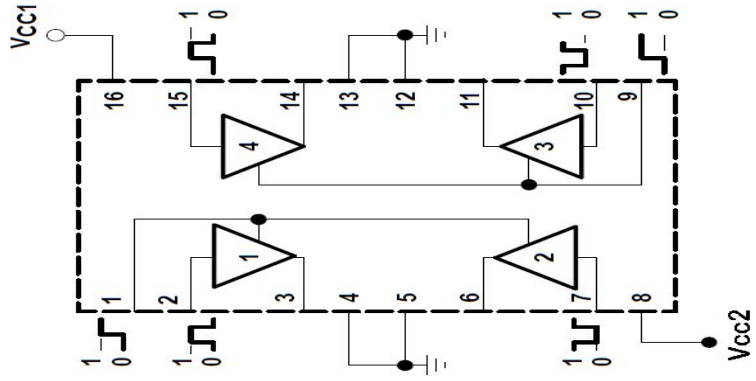
Hence we arrive at a regulated voltage as we wished to. The capacitor after this I.C. is used to take away whatever little ripples are left.

Motor Driver IC's (L293D) :

The motor Driver Circuit L293D is a quad comparator IC which is used to drive motors forward and backward as required. Internally it contains 4 H bridges which are used to drive motors. Block Diargram is shown Below :



The job of the Comparator is to compare the input voltage with a reference voltage and give a logical high if the Input voltage is greater than the reference voltage and a logical low otherwise. The comparator H bridge is shown below.



The Two motors as shown in the above figure are connected to the output pins and the direction of the motor is decided by the voltage that is supplied to the pins I1, I2, I3, I4.
I1, I2 - For motor 1
I3, I4 - For motor 2.

This motor driver IC can be interfaced with out ATmega16 Development Board and a code can be written as follows to run the motors as required.

Note : A PWM across the motors can be generated using Enable pins on this IC.

Code :

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define setit ~0
#define setit1 0
void W()
{
        PORTC=(1<<PC0)|(1<<PC4);

}
void A()
{
        PORTC=(1<<PC0)|(1<<PC5);

}
void S()
{
        PORTC=(1<<PC1)|(1<<PC5);
}
void D()
{
        PORTC=(1<<PC1)|(1<<PC4);
}
int main(void)
{
        DDRC=setit;
        DDRD=setit;
        PORTC=setit1;
        PORTD=(1<<PD4)|(1<<PD5);
   while(1)
   {
                    W();
                   _delay_ms(500);
                   A();
```

```
                _delay_ms(500);
                S();
                _delay_ms(500);
                D();
                _delay_ms(500);
                //TODO:: Please write your application code
    }
}
```

ADC (Analog To Digital Conversion) :

The ATmega16 has the capability to take analog input from some of its pins and some of the pins have been reserved for this purpose. These pins are PA0, PA1, PA2, PA3, PA4, PA5, PA6, PA 7.
These pins can take analog inputs and convert it into a 10 bit digital output i.e. a potential difference of 0 - 5 V can be converted into a data packet which has readings from 0 to 1023.

Specific Registers are set in the ATmega16 chip which carry out this conversion and store the result . These registers are explained below.

ADMUX – ADC Multiplexer Selection Register

The ADMUX register is as follows.

Bit	7	6	5	4	3	2	1	0	
	REFS1	**REFS0**	**ADLAR**	**MUX4**	**MUX3**	**MUX2**	**MUX1**	**MUX0**	**ADMUX**
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	

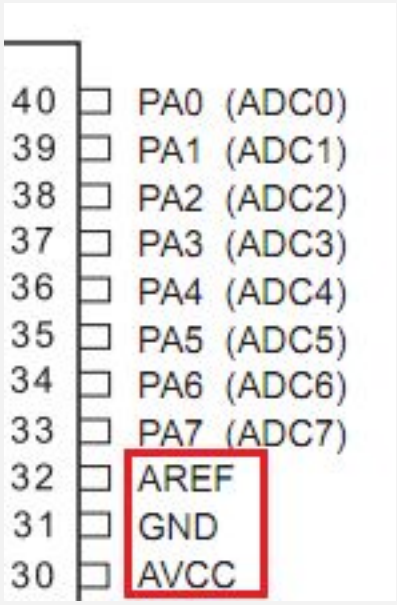
ADMUX Register

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bits 7:6 ï REFS1:0 ï Reference Selection Bits** – These bits are used to choose the reference voltage. The following combinations are used.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Reference Voltage Selection



ADC Voltage Reference Pins

The ADC needs a reference voltage to work upon. For this we have a three pins AREF, AVCC and GND. We can supply our own reference voltage across AREF and GND. For this, **choose the first option**. Apart from this case, you can either connect a capacitor across AREF pin and ground it to prevent from noise, or you may choose to leave it unconnected. If you want to use the VCC (+5V), **choose the second option**. Or else, **choose the last option** for internal Vref. Let's choose the second option for Vcc = 5V.

- **Bit 5 ï ADLAR ï ADC Left Adjust Result** – Make it '1' to Left Adjust the ADC Result. We will discuss about this a bit later.
- **Bits 4:0 ï MUX4:0 ï Analog Channel and Gain Selection Bits** – There are 8 ADC channels (PA0…PA7). You can choose one specific channel by setting these bits. Since

there are 5 bits, it consists of 2^5 = 32 different conditions. However, we are concerned only with the first 8 conditions. Initially, all the bits are set to zero.

ADCSRA – ADC Control and Status Register A

The ADCSRA register is as follows.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

ADCSRA Register

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bit 7 ï ADEN ï ADC Enable** – As the name says, it enables the ADC feature. Unless this is enabled, ADC operations cannot take place across PORTA i.e. PORTA will behave as GPIO pins.
- **Bit 6 ï ADSC ï ADC Start Conversion** – Write this to '1' before starting any conversion. This 1 is written as long as the conversion is in progress, after which it returns to zero. Normally it takes 13 ADC clock pulses for this operation. But when you call it for the first time, it takes 25 as it performs the initialization together with it.
- **Bit 5 ï ADATE ï ADC Auto Trigger Enable** – Setting it to '1' enables auto-triggering of ADC. ADC is triggered automatically at every rising edge of clock pulse. View the SFIOR register for more details.
- **Bit 4 ï ADIF ï ADC Interrupt Flag** – Whenever a conversion is finished and the registers are updated, this bit is set to '1' automatically. Thus, this is used to check whether the conversion is complete or not.
- **Bit 3 ï ADIE ï ADC Interrupt Enable** – When this bit is set to '1', the ADC interrupt is enabled. This is used in the case of interrupt-driven ADC.
- **Bits 2:0 ï ADPS2:0 ï ADC Prescaler Select Bits** – The prescaler (division factor between XTAL frequency and the ADC clock frequency) is determined by selecting the proper combination from the following.

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADC Prescaler Selections

Assuming XTAL frequency of 16MHz and the frequency range of 50kHz-200kHz, we choose a prescaler of 128.
Thus, F_ADC = 16M/128 = 125kHz.

And the Last but very Important register THE ADC Register;

ADCL and ADCH – ADC Data Registers

The result of the ADC conversion is stored here. Since the ADC has a resolution of 10 bits, it requires 10 bits to store the result. Hence one single 8 bit register is not sufficient. We need two registers – ADCL and ADCH (ADC Low byte and ADC High byte) as follows. The two can be called together as ADC.

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	**ADCH**
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 0

ADC Data Registers (ADLAR = 0)

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	*ADLAR = 1*

ADC Data Registers (ADLAR = 1)

You can very well see the the effect of ADLAR bit (in ADMUX register). Upon setting ADLAR = 1, the conversion result is left adjusted.

Note : AS SOON AS THE ADCH REGISTER IS READ BY THE MICROCONTROLLER THE VALUES OF ADCH AND ADCL ARE DUMPED. Hence while using ADLAR =1 mode we can get an accuracy of 2^8 only.

A sample Code That Uses ADC is given Below.

```
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/io.h>
#define setit ~0
#define setit1 0
int main(void)
{
	DDRB = setit;
	DDRA = setit1;
	PORTB = setit1;
	ADCSRA = (1<<ADEN)|(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2);
	ADMUX = (1<<REFS0);
	while(1)
	{
		//TODO:: Please write your application code
		ADCSRA |=(1<<ADSC);
		int a = ADC;
		if (a>820)
		{
			PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3)|(1<<PB4);
		}
		else if (a>615)
```

```
                {
                        PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3);
                }
                else if (a>410)
                {
                        PORTB=(1<<PB0)|(1<<PB1)|(1<<PB2);
                }
                else if (a>205)
                {
                        PORTB=(1<<PB0)|(1<<PB0);
                }
                else
                {
                        PORTB=(1<<PB0);
                }
        }
}
```

This code utilises the data given by the POTENTIOMETER AT PA0 and lights up status LED's.

Timer0/ 8 - Bit Timer with PWM.

- Timers are made up of registers, whose value automatically increases/decreases. Thus, the terms timer/counter are used interchangeably.
- In AVR, there are three types of timers – TIMER0, TIMER1 and TIMER2. Of these, TIMER1 is a 16-bit timer whereas others are 8-bit timers.
- Prescalers are used to trade duration with resolution.
- As we want less precision but higher values we choose our prescalers to bigh high in the range of 256 to 1024.

TCNT0 Register

The **Timer/Counter Register** – TCNT0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
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	

TCNT0 Register

This is where the uint 8-bit counter of the timer resides. The value of the counter is stored here and increases/decreases automatically. Data can be both read/written from this register.

TCCR0 Register

The **Timer/Counter Control Register** – TCCR0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	FOC0	**WGM00**	**COM01**	**COM00**	**WGM01**	**CS02**	**CS01**	**CS00**	**TCCR0**
Read/Write	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	

TCCR0 Register

Right now, we will concentrate on the highlighted bits. The other bits will be discussed as and when necessary. By selecting these three **Clock Select Bits**, **CS02:00**, we set the timer up by choosing proper prescaler. The possible combinations are shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(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 T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Clock Select Bit Description

Bit 6,3 – WGM01:0 – Waveform Generation Mode – These bits can be set to either “00” or “01” depending upon the type of PWM you want to generate. Here’s the look up table.

Bit 5,4 – COM01:0 – Compare Match Output Mode – These bits are set in order to control the behavior of Output Compare pin in accordance with the WGM01:0 bits.

TIMSK Register

The **Timer/Counter Interrupt Mask** – TIMSK Register is as follows. It is a common register for all the three timers. For TIMER0, bits 1 and 0 are allotted. Right now, we are interested in the 0th bit **TOIE0**. Setting this bit to '1' enables the TIMER0 overflow interrupt.

TIMSK Register

TIFR Register

The **Timer/Counter Interrupt Flag Register**- TIFR is as follows. Even though we are not using it in our code, you should be aware of it.

TIFR Register

This is also a register shared by all the timers. Even here, bits 1 and 0 are allotted for TIMER0. At present we are interested in the 0th bit **TOV0** bit. This bit is set (one) whenever TIMER0 overflows. This bit is reset (zero) whenever the Interrupt Service Routine (ISR) is executed. If there is no ISR to execute, we can clear it manually by writing one to it

Here Are some sample codes using timers and PWM

1. LED BLINKING WITHOUT DELAY

```
#include <avr/io.h>
#define setit ~0
#define setit1 0
int main(void)
{
        DDRB = setit;
        PORTB= setit1;
```

```
	long ctr=0;
	TCCR0 = (1<<CS02)|(1<<CS00);
	TCNT0=0;
	while(1)
	{
		if (TCNT0==255)
		{
			ctr++;
			TCNT0=0;
		}
		if ((TCNT0 + ctr*255)==4650)
		{
			PORTB^=~0;
			ctr=0;
		}
		else
		{
		}

		//TODO:: Please write your application code
	}
}
```

2. PWM + ADC duty cycles managed by voltage supplied at PA0;

```
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/io.h>
#define setit ~0
#define setit1 0
int main(void)
{
	DDRB = setit;
	DDRA = setit1;
	PORTB = setit1;
	ADCSRA = (1<<ADEN)|(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2);
	ADMUX = (1<<REFS0);
	while(1)
	{
		//TODO:: Please write your application code
		ADCSRA |=(1<<ADSC);
		//check adsc
		int a = ADC;
		if (a>820)
		{
			TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00)|(1<<CS02);
			OCR0 = 0;
```

```
                }
                else if (a>615)
                {
                        TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00)|(1<<CS02);
                        OCR0 = 26;
                }
                else if (a>410)
                {
                        TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00)|(1<<CS02);
                        OCR0 = 102;
                }
                else if (a>205)
                {
                        TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00)|(1<<CS02);
                        OCR0 = 153;
                }
                else
                {
                        TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00)|(1<<CS02);
                        OCR0 = 204;
                }
        }
}
```

3. FADING LED USING PWM

```
#include<avr/io.h>
//#include<avr/interrupt.h>
#define F_CPU 16000000UL
#include<util/delay.h>
#define setit ~0
#define setit1 0
float Dutycycle = 0;


int main()
{
        //sei();
        DDRB = setit;
        TCCR0 = (1<<WGM00)|(1<<WGM01)|(1<<COM01)|(1<<CS00)|(1<<CS02)|(1<<COM00);
        //TIMSK = (1<<TOIE0);
        OCR0 = (Dutycycle/100)*255;
        TCNT0 = 0;
        while (1)
        {
                for (int i=0; i<10; i++)
                {
                        Dutycycle+=10;
                        OCR0 = (Dutycycle/100)*255;
```

```
                _delay_ms(100);
            }
            for(int j=0; j<10; j++)
            {
                Dutycycle-=10;
                OCR0 = (Dutycycle/100)*255;
                _delay_ms(100);
            }
            _delay_ms(300);
        }
}
```

Ėmsdqq t osr :

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 Enabled** bit and the**Interrupt Enable** bit of the specific interrupt.

Interrupt Service Routine or Interrupt Handler

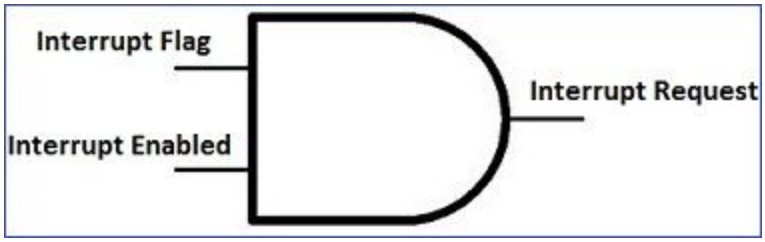
An **Interrupt Service Routine (ISR)** or **Interrupt Handler** is a piece of code that should be execute when an interrupt is triggered. Usually each enabled interrupt has its own ISR. In AVR assembly language each ISR **MUST**end with the **RETI** instruction which indicates the end of the ISR.

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

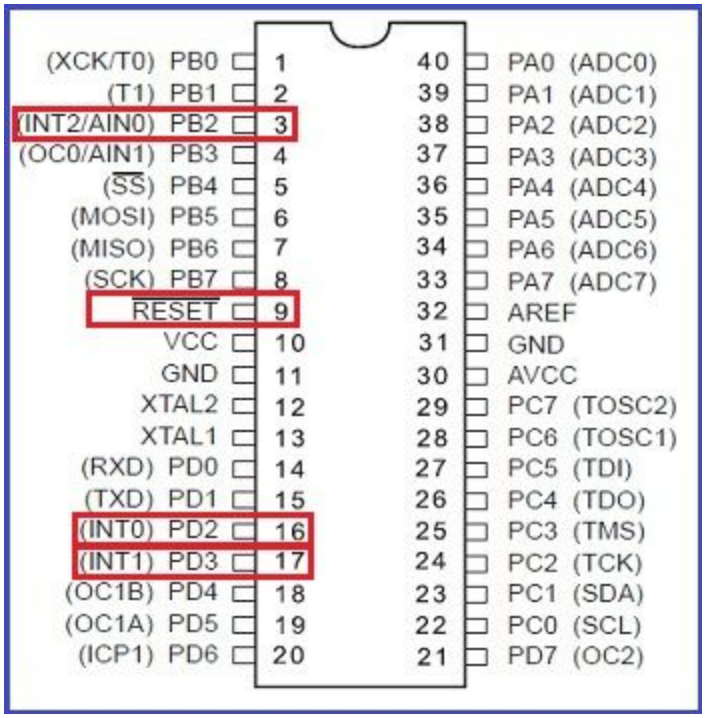
Apart from the enabled bits for the specific interrupts the global interrupt enabled bit **MUST** be enabled for interrupts to be activated in the microcontoller.

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.

Interrupt sources provided with the AVR microcontroller

The AVR 8-bits microcontroller provide both internal and external interrupt sources. The internal interrupts are associated with the microcontroller's peripherals. That is the **Timer/Counter, Analog Comparator, etc**. The external interrupts are triggered via external pins. The figure below shows the pins, on which the external interrupts can be triggered, for an AVR 8-bit microcontroller. On this microcontroller there are four (4) external interrupts:

1. The **RESET** interrupt - Triggered from pin 9.
2. **External Interrupt 0 (INT0)** - Triggered from pin 16.
3. **External Interrupt 1 (INT1)** - Triggered from pin 17.
4. **External Interrupt 2 (INT2)** - Triggered from pin 3.



Very Important

When writing assembly codes for your AVR microcontroller utilizing the interrupt feature the following **MUST** be observed:

- The interrupt **MUST** be enabled by setting its enabled bit in the appropriate I/O register.
- The Global Interrupt bit, the **I** bit, in the microcontroller's status register (SREG) **MUST** also be enabled.
- The stack **MUST** be initialized. When an interrupt is being service the microcontroller need to store critical information on the stack and so it must be initialized.
- The Interrupt Service Routine (ISR) **MUST** end with the **RETI** instruction, which indicates the end of the ISR. The microcontroller needs to know when it reaches the end of the ISR so it can return to its previous task.

Steps taken in servicing an interrupt

Upon the triggering of an interrupt the following sequence is followed by the microcontroller providing that the both the specific interrupt and global interrupts are enabled in the microcontroller: