GESTURE CONTROL ROBOTCS DOCUMENTATION

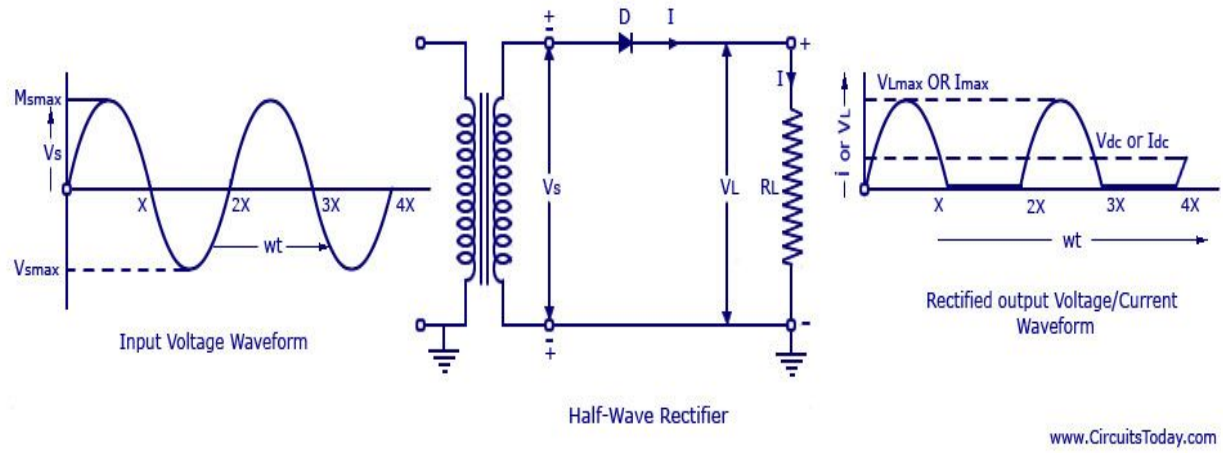
DAY-1

Basic Analogies of a Robot

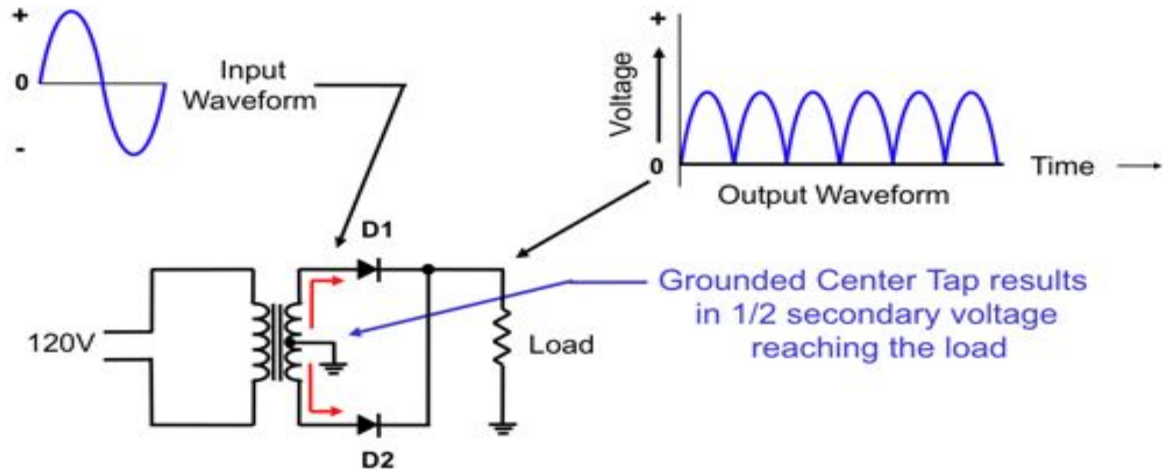
1. **Power Supply**
2. **Sensors**
3. **Microcontroller**
4. **Motor**

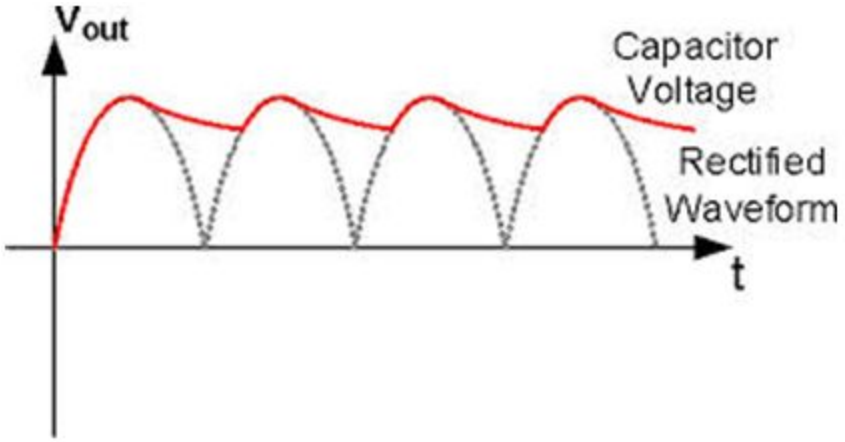
Power Supply

We use DC supply for running the robot circuits. In case of an AC supply, rectifiers are used to convert the input signal to DC. The different types of rectifiers are listed below:
a) Half-wave rectifiers: This rectifier converts only one half of the input AC supply, and leaves the other half.

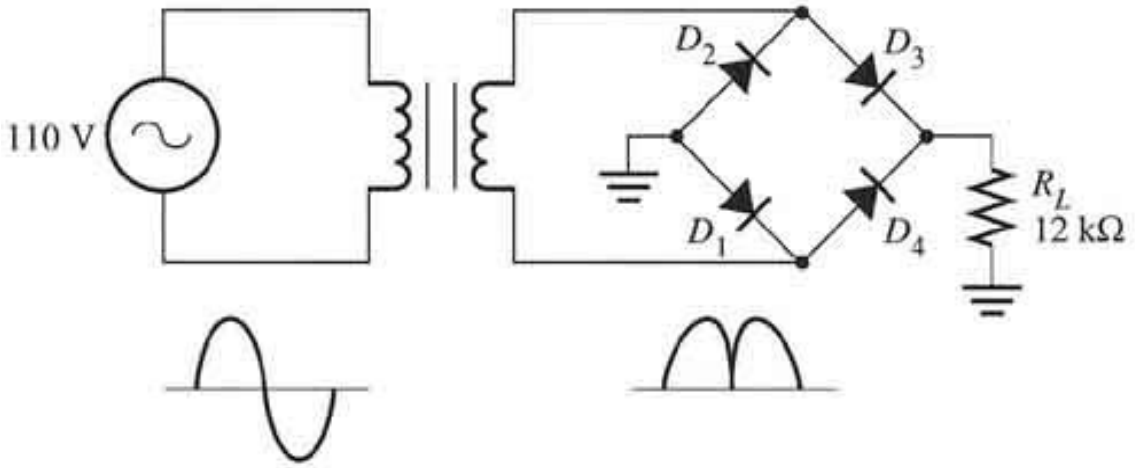


b) Full-wave rectifiers: This rectifier converts both halves of the AC supply to DC supply.





c) Bridge rectifiers: It is similar to full wave rectifier.



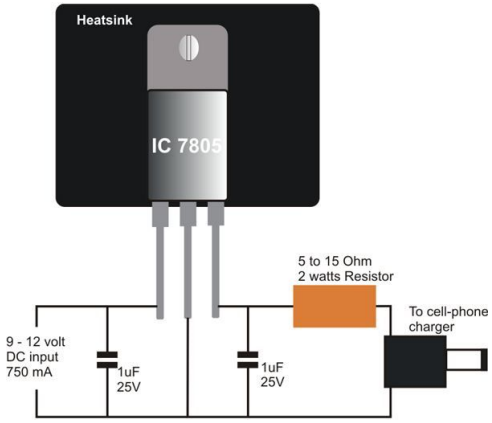
****The RMS value of voltage is the value for DC supply after rectification.**
****Adapters are used to power the robots. Batteries are not used as they are unreliable.**

Types Of Electronic Circuits:
a) Analog Circuits.
b) Digital Circuits.

Analog Circuits are outdated. Digital circuits work on basic high and low values of signals.

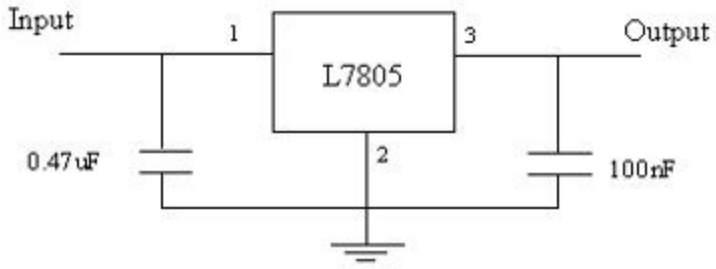
Voltage limit for most of robot circuits is around 5V usually.
The adapters used for running the robots are of the range 9/12 V. To bring down the voltage to usable form and limit, Integrated Circuits(ICs) are used. General ICs used are

LM78XX series(where XX represents the output voltage of the IC).



The above is a simple diagram of LM7805.

Every IC runs on values and ranges specified by its Data Sheet.
The basic circuit of LM7805 is somewhat like



****Function of capacitors- As we know, capacitors allow the AC part of a signal through them. We use the same property of capacitors here. The lesser ripples in the input voltage, the better. The capacitors lessen the ripples in the input voltage to provide almost constant DC supply to the robot circuit.**

Microcontroller

Microcontroller is the brain of the robot. There is a difference between a microcontroller and a microprocessor.
A microprocessor has arithmetic and logical units which function as per the input and give the output based on logical analysis.
On the other hand, a microcontroller is a pre-programmed circuit that takes a range of input and gives a range of output as per the instructions fed to it. Also, a microcontroller has its own memory which is not there in a microprocessor.

An example of a microcontroller is ATmega16. It is a basic microcontroller.

There are three types of memories used in a microcontroller:

- **Flash memory- Flash memory is an electronic non-volatile computer storage medium that can be electrically erased and reprogrammed**
- **EEPROM- Electronically Erasable and programmable Read Only Memory is a type of memory that retains its data when its power supply is switched off.**
- **SRAM- Static Random Access memory holds data as long as the power is switched on. It loses everything once the power supply is gone.**

DAY-2

BITWISE Operators

a)AND
b)OR
c)XOR
d)NOT
e)Left Shift(<<)
f)Right Shift(>>)

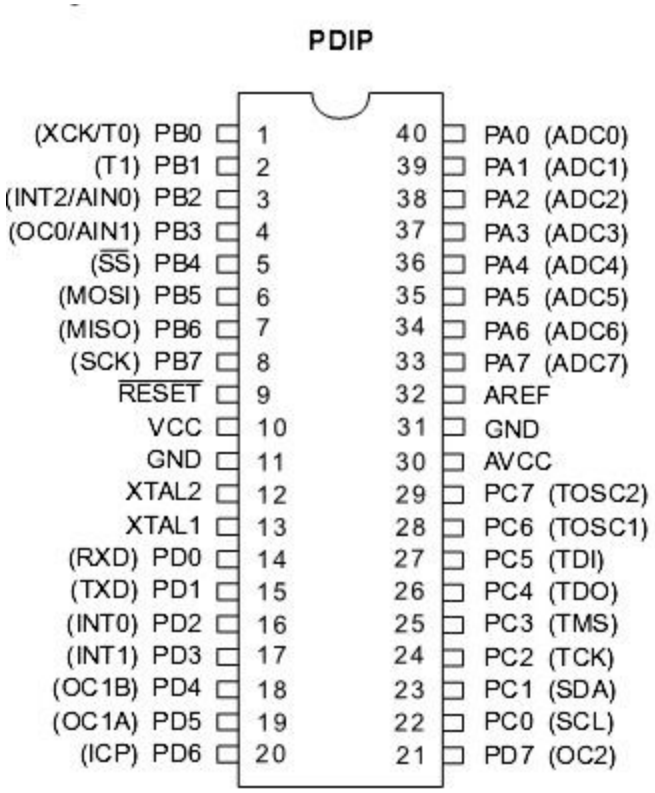
Basic Operations

a)SET: this operation sets any bit value to 1. The code for it is A|(1<<i).
where A is the given byte and i is the bit number to be set to 1.

b)CLEAR: this operation sets any bit to 0. The code fir it is A&(~(1<<i)).

c)TOGGLE: this operation reverses the value of any bit. XOR operator is used. The code for it is A^(1<<i).

ATmega16 Pin Configuration Details



A,B,C,D are the respective ports with 8 pins each(each pin a bit). Out of these, only port A is used as an analog to digital converter(ADC).
Reset pin is used to reset the controller for new usage.
VCC is the input voltage point of the microcontroller. GND represents the grounded pin. XTAL 1&2 are used for providing external oscillator to the circuit if needed. AVCC is the pin for supply voltage to port A if ADC is to be used. AREF is the pin used to set the upper limit of voltage to be used in the ADC.

Registers

Storage memories in microcontrollers are registers. We'll use the registers in our coding:

a. **DDRX**
b. **PINX**
c. **PORTX**

(X stands for port name)

DDRX is used to set the port pins as output or input points. If 0 is assigned to a pin, it is input pin; if 1 is assigned, it is output.

PINX stores the state of the port.

PORTX assigns the usage state of pins of the port. It gives the output value of the pins.

Day 2 & 3 were spent in simple coding implementations like lighting LEDS and patterns in LEDS.
The next topic which comes is the ADC i.e. Analog to Digital Converter.

DAY-3

Analog to Digital Converter

The signal given as input to the microcontroller are in terms of voltages(analog signals). They need to be converted to digital signals for the microcontroller to function. For this, the microcontroller has a separate unit(ADC) for accepting analog signals and converting them to digital ones.

The ADC resides in port A, thus only port A can accept analog signals and convert them to digital.
The ATmega16 features a 10-bit successive approximation ADC. The ADC is connected to an 8-channel Analog Multiplexer which allows 8 single-ended voltage inputs constructed from the pins of Port A. The single-ended voltage inputs refer to 0V (GND). The device also supports 16 differential voltage input combinations.

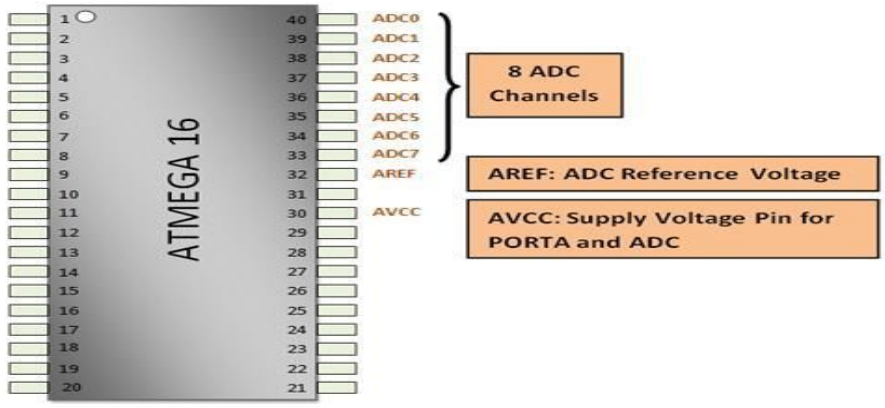
Operation:-The ADC converts an analog input voltage to a 10-bit digital value through successive approximation.
The ADC has two registers:

a. **ADMUX (Analog to Digital Multiplexer)**
b. **ADCSRA(AD control and Status Register A)**

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	

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	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	

The minimum value represents GND and the maximum value represents the voltage on the AREF pin minus 1 LSB. Optionally, AVCC or an internal 2.56V reference voltage may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register.
The analog input channel and differential gain are selected by writing to the MUX bits in ADMUX. Any of the ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended inputs to the ADC.
The ADC is enabled by setting the ADC Enable bit, ADEN in ADCSRA. Voltage reference and input channel selections will not go into effect until ADEN is set. The ADC does not consume power when ADEN is cleared, so it is recommended to switch off the ADC before entering power saving sleep modes.
The ADC generates a 10-bit result which is presented in the ADC Data Registers, ADCH and ADCL. By default, the result is presented right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX.
If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH, to ensure that the content of the Data Registers belongs to the same conversion. Once ADCL is read, ADC access to Data Registers is blocked. This means that if ADCL has been read, and a conversion completes before ADCH is read, neither register is updated and the result from the conversion is lost. When ADCH is read, ADC access to the ADCH and ADCL Registers is re-enabled.



The REF0:1 bits are used to set the voltage limit settings. The MUX bits are used to set the input pins in portA. The ADLAR register is used to select ADCH/ADCL regsters depending upon our precision requirements of the signals. Therefore after the ADMUX register, ADC knows

a. **Reference voltage**
b. **ADC input pins**
c. **Bits to be read**

Coming to the ADCSRA register, the ADEN pin is set when ADC is to be enabled. The ADSC pin is set to begin the analog to digital conversion and sets automatically back to 0 when the conversion is done. Once ADSC-0, we can read the converted value from ADCH

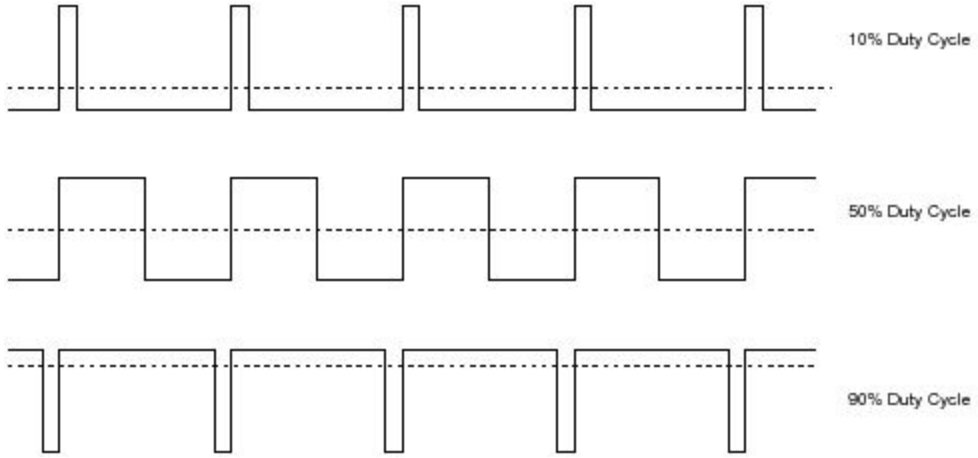
and ADCL. The ADPS bits are used to set the prescaler which is used to set the clock frequency of ADC. The more resolution we want, the higher prescaler we use.

After setting the ADCSRA & ADMUX to desired values, we get started with ADC. The inputs go through portA as analogs and get converted and stored in ADCH & ADCL.

DAY-4

Pulse Width Modulation (PWM)

The digital signal consists of a high and a low in one cycle. The part of cycle for which the signal is high decides the amount of average input of power to the output.



Duty Cycle is define as the ratio of time for which signal is high in a cycle and total cycle time.

The high is 5V and low is 0V. But if the voltage requirement is 3V, what can be done?
In such cases, we vary the duty cycle using timers in the microcontrollers which leads to varying outputs between 0 and 5 V.

This variation is done using timers and clocks in ATmega16 microcontroller. There are three timers in ATmega16:

a. **8-bit timers(2)**
b. **16-bit timer(1)**

TCCR0 and TCCR2 are registers for the two 8-bit timers respectively. TCCR1 is the register for the 16-bit timer.
A unit known as Output Compare Unit(OCU) compares a given value(OCRO) to the TCCR0 value and gives the notification. By varying the OCRO we vary the duty cycle.

Basic programmes on varying the brightness of LEDs were practiced to get the codes clear in the workshop.

The next topic covered is Serial Communication.

DAY-6

Serial Communication

In Telecommunication and Computer Science, serial communication is the process of sending/receiving data in one bit at a time.

Parallel communication is the process of sending/receiving multiple data bits at a time through parallel channels.

Serial Communication is implemented in ATmega16 using UART(universal asynchronous receiver transmitter) and USART(universal synchronous asynchronous receiver transmitter).

UART stands for Universal Asynchronous Receiver/Transmitter. From the name itself, it is clear that it is asynchronous i.e. the data bits are not synchronized with the clock pulses.

USART stands for Universal Synchronous AsynchronousReceiver/Transmitter. This is of the synchronous type, i.e. the data bits are synchronized with the clock pulses.

USART Pin Configuration

Now lets have a look at the hardware pins related to USART. The USART of the AVR occupies three hardware pins pins:

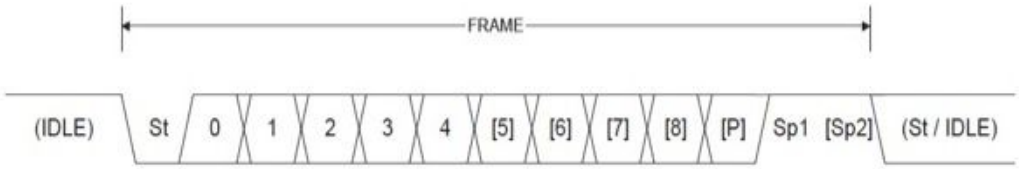
1. **RxD: USART Receiver Pin (ATMega8 Pin 2; ATMega16/32 Pin 14)**
2. **TxD: USART Transmit Pin (ATMega8 Pin 3; ATMega16/32 Pin 15)**
3. **XCK: USART Clock Pin (ATMega8 Pin 6; ATMega16/32 Pin 1)**

Order of Bits

1. **Start bit (Always low)**
2. **Data bits (LSB to MSB) (5-9 bits)**
3. **Parity bit (optional) (Can be odd or even)**
4. **Stop bit (1 or 2) (Always high)**

A frame starts with the start bit followed by the least significant data bit. Then the next data bits, up to a total of nine, are succeeding, ending with the most significant bit. If enabled, the parity bit is inserted after the data bits, before the stop bits. When a

complete frame is transmitted, a new frame can directly follow it, or the communication line can be set to an idle (high) state. Here is the frame format as mentioned in the AVR datasheet-



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high.

Frame Format

Setting the Number of DATA Bits

The data size used by the USART is set by the UCSZ2:0, bits in UCSRC Register. The Receiver and Transmitter use the same setting.

Note: Changing the settings of any of these bits (on the fly) will corrupt all ongoing communication for both the Receiver and Transmitter. Make sure that you configure the same settings for both transmitter and receiver.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Data Bit Settings

Setting Number of STOP Bits

This bit selects the number of stop bits to be inserted by the transmitter. *The Receiver ignores this setting.* The USBS bit is available in the UCSRC Register.

USBS	Stop Bit(s)
0	1-bit
1	2-bit

Stop Bit Settings

Parity Bits

Parity bits always seem to be a confusing part. Parity bits are the simplest methods of error detection. Parity is simply the number of '1' appearing in the binary form of a number. For example, '55' in decimal is 0b00110111, so the parity is 5, which is odd.

Even and Odd Parity

In the above example, we saw that the number has an odd parity. In case of even parity, the parity bit is set to 1, if the number of ones in a given set of bits (not including the parity bit) is odd, making the number of ones in the entire set of bits (including the parity bit) even. If the number of ones in a given set of bits is already even, it is set to a 0. When using odd parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is even, making the number of ones in the entire set of bits (including the parity bit) odd. When the number of set bits is odd, then the odd parity bit is set to 0.

Still confused? Simply remember – even parity results in even number of 1s, whereas odd parity results in odd number of 1s. Lets take another example. 0d167 = 0b10100111. This has five 1s in it. So in case of even parity, we add another 1 to it to make the count rise to six (which is even). In case of odd parity, we simply add a 0 which will stall the count to five (which is odd). This extra bit added is called the parity bit! Check out the following example as well (taken from Wikipedia):

7 bits of data	(count of 1 bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010

1101001	4	11010010	11010011
1111111	7	11111111	11111110

But why use the Parity Bit?

Parity bit is used to detect errors. Lets say we are transmitting 0d167, i.e. 0b10100111. Assuming an even parity bit is added to it, the data being sent becomes 0b101001111 (pink bit is the parity bit). This set of data (9 bits) is being sent wirelessly. Lets assume in the course of transmission, the data gets corrupted, and one of the bits is changed. Ultimately, say, the receiver receives 0b100001111. The blue bit is the error bit and the pink bit is the parity bit. We know that the data is sent according to even parity. Counting the number of 1s in the received data, we get four (excluding even parity bit) and five (including even parity bit). Now doesn't it sound amusing? There should be even number of 1s including the parity bit, right? This makes the receiver realize that the data is corrupted and will eventually discard the data and wait/request for a new frame to be sent. The Parity Generator calculates the parity bit for the serial frame data. When parity bit is enabled (UPM1 = 1), the Transmitter control logic inserts the parity bit between the last data bit and the first stop bit of the frame that is sent. The parity setting bits are available in the UPM1:0 bits in the UCSRC Register.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

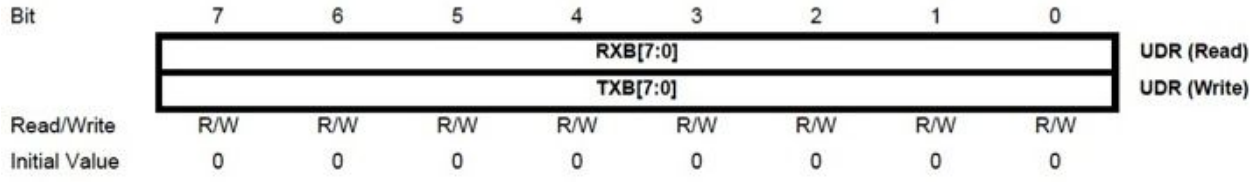
Parity Settings

Although most of the times, we do not require parity bits.

Register Description

Now lets learn about the registers which deal with the USART. We need to program the registers in order to make the peripheral work. The same is the case with USART. The USART of AVR has five registers, namely UDR, UCSRA, UCSRB, UCSRC and UBBR. We have already discussed about UBBR earlier in this post, but we will have another look.

UDR: USART Data Register (16-bit)



UDR – UART Data Register

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

Following this, basic coding on 8-bit transferring was done and run on the microcontroller via USART system.

Towards the end of the day, the final coding for gesture recognition had begun and finished by the next morning, when the motors responded perfectly to the code.

DAY-7

Final Code running on bot is achieved.

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

```
#include <math.h>
#define ob_x 345
#define ob_y 353
#define ob_z 351
#define EN_1 4
#define EN_2 5
#define PC_0 0
#define PC_1 1
#define PC_2 2
#define PC_3 3
#include "LCD.h"
#include "LCD.c"


int angle(int x,int y,int z)
{
	float a;
	int b;
	a=(x)/(sqrt(y*y+z*z)) ;
	a=atan(a);
	a*=180;
	a/=M_PI;
	b=a;
	return b;
}
void ADC_init()
{
	ADCSRA=ADCSRA|(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}
uint16_t ADC_output(int x)
{
	uint16_t a;
	ADMUX=0;
	ADMUX|=(1<<REFS0);
	ADMUX+=x;
	ADCSRA|=(1<<ADSC);
```

```
        while((ADCSRA&(1<<ADSC))!=0);
        a=ADC;
        return a;
}

        int main()
        {
                int x,y,z,theta,phi;
ADC_init();
DDRA=0;
DDRB=255;
DDRC=0x07;
DDRD|=(1<<EN_1)|(1<<EN_2);
DDRC|=(1<<PC_0)|(1<<PC_1)|(1<<PC_2)|(1<<PC_3);
TCCR1A|=(1<<COM1A1)|(1<<COM1B1)|(1<<WGM10);
TCCR1B|=(1<<WGM12)|(1<<CS10)|(1<<CS11);
PORTD|=(1<<EN_1)|(1<<EN_2);


init_LCD();
print_string("-----Robotics :D-----");
while(1){
        x=ADC_output(0);
        y=ADC_output(1);
        z=ADC_output(2);
        x-=ob_x;
        y-=ob_y;
        z-=ob_z;
        theta=angle(x,y,z);
        phi=angle(y,x,z);
        if((theta>=-15)&&(theta<=15)) theta=0;
        else {if(theta<0) theta+=15;else theta-=15;}
                if((phi>=-15)&&(phi<=15)) phi=0;
                else {if(phi<0) phi+=15;else phi-=15;}
        if((theta>0)&&(phi==0))         {PORTC=0b00000110;          OCR1A=(theta*255)/75;
OCR1B=(theta*255)/75;}
```

```
        else                                                    if((theta<0)&&(phi==0))
{PORTC=0b00001001;OCR1A=((0-theta)*255)/75;OCR1B=((0-theta)*255)/75;}
        if((theta==0)&&(phi>=0)) {PORTC=0b00000101; OCR1A=(phi*255)/75; OCR1B=(phi*255)/75;}
        else                                                    if((theta==0)&&(phi<=0)){
PORTC=0b00001010;OCR1A=((0-phi)*255)/75;OCR1B=((0-phi)*255)/75;}
        if((theta>0)&&(phi>0))    {PORTC=0b00000110;    OCR1A=((theta+phi)*255)/150;if(theta>phi)
OCR1B=((theta-phi)*255)/150;else OCR1B=((phi-theta)*255)/150;}
        if((theta>0)&&(phi<0))
{PORTC=0b00000110;OCR1B=((theta+phi)*255)/150;phi=(-1)*phi;if(theta>phi)OCR1B=((theta-phi)*2
55)/150;else OCR1B=((phi-theta)*255)/150;}
        if((theta<0)&&(phi>0))                          {PORTC=0b00001001;theta=(-1)*theta;
OCR1A=((theta+phi)*255)/150;if(theta>phi)                   OCR1B=((theta-phi)*255)/150;else
OCR1B=((phi-theta)*255)/150;}
        if((theta<0)&&(phi<0))                          {PORTC=0b00001001;theta=(-1)*theta;
OCR1B=((theta+phi)*255)/150;phi=(-1)*phi     ;if(theta>phi)     OCR1B=((theta-phi)*255)/150;else
OCR1B=((phi-theta)*255)/150;}
        }
        return 0;
        }
```