WINTER WORKSHOP DOCUMENTATION

DAY-1

PROBLEM STATEMENT: WIRELESS GESTURE RECOGNITION WITH OBSTACLE AVOIDANCE

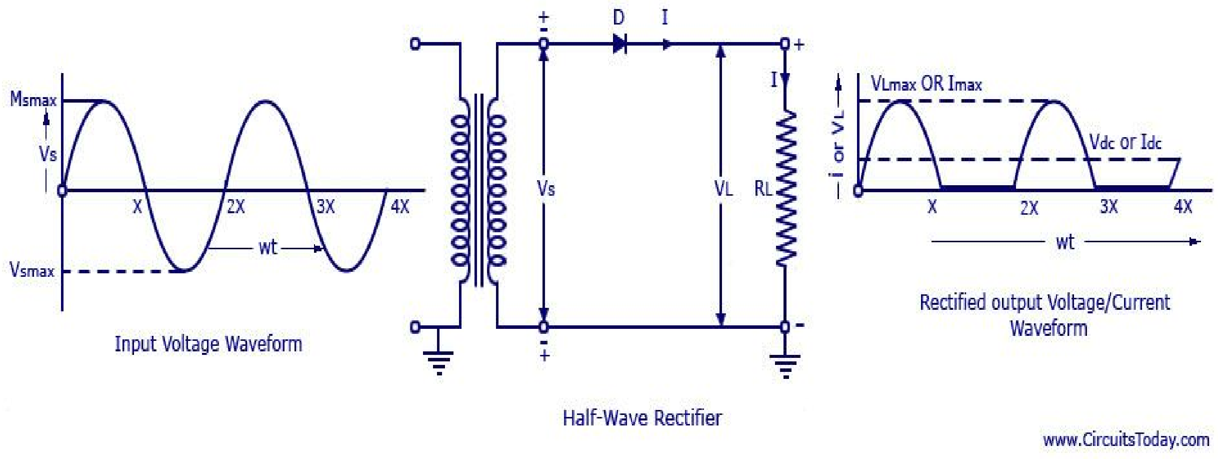
Basic Analogies of a Robot

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

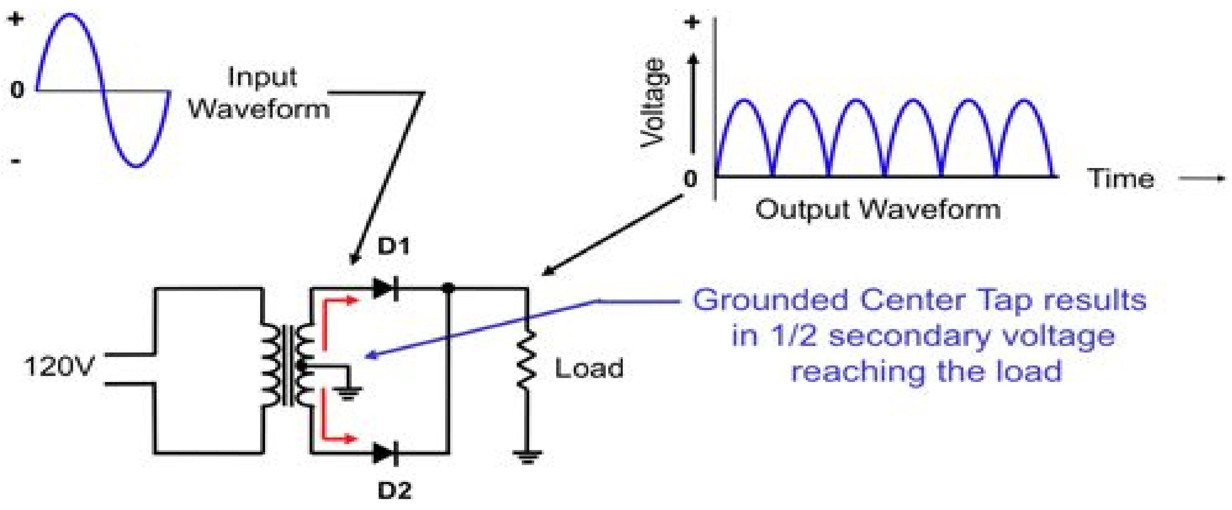
Power Supply

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:

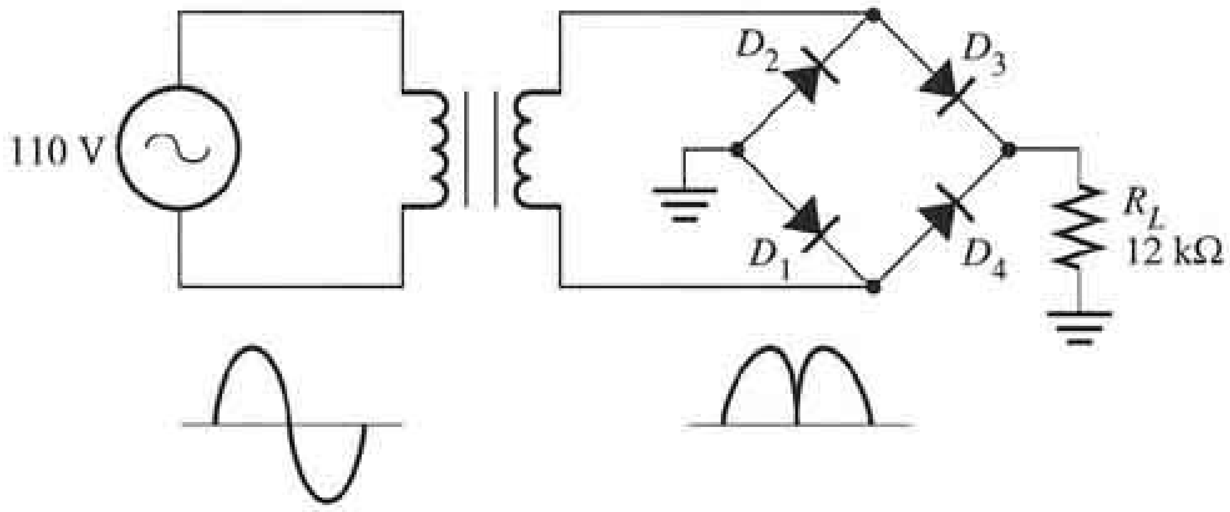
a) Half-wave rectifiers: This rectifier converts only one half of the input AC supply, and leaves the other half.



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


+
0
-
Input Waveform
Voltage
+
0
Time
Output Waveform
D1
Grounded Center Tap results in 1/2 secondary voltage reaching the load
120V
Load
D2

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



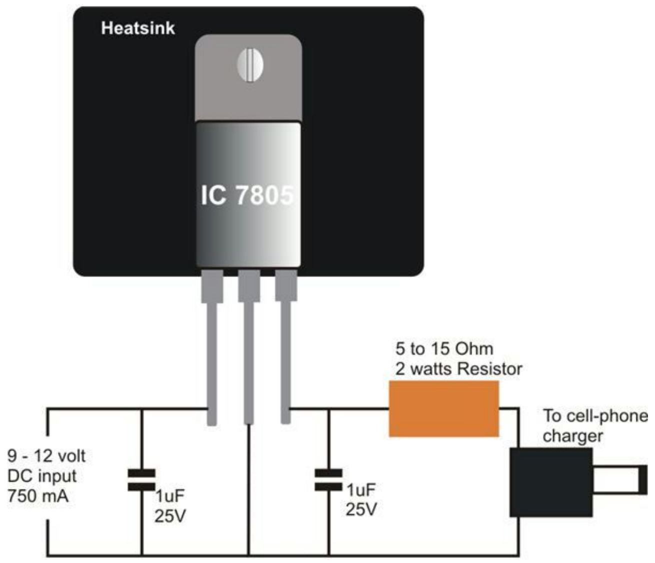
**The RMS value of voltage is the value for DC supply after rectification.
**Adapters are used to power the robots. Batteries are not used as they are unreliable.

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Analog Circuits are outdated.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.

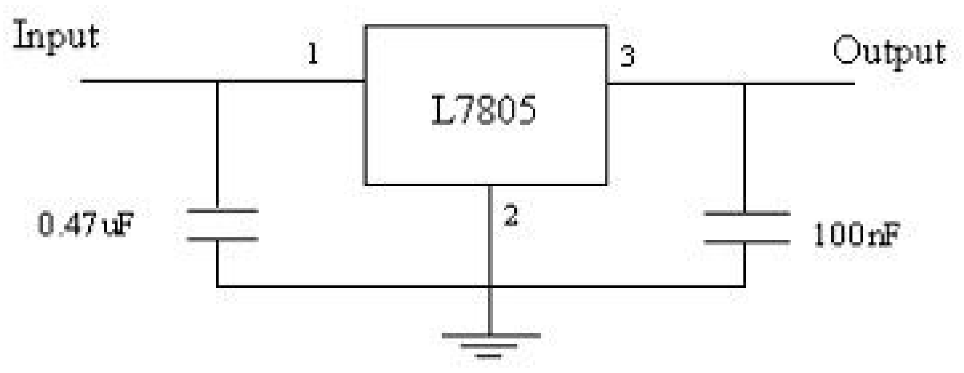
Voltage limit for most of robot circuits is around 5V usually.
The adapters used for running the robots are of the range 9/12 V. To bring down the voltage to usable form and limit, Integrated Circuits(ICs) are used. General ICs used are LM78XX

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.

**Through-hole ICs are used in robotics. Tracer ICs are used in PCBs.

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

1. **AND**
2. **OR**
3. **NOT**
4. **XOR**
5. **<< (LEFT SHIFT)**
6. **>>(RIGHT SHIFT)**

THESE OPERATORS ARE VERY USEFUL IN BUILDING THE PROGRAMME .

OPERATIONS ON A BYTE

- **SET**

WHEN 'SET' IS OPERATED ON A BIT IT CHANGES ITS VALUE TO 1
THE SYNTAX IS A|=(1<<i);

- **CLEAR**

WHEN 'CLEAR' IS OPERATED ON A BIT IT CHANGES ITS VALUE TO 0
THE SYNTAX IS A&=~(1<<i);

- **TOGGLE**

WHEN TOGGLE IS OPERATED ON BIT IT CHANGES ITS VALUE TO 0 IF INITIALLY IT IS 1 , AND TO 1 IF INITIALLY IT IS 0 .
THE SYNTAX IS A^=(1<<i);

PINS AND PORTS

THERE ARE A TOTAL OF 40 PINS WHICH INCLUDE 4 PORTS AS SHOWN BELOW

REGISTERS

- **DDRX**

THIS FUNCTION SPECIFIES THE INPUT AND OUTPUT PINS OF THE PORT HERE X STANDS FOR PORT NAME
e.g DDRA=0b00000000;   THIS MEANS ALL THE PINS ARE INPUT PINS
DDRA=0b11111111;   THIS MEANS ALL THE PINS ARE OUTPUT PINS

- **PINX**

IT IS USED TO SPECIFY THE STATE OF THE PORT

- **PORTX**

IT ASSIGNS OUTPUT VALUE TO THE OUTPUT PINS

DELAY COMMAND

IT APPEARS LIKE _delay_ms(500);
THIS COMMAND CREATES TIME LAG . IT CAN USED BETWEEN TWO COMMANDS IN ORDER TO CREATE TIME LAG BETWEEN TWO COMMANDS .

```
PORTA=0b11111111;
_delay_ms(500);
PORTA=0b00000000:
```

HEADER FILES

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

WE USE THESE KNOWLEDGE TO MAKE A PROGRAMME THAT WOULD LIT UP SOME LED PATTERNS .

ADC

An analog-to-digital converter (ADC)is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude.
ADC value is the corresponding digital value of a given analog input .

REGISTERS

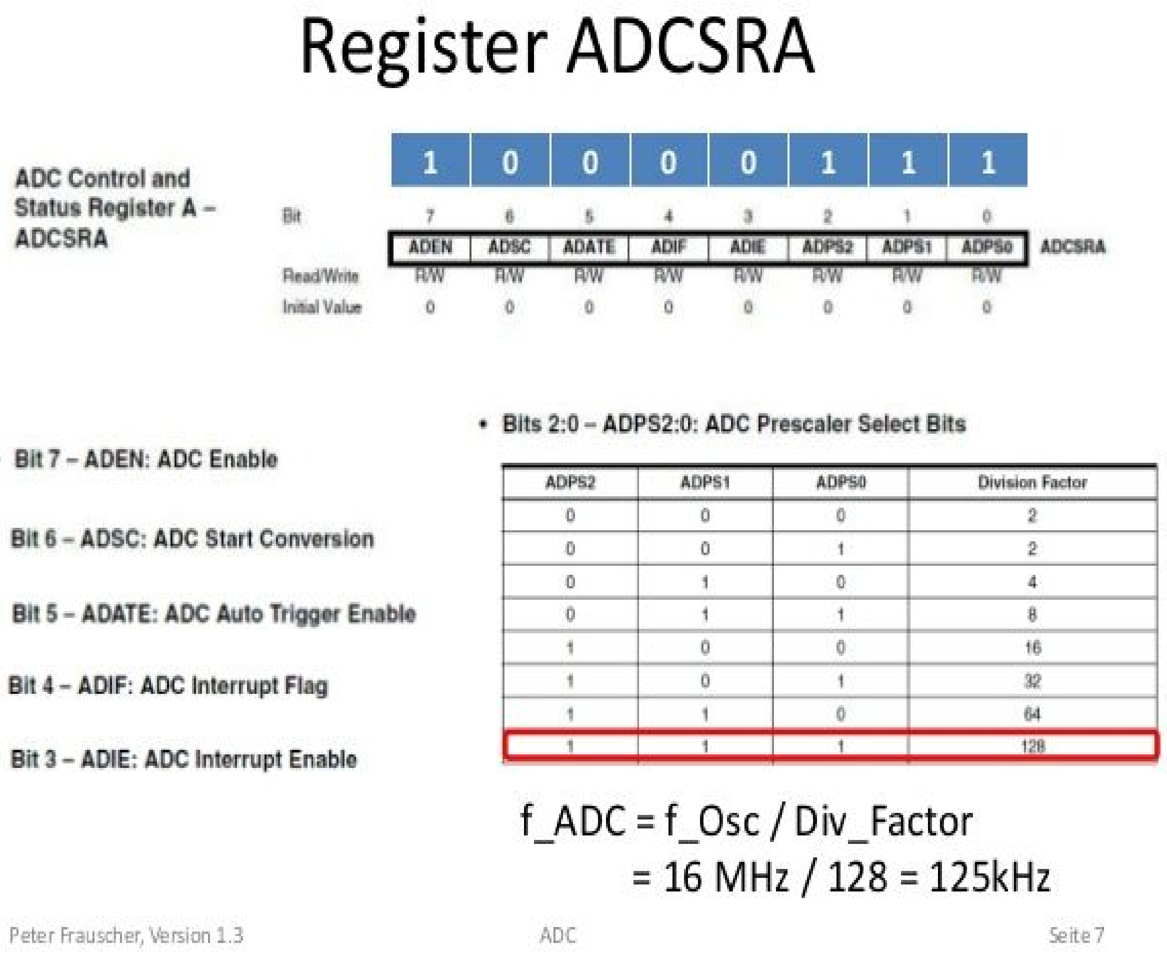
1. ADCSRA

ADEN: **ADC Enable bit . When this bit is set to 1 ADC turns on.**

ADSC: **ADC Start Conversion . When this bit is set to 1 it begins ADC conversion, and the value is set back to 0 when the conversion is complete .**

ADPS: **ADC Prescaler . These bits are set the ADC clock frequency.**

ADC clock frequency = XTAL frequency / Prescaler



Register ADCSRA

ADC Control and Status Register A – ADCSRA

	1	0	0	0	0	1	1	1	
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	

- Bit 7 – ADEN: ADC Enable
- Bit 6 – ADSC: ADC Start Conversion
- Bit 5 – ADATE: ADC Auto Trigger Enable
- Bit 4 – ADIF: ADC Interrupt Flag
- Bit 3 – ADIE: ADC Interrupt Enable

- Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits

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

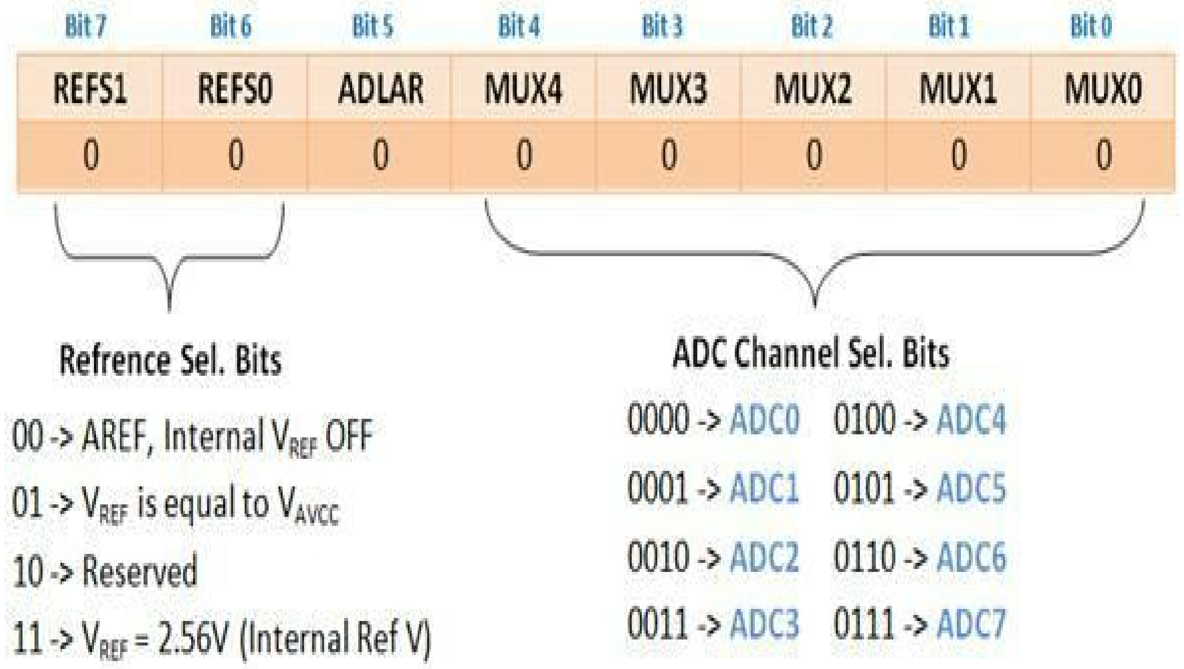
f_ADC = f_Osc / Div_Factor
= 16 MHz / 128 = 125kHz

Peter Frauscher, Version 1.3 ADC Seite 7

2. ADMUX

<u>REFS1 and REFS0</u> bits determine the source of reference voltage whether it is internal or the external voltage source connected to AREF pin.

<u>MUX</u> bits are used to select between the channels which will provide data to ADC for conversion.



ADLAR bit when set to 1 gives the left adjusted result in data registers ADCH and ADCL. These help to get the required precision in the output.

ADLAR

The ADC Data Register – ADCL and ADCH

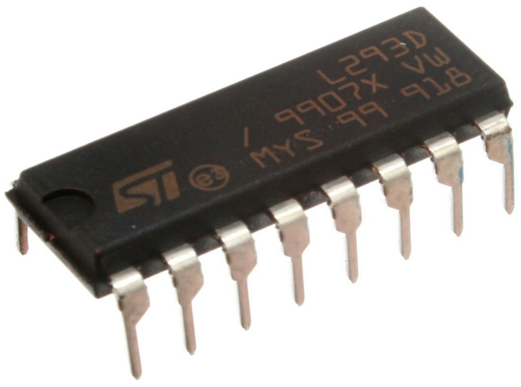
ADLAR = 0

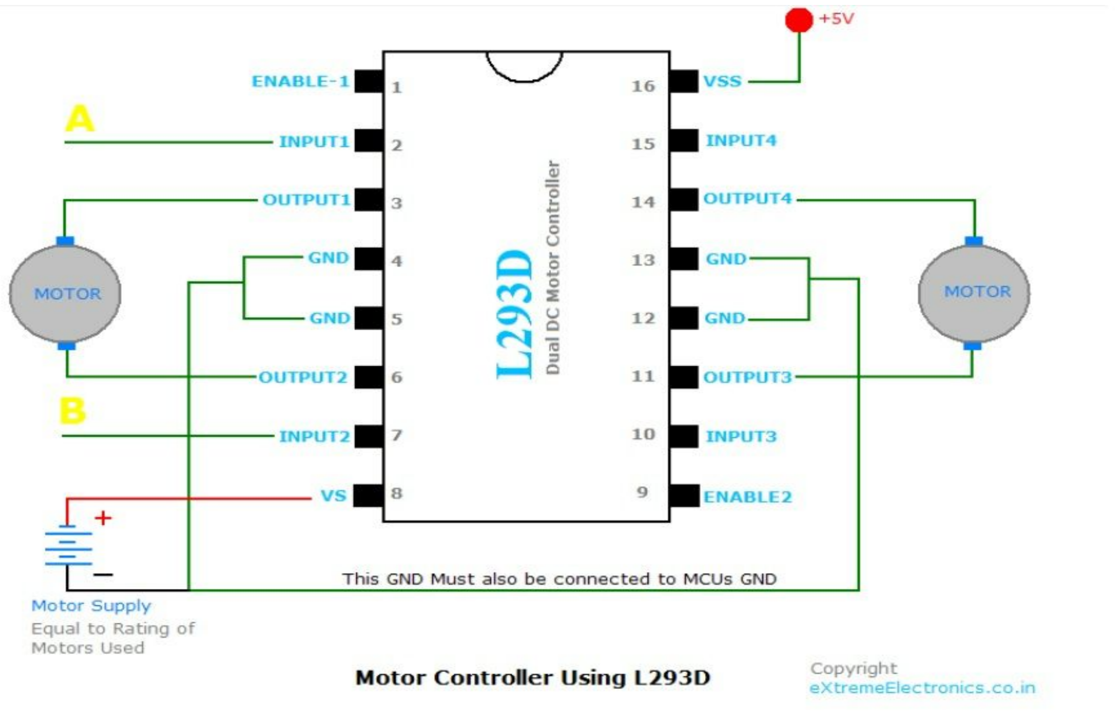
Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

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	

MOTOR DRIVER L293D

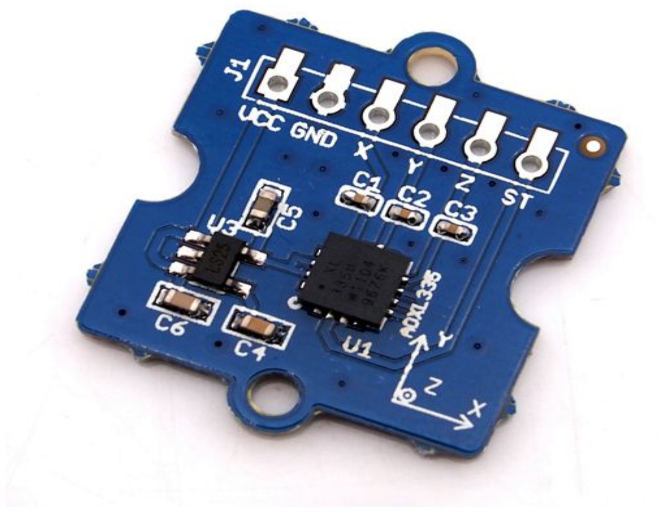


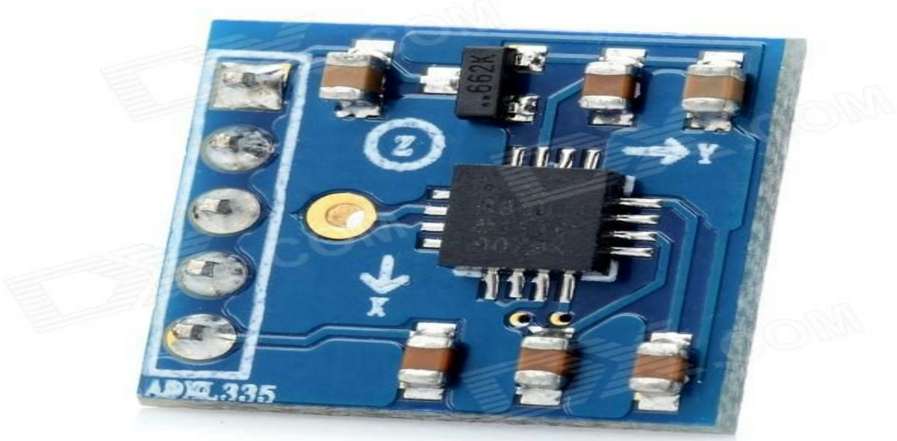


Motor Controller Using L293D

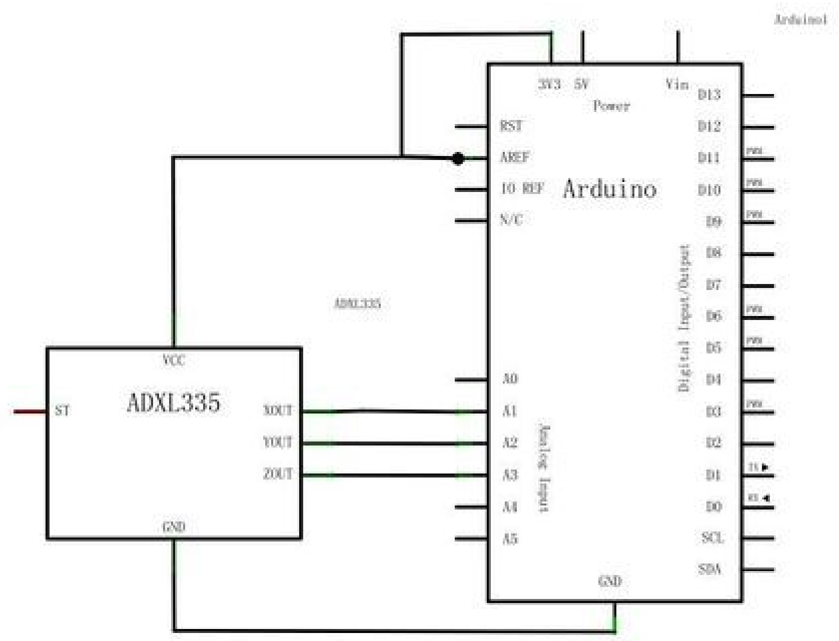
ENABLE 1 AND ENABLE 2 ARE USED TO ON OR OFF THE LEFT AND LIGHT CIRCUIT . ACCORDING THE ENABLE VALUE OUTPUT FOLLOW ENABLE CURVE.

ACCELEROMETER





IT’S GIVE ANALOG OUTPUT VOLTAGE ACCORDING TO ITS ANGLE WITH HORIZONTAL .
IT WORKS ON GRAVITATIONAL FORCE IN 3 DIMENSIONAL AXIS.

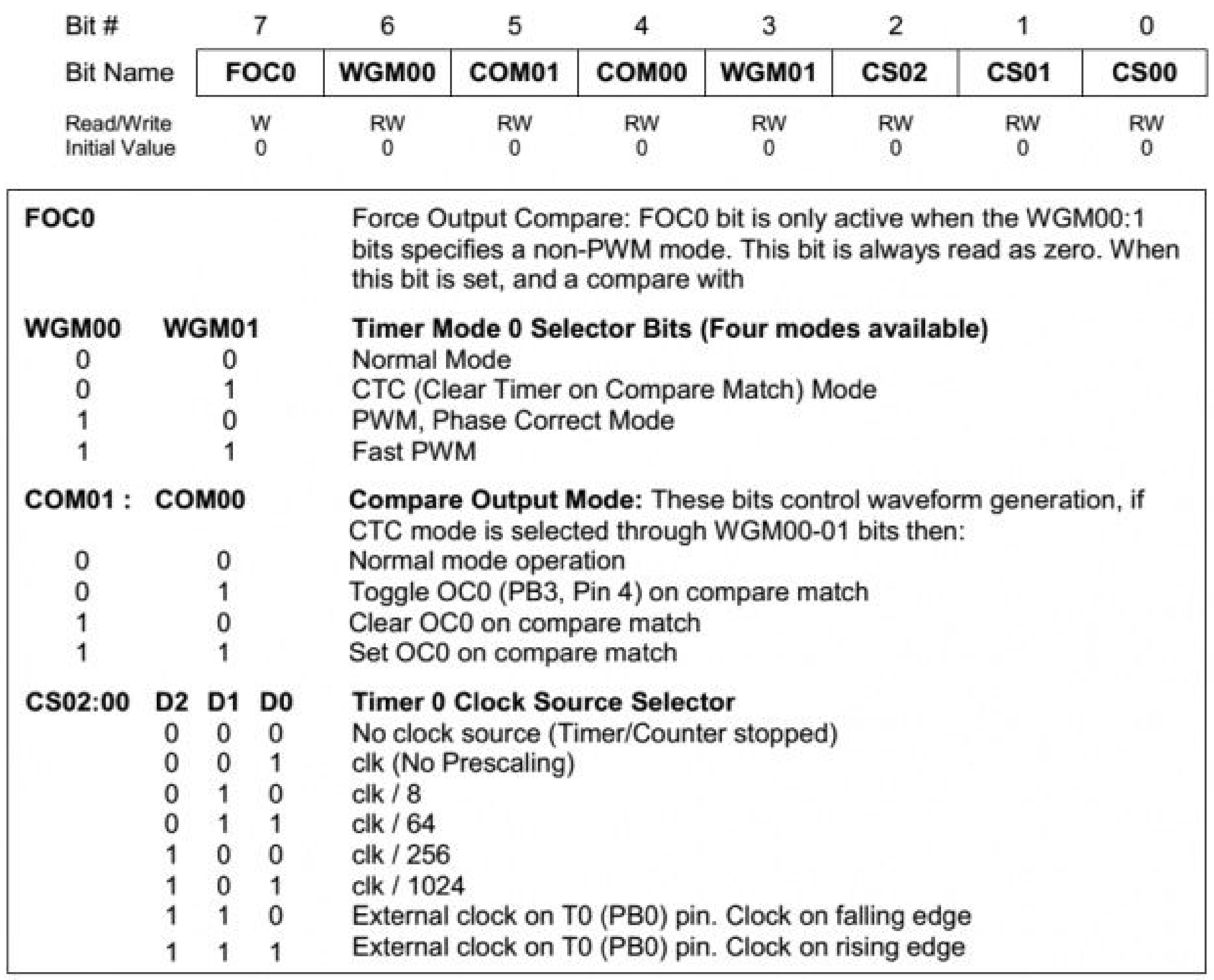


Day 3

Timers:-

3 Timers are there ,2 8 bit timer TCCR0 & TCCR2,1 16bit Timer TCCR1.

TCCR0 =>

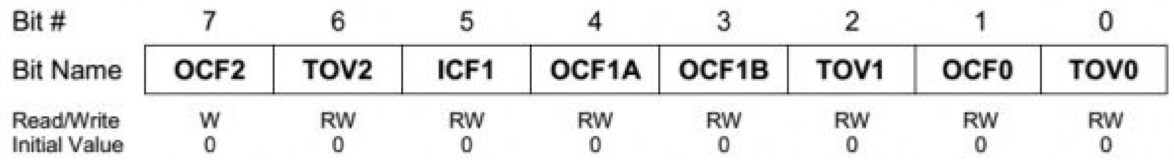


Bit #	7	6	5	4	3	2	1	0
Bit Name	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Read/Write	W	RW	RW	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0

FOC0			Force Output Compare: FOC0 bit is only active when the WGM00:1 bits specifies a non-PWM mode. This bit is always read as zero. When this bit is set, and a compare with
WGM00	**WGM01**		**Timer Mode 0 Selector Bits (Four modes available)**
0	0		Normal Mode
0	1		CTC (Clear Timer on Compare Match) Mode
1	0		PWM, Phase Correct Mode
1	1		Fast PWM
COM01 :	**COM00**		**Compare Output Mode:** These bits control waveform generation, if CTC mode is selected through WGM00-01 bits then:
0	0		Normal mode operation
0	1		Toggle OC0 (PB3, Pin 4) on compare match
1	0		Clear OC0 on compare match
1	1		Set OC0 on compare match

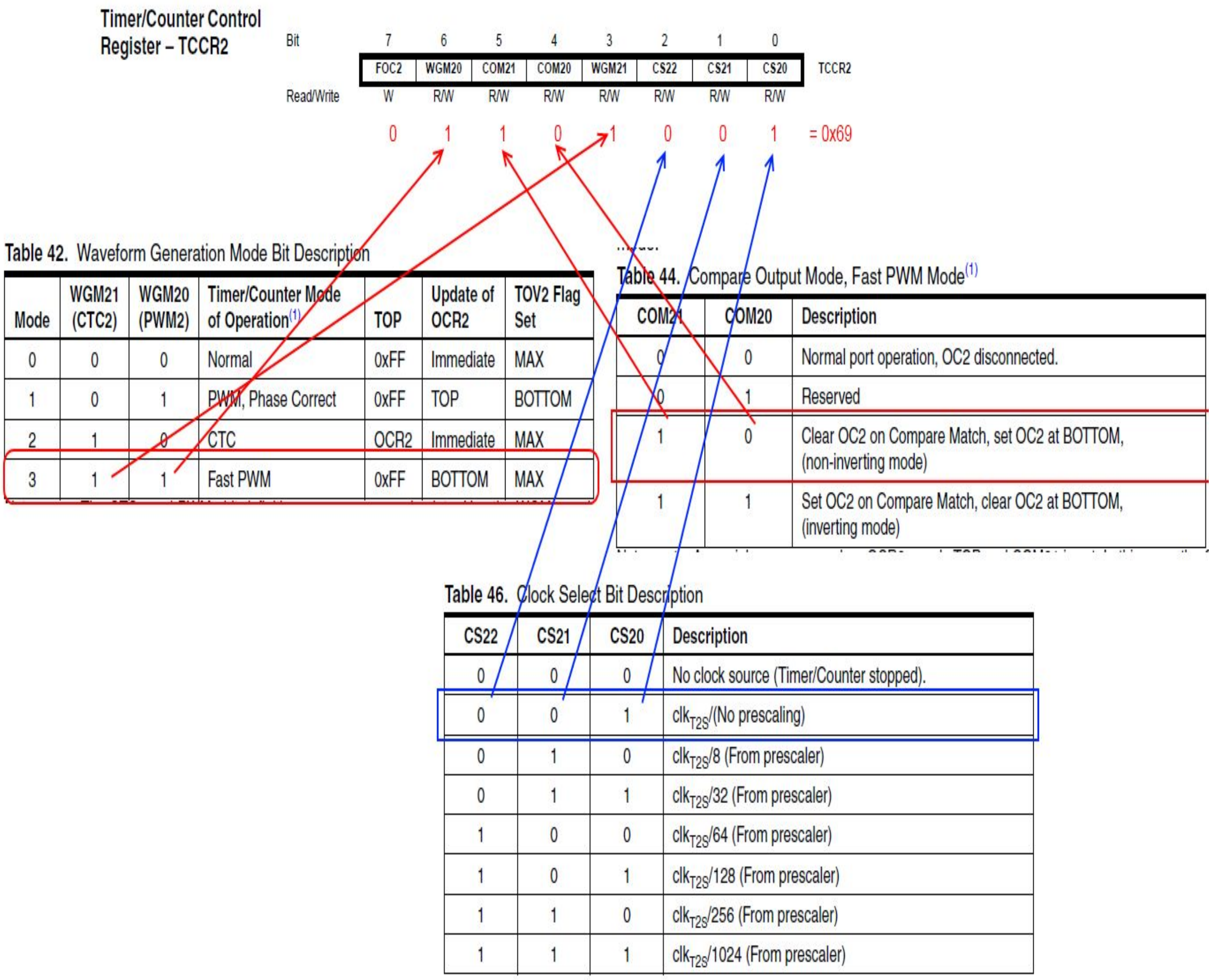
CS02:00	D2	D1	D0	
				Timer 0 Clock Source Selector
	0	0	0	No clock source (Timer/Counter stopped)
	0	0	1	clk (No Prescaling)
	0	1	0	clk / 8
	0	1	1	clk / 64
	1	0	0	clk / 256
	1	0	1	clk / 1024
	1	1	0	External clock on T0 (PB0) pin. Clock on falling edge
	1	1	1	External clock on T0 (PB0) pin. Clock on rising edge

TCCR0 (Timer / Counter Control Register)



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

TCCR2 =>



Timer/Counter Control Register – TCCR2

Bit	7	6	5	4	3	2	1	0	
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	TCCR2
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	0	1	1	0	1	0	0	1	= 0x69

Table 42. Waveform Generation Mode Bit Description

Mode	WGM21 (CTC2)	WGM20 (PWM2)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR2	TOV2 Flag Set
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR2	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

Table 44. Compare Output Mode, Fast PWM Mode[1]

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Reserved
1	0	Clear OC2 on Compare Match, set OC2 at BOTTOM, (non-inverting mode)
1	1	Set OC2 on Compare Match, clear OC2 at BOTTOM, (inverting mode)

Table 46. Clock Select Bit Description

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk_{T2S}/(No prescaling)
0	1	0	clk_{T2S}/8 (From prescaler)
0	1	1	clk_{T2S}/32 (From prescaler)
1	0	0	clk_{T2S}/64 (From prescaler)
1	0	1	clk_{T2S}/128 (From prescaler)
1	1	0	clk_{T2S}/256 (From prescaler)
1	1	1	clk_{T2S}/1024 (From prescaler)

TCCR1 =>

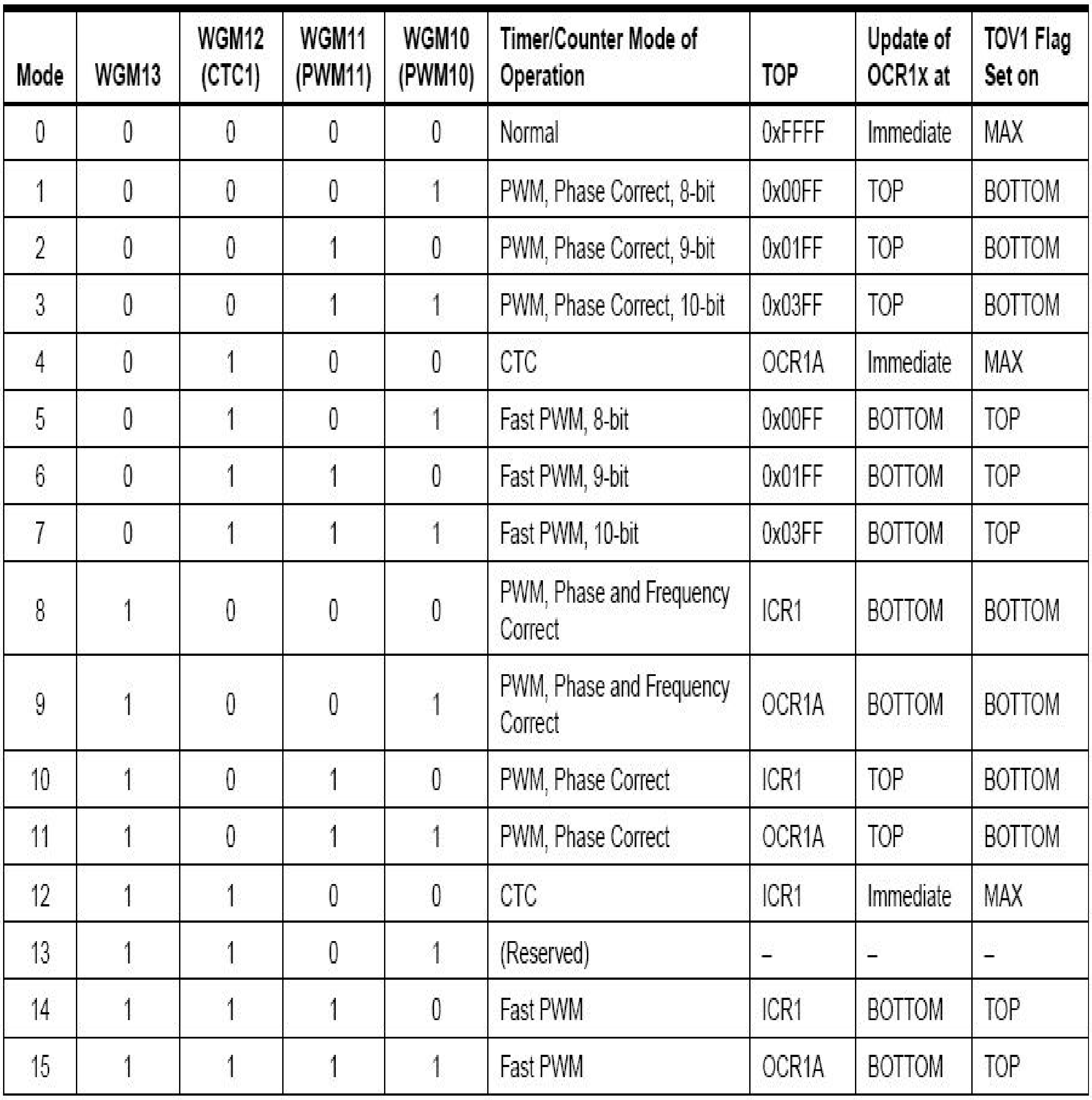
TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter1 Control Register B

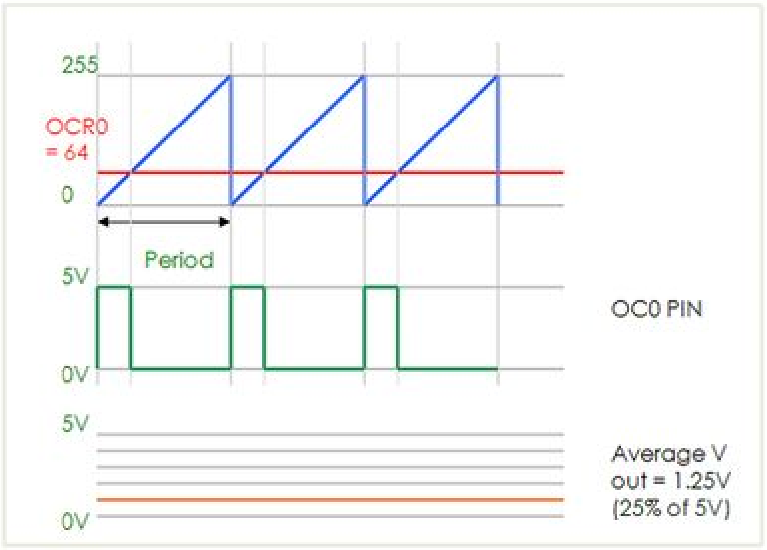
Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on compare match
1	0	Clear OC1A/OC1B on compare match (Set output to low level)
1	1	Set OC1A/OC1B on compare match (Set output to high level)

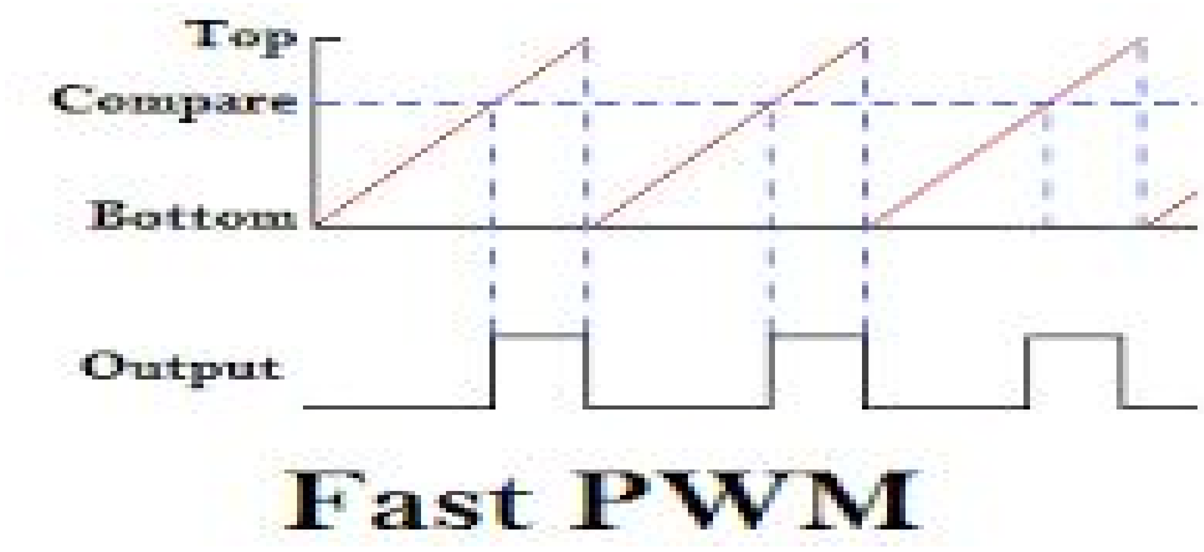


Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1X at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

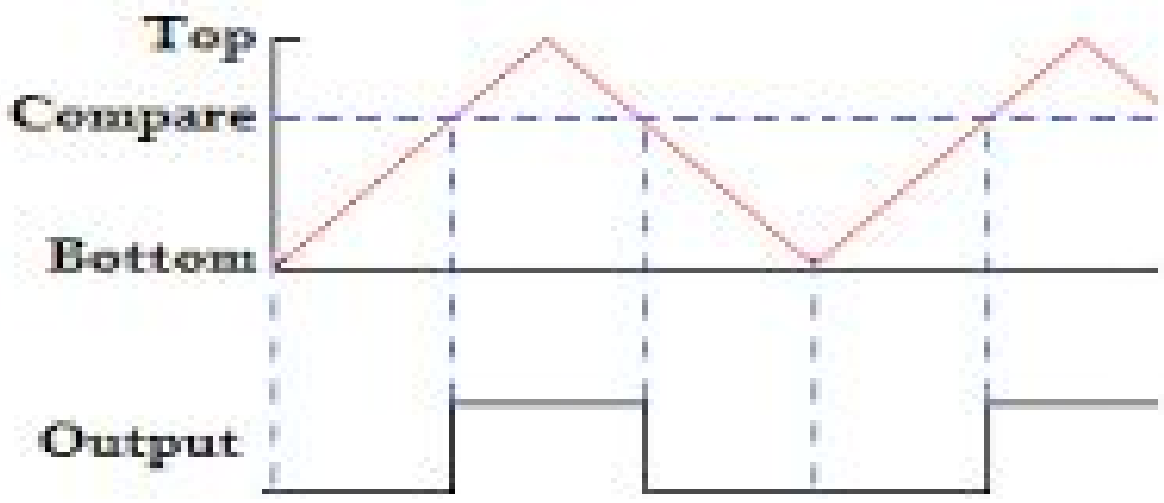
Types of PWM =>
 1)Normal PWM-



2)Fast PWM-



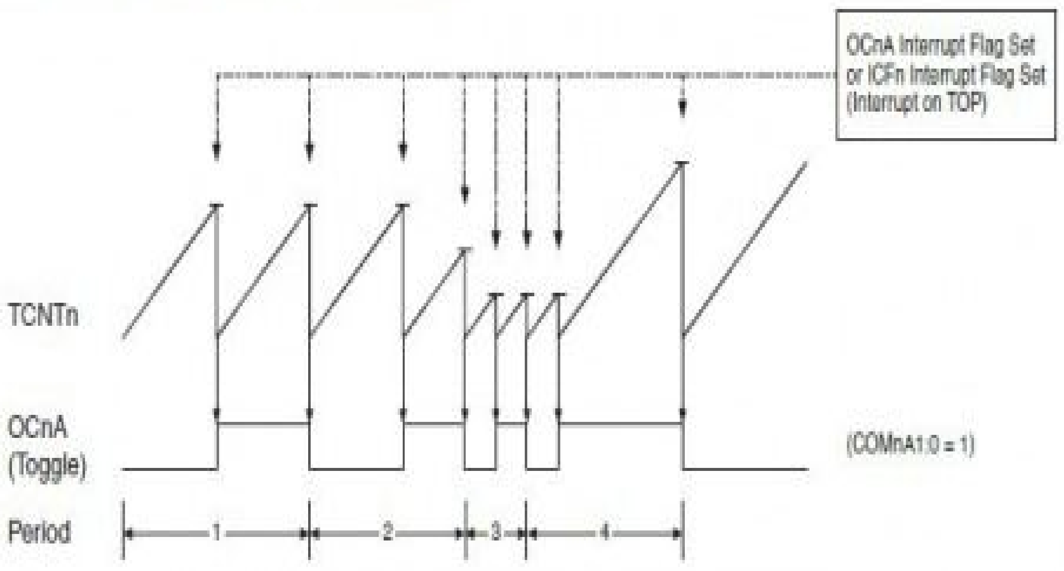
3)Phase Correct PWM-



Phase Correct PWM

4)Clear Time On Compare Match Mode(CTC)-

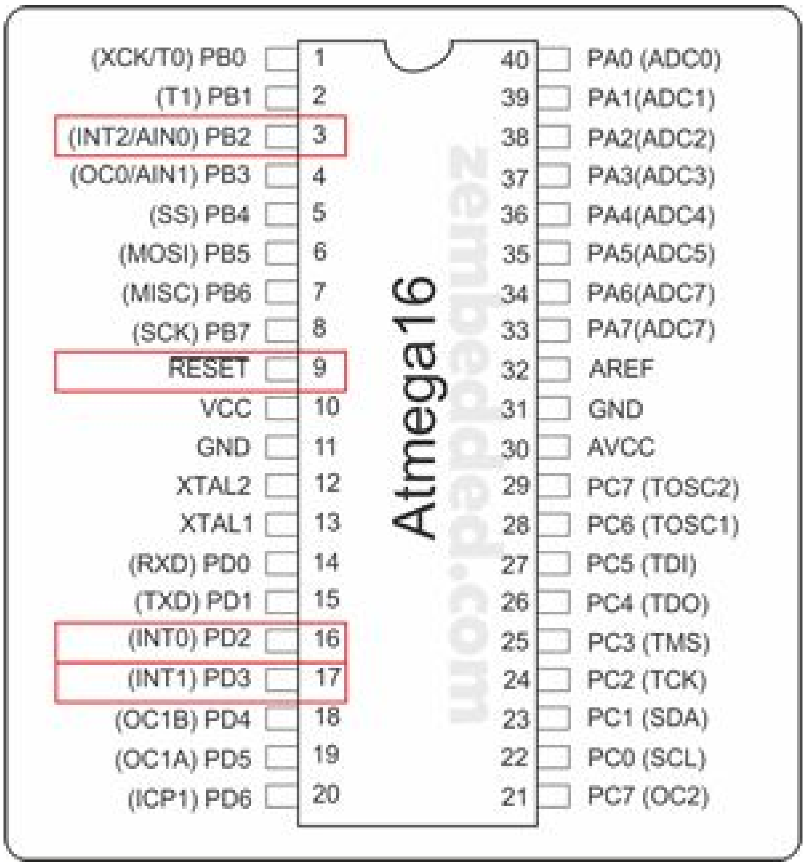
Figure 16-6. CTC Mode, Timing Diagram


OCnA Interrupt Flag Set or ICFn Interrupt Flag Set (Interrupt on TOP)
TCNTn
OCnA (Toggle)
(COMnA1:0 = 1)
Period
1
2
3
4

Day 4

External Interrupt :-

Hi today we read about the INTERRUPT, in our practical life this word causes an annoying situation. Nobody wanted to get interrupted, but in controllers this offers a great deal of flexibility. One of the most useful principles of modern embedded processors is interrupt. Interrupts does exactly what it means. Normally, you would expect a program to keep on executing sequentially in the way you have defined. But when an interrupt occurs, the normal flow of instruction is suspended by the microcontroller and the interrupt service routine (ISR) of the according event is executed. So basically when ever interrupt event occur, we stop current task, handle the event then resume back where we left off.



What exactly happens when interrupt happens?

Following is what happens when an interrupt occurs:

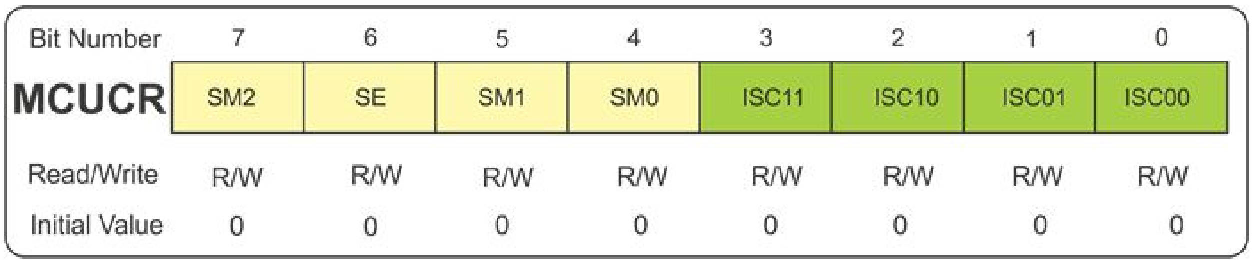
1. Controller completes the instruction which is being executed.
2. Control gets transfers to Interrupt Service Routine (ISR). (Each interrupt have an associated ISR which is a piece of code which tells the microcontroller what to do when an interrupt has occurred.)

3. Execution of ISR is performed by loading the beginning address of the corresponding ISR into program counter.
4. Execution of ISR continues until the return from the interrupt instruction (RETI) is encountered.
5. When ISR is complete, the Controller resumes processing where it left off before the interrupt occurred.

There are two mainly two sources of interrupts available, but 8-bit AVR lacks software interrupts so we fall back to Hardware interrupts.
Hardware interrupt which occurs in response to a timer reaching to predefined value or a particular pin changing its state.
To check number of interrupts available for a particular AVR microcontroller, please go through datasheets. You will find a chapter called interrupts, there you will find a reset and interrupt vector table. This table contains full list of possible available interrupts. This table contains Vector table address, source name and interrupts definition.
The following example will show how to use external interrupts (as opposed to timer interrupts) on an Atmel AVR using software written in GCC. The first thing to do is to see what interrupts are available for your model of processor. For this example I will be using the ATmega16.
You can use the data sheet available from Atmel to find the interrupts.
The ATMEGA16 has 3 external interrupt lines: INTO, INT1 and INT2, on pins PD2, PD3 and PB2.
Interrupts on **INT0** and **INT1** can be level-triggered (meaning that the interrupt is triggered when the signal goes low i.e. 0V for some time or edge-triggered(meaning that the interrupt is triggered when the signal changes from high to low or low to high)
INT2 can only be used as an edge-triggered interrupt.

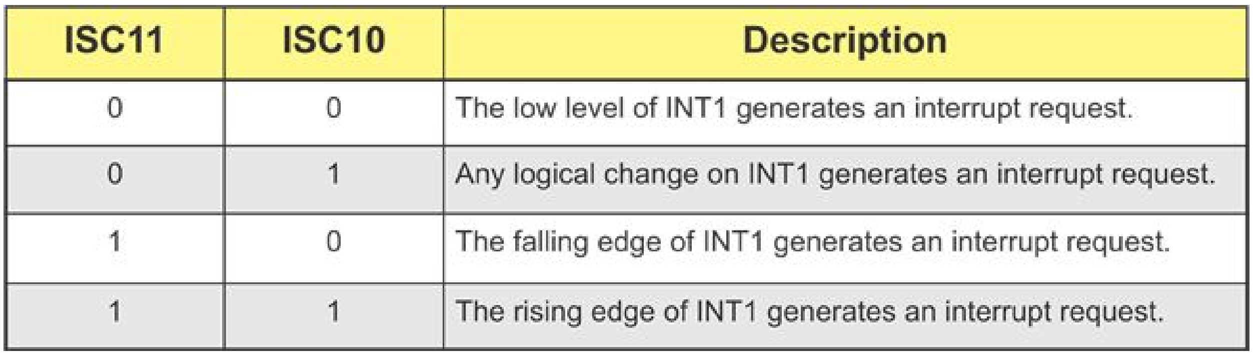
How to set interrupt control register?

MCUCR – MCU Control Register:
The MCU Control Register contains control bits for interrupt sense control and general MCU functions. The Bit0, Bit1, Bit2 and Bit3 of MCUCR register determines the nature of signal at which the interrupt 0 (INT0) and interrupt 1 (INT1) should occur.



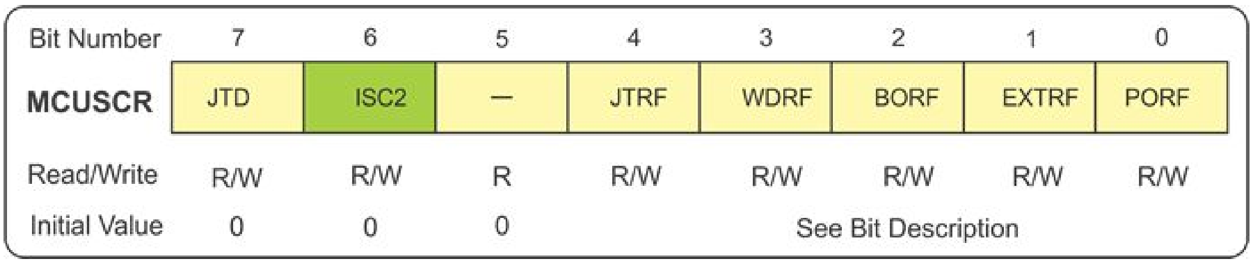
Bit Number	7	6	5	4	3	2	1	0
MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

MCUCR register Bits 3,2 are used for sensing an external interrupt on line INT1. Similarly Bits 1,0 are used for sensing an external interrupt on line INT0. Following table shows different patterns to set the bits.
Interrupt 1 Sense Control



ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

MCUCSR – MCU Control & Status Register:
The MCU Control and Status Register provide information on which reset source caused an MCU Reset. The Bit6 of MCUCSR register determines the nature of signal at which the external interrupt 2 (INT2) should occur. INT2 is edge triggered only, it cannot be used for level triggering like INT0 and INT1.



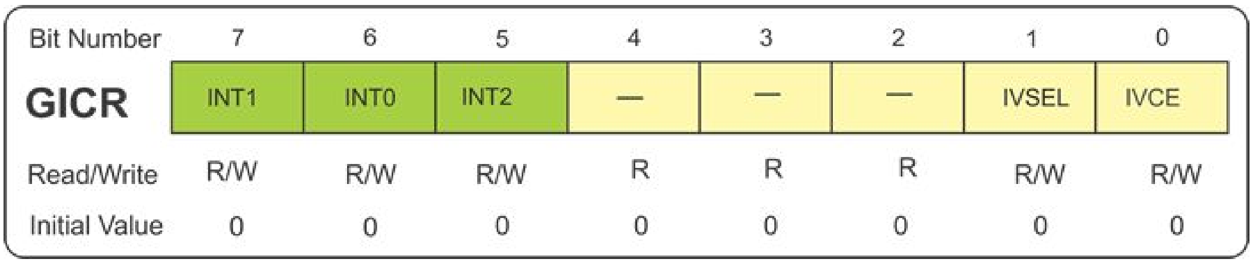
Bit Number	7	6	5	4	3	2	1	0
MCUSCR	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	See Bit Description				

INT2 = 0 Falling Edge
INT2 = 1 Rising Edge
To avoid occurrence of interrupt while changing the ISC2 bit, please follow following procedure:

- Disable INT2 by clearing its interrupt in GICR register.
- The ISC2 bit can be changed

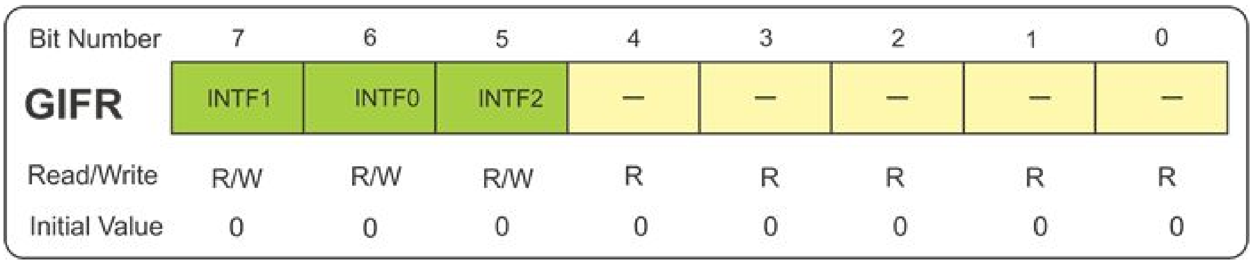
- The INT2 interrupt Flag should be cleared by writing one to flag bit (INTF2) in the GIFR register before the interrupt is re-enabled.

GICR – General Interrupt Control Register:
The General Interrupt Control Register controls the placement of the Interrupt Vector table.
When bit value of Bit5, Bit6 and Bit7 are set to one, enables INT0, INT1 and INT2 interrupts. To disable or mask them just set it to zero.



Bit Number	7	6	5	4	3	2	1	0
GICR	INT1	INT0	INT2	—	—	—	IVSEL	IVCE
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

GIFR – General Interrupt Flag Register:
Interrupt requests are managed by Bit 7, 6, 5. The flag is set for each of these when the respective line is triggered, and cleared when the corresponding interrupt service routine is executed. Alternatively, we can clear the flags manually by writing it to 1.



Bit Number	7	6	5	4	3	2	1	0
GIFR	INTF1	INTF0	INTF2	—	—	—	—	—
Read/Write	R/W	R/W	R/W	R	R	R	R	R
Initial Value	0	0	0	0	0	0	0	0

Programming Steps:

For programming an interrupt, the following steps must be followed:
1. Clear Global Interrupt enable bit in SREG register.
2. Initialize the interrupt by appropriately configuring the MCUCR, MCUCSR and GICR registers.
3. Set Global Interrupt Enable bit in SREG register.

4. Define the appropriate Interrupt service routine (ISR) for the interrupt.

Timer Interrupt:-

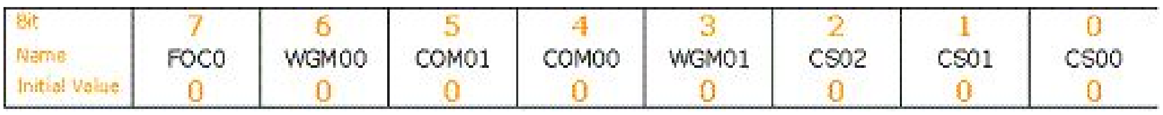
Using The 8 BIT Timer (TIMER0)

The ATmega16 has three different timers of which the simplest is TIMER0. Its resolution is 8 BIT i.e. it can count from 0 to 255.

TIMER0 Registers

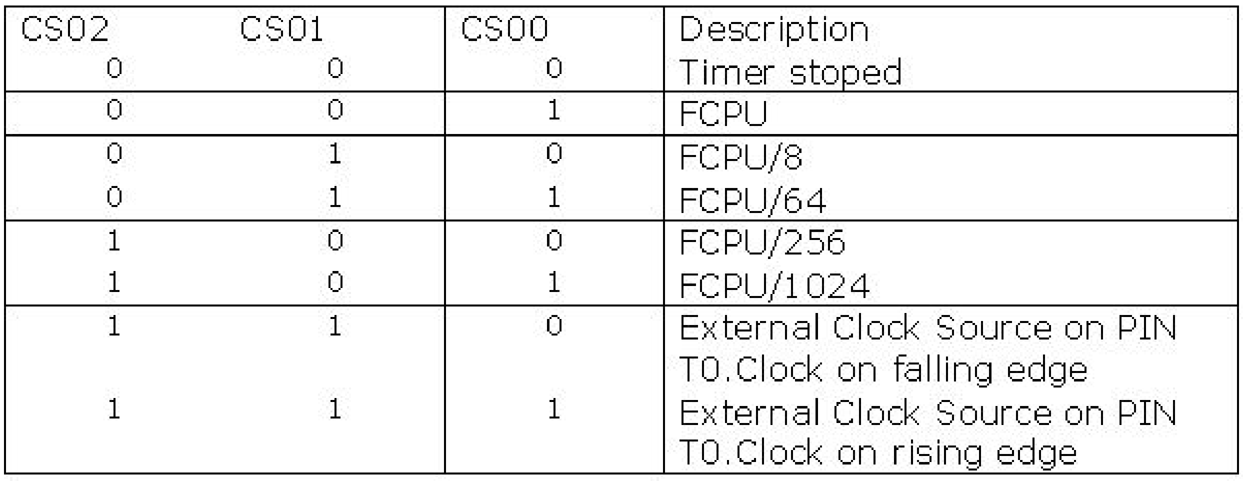
As you may be knowing from the article “Internal Peripherals of AVRs” every peripheral is connected with CPU from a set of registers used to communicate with it. The registers of TIMERs are given below.

TCCR0 – Timer Counter Control Register. This will be used to configure the timer.



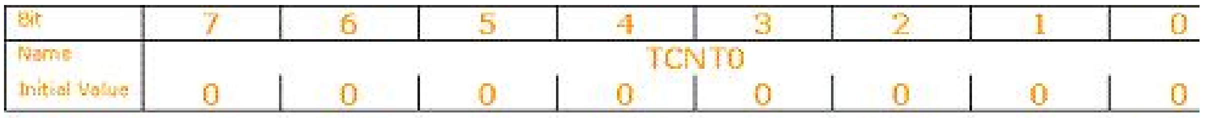
Bit	7	6	5	4	3	2	1	0
Name	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Initial Value	0	0	0	0	0	0	0	0

As we can see there are 8 Bits in this register each used for certain purpose.We will only focus on the last three bits CS02 CS01 CS00 They are the CLOCK SELECT bits. They are used to set up the Prescaler for timer.



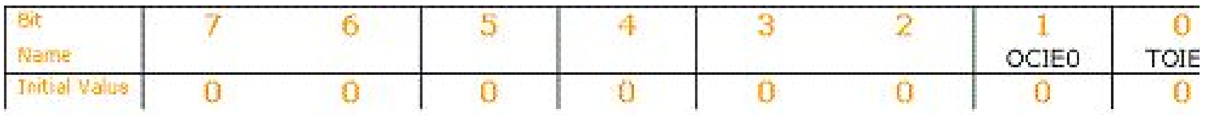
CS02	CS01	CS00	Description
0	0	0	Timer stoped
0	0	1	FCPU
0	1	0	FCPU/8
0	1	1	FCPU/64
1	0	0	FCPU/256
1	0	1	FCPU/1024
1	1	0	External Clock Source on PIN T0.Clock on falling edge
1	1	1	External Clock Source on PIN T0.Clock on rising edge

TCNT0 – Timer Counter 0



Bit	7	6	5	4	3	2	1	0
Name	TCNT0							
Initial Value	0	0	0	0	0	0	0	0

Timer Interrupt Mask Register TIMSK



Bit	7	6	5	4	3	2	1	0
Name							OCIE0	TOIE
Initial Value	0	0	0	0	0	0	0	0

This register is used to activate/deactivate interrupts related with timers. This register controls the interrupts of all the three timers. The last two bits (BIT 1 and BIT 0) Controls the interrupts of TIMER0. TIMER0 has two interrupts .We will increment a variable “count” at every interrupt(OVERFLOW) if count reaches 61 we will toggle PORTC0 which is connected to LED and reset “count= 0”. Clock input of TIMER0 = 16MHz/1024 = 15625 Hz Frequency of Overflow =

15625 /256 = 61.0352 Hz if we increment a variable “count” every Overflow when “count reach 61” approx one second has elapse.

Setting Up the TIMER0

```
// Prescaler = FCPU/1024
   TCCR0|=(1<<CS02)|(1<<CS00);

//Enable Overflow Interrupt Enable
   TIMSK|=(1<<TOIE0);

//Initialize Counter
   TCNT0=0;
```

Now the timer is set and firing Overflow interrupts at 61.0352 Hz

The ISR

```
ISR(TIMER0_OVF_vect)
{
//This is the interrupt service routine for TIMER0 OVERFLOW Interrupt.
//CPU automatically call this when TIMER0 overflows.

//Increment our variable

   count++;
   if(count==61)
   {
   PORTC=~PORTC; //Invert the Value of PORTC
   count=0;
   }
}
```

Timer Interrupt 1

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.

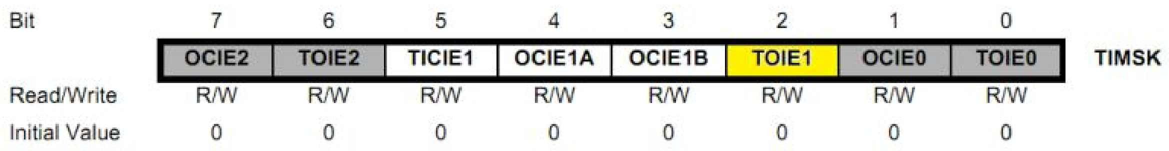
TCNT1 Registers:-

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 whereasTCNT1L 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	

TIMSK Register

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



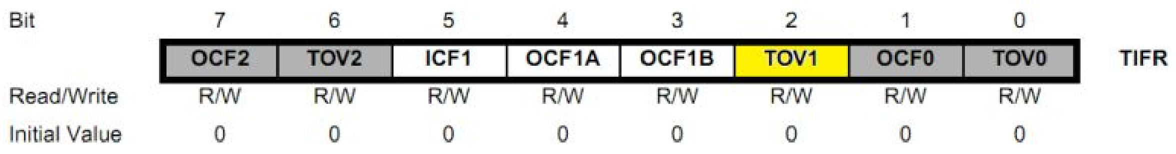
Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

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.

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	

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.

Timer Interrupt 2

TCCR2 Register

The Timer/Counter Control Register – TCCR2 is as follows:

Bit	7	6	5	4	3	2	1	0	
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	TCCR2
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	

We are only concerned with Bits2:0 – CS22:20 – Clock Select Bits. Unlike other timers, TIMER2 offers us with a wide range of prescalers to choose from. In TIMER0/1 the prescalers available are 8, 64, 256 and 1024, whereas in TIMER2, we have 8, 32, 64, 128, 256 and 1024!

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk_{T2S}/(No prescaling)
0	1	0	clk_{T2S}/8 (From prescaler)
0	1	1	clk_{T2S}/32 (From prescaler)
1	0	0	clk_{T2S}/64 (From prescaler)
1	0	1	clk_{T2S}/128 (From prescaler)
1	1	0	clk_{T2S}/256 (From prescaler)
1	1	1	clk_{T2S}/1024 (From prescaler)

Since we are choosing 256 as the prescaler, we choose the 7th option (110).

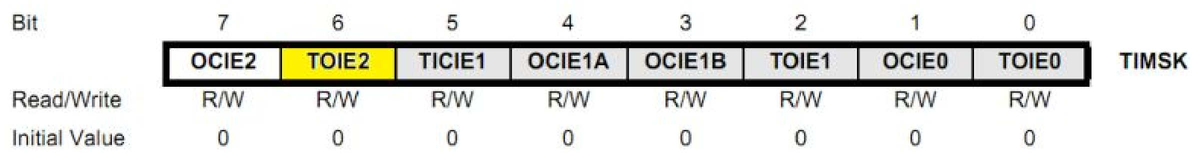
TCNT2 Register

In the Timer/Counter Register – TCNT2, the value of he timer is stored. Since TIMER2 is an 8-bit timer, this register is 8 bits wide.

Bit	7	6	5	4	3	2	1	0	
	TCNT2[7:0]								TCNT2
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

The Timer/Counter Interrupt Mask – TIMSK Register is as follows. It is a register common to all the timers.

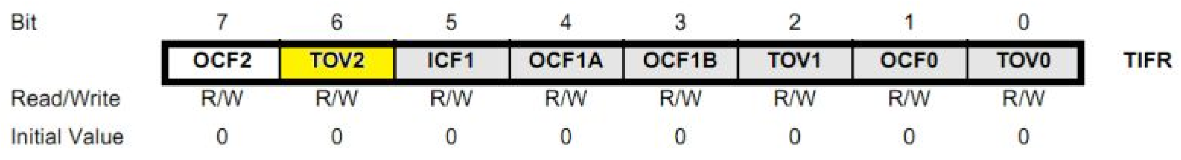


Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Here we are concerned with the 6th bit – TOIE2 – Timer/Counter2 Overflow Interrupt Enable. We set this to ‘1’ in order to enable overflow interrupts.

TIFR Register

The Timer/Counter Interrupt Flag Register – TIFR is as follows. It is a register common to all the timers.



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	

Here we are concerned with the 6th bit – TOV2 – Timer/Counter2 Overflow Flag. This bit is set (one) whenever the timer overflows. It is cleared automatically whenever the corresponding Interrupt Service Routine (ISR) is executed. Alternatively, we can clear it by writing ‘1’ to it.

The USART of the AVR

Here we are with the AVR communication protocols series, starting with the most basic ones, UART and USART!

Contents

UART and USART

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.

Some of the main features of the AVR USART are:

- Full Duplex Operation (Independent Serial Receive and Transmit Registers)
- Asynchronous or Synchronous Operation
- Master or Slave Clocked Synchronous Operation
- High Resolution Baud Rate Generator
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data bits and 1 or 2 Stop Bits

USART Layout – How to set it up?

AVR USART is fully compatible with the AVR UART in terms of register bit locations, baud rate generation, transmitter/receiver operations and buffer functionality. So let us now have a quick look at how to set up USART in general.

The first step is to set the baud rate in both, the master and the slave. The baud rate has to be the same for both – master and slave.

1. Set the number of data bits, which needs to be sent.
2. Get the buffer ready! In case of transmission (from AVR to some other device), load it up with the data to be sent, whereas in case of reception, save the previous data so that the new received data can be overwritten onto it.
3. Then enable the transmitter/receiver according to the desired usage.

One thing to be noted is that in UART, there is no master or slave since master is defined by the MicroController, which is responsible for clock pulse generation. Hence Master and Slave terms occur only in the case of USART.

Master µC is the one which is responsible for Clock pulse generation on the Bus.

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 (Pin 14)
2. TxD: USART Transmit Pin (Pin 15)
3. XCK: USART Clock Pin (Pin 1)

Modes of Operation

The USART of the AVR can be operated in three modes, namely-

1. Asynchronous Normal Mode
2. Asynchronous Double Speed Mode
3. Synchronous Mode

Asynchronous Normal Mode

In this mode of communication, the data is transmitted/received asynchronously, i.e. we do not need (and use) the clock pulses, as well as the XCK pin. The data is transferred at the BAUD rate we set in the UBBR register. This is similar to the UART operation.

Asynchronous Double Speed Mode

This is higher speed mode for asynchronous communication. In this mode also we set the baud rates and other initializations similar to Normal Mode. The difference is that data is transferred at double the baud we set in the UBBR Register.

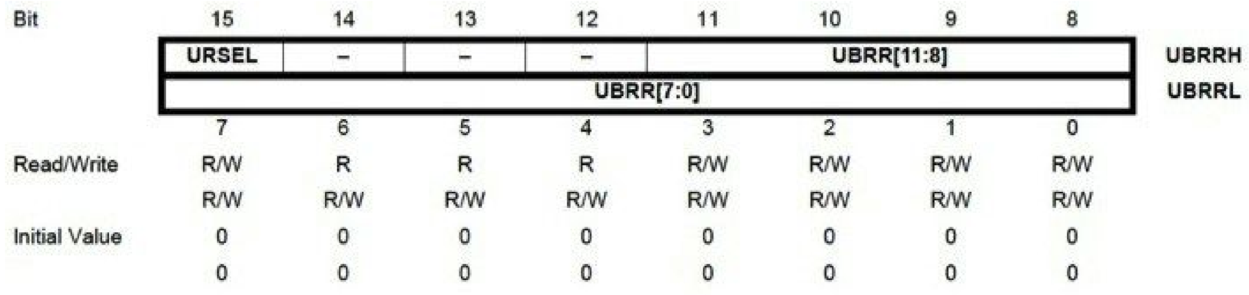
Setting the U2X bit in UCSRA register can double the transfer rate. Setting this bit has effect only for the asynchronous operation. Set this bit to zero when using synchronous operation. Setting this bit will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication. Note however that the Receiver will in this case only use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery, and therefore a more accurate baud rate setting and system clock are required when this mode is used. For the Transmitter, there are no downsides.

Synchronous Mode

This is the USART operation of AVR. When Synchronous Mode is used (UMSEL = 1 in UCSRC register), the XCK pin will be used as either clock input (Slave) or clock output (Master).

Baud Rate Generation

The baud rate of UART/USART is set using the 16-bit wide UBRR register. The register is as follows:



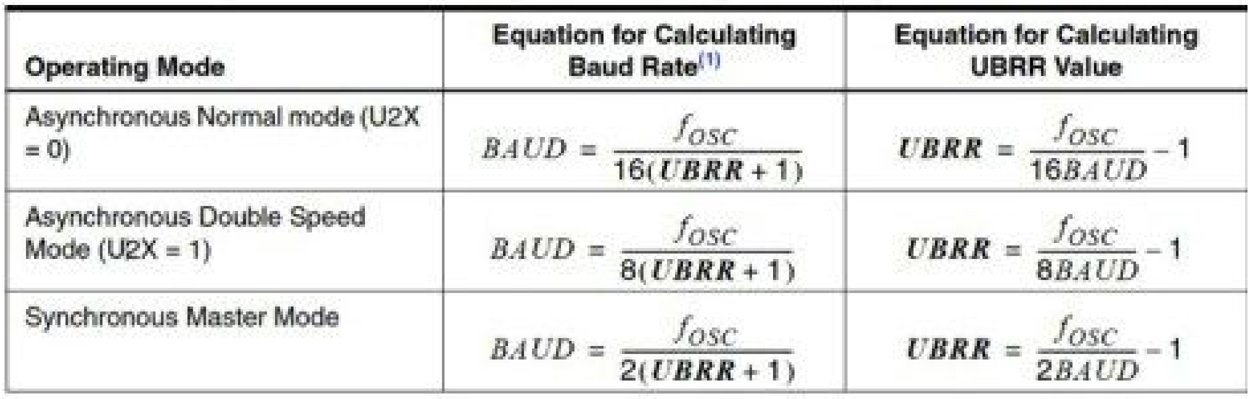
Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Since AVR is an 8-bit microcontroller, every register should have a size of 8 bits. Hence, in this case, the 16-bit UBRR register is comprised of two 8-bit registers – UBRRH (high) and UBRRL (low). This is similar to the 16-bit ADC register (ADCH and ADCL, remember?). Since there can be only specific baud rate values, there can be specific values for UBRR, which when converted to binary will not exceed 12 bits. Hence there are only 12 bits reserved for UBRR[11:0]. We will learn how to calculate the value of UBRR in a short while in this post.

The USART Baud Rate Register (UBRR) and the down-counter connected to it functions as a programmable prescaler or baud rate generator. The down-counter, running at system clock (FOSC), is loaded with the UBRR value each time the counter has counted down to zero or when the UBRRL Register is written. A clock is generated each time the counter reaches zero.

This clock is the baud rate generator clock output (`= FOSC/(UBRR+1)`). The transmitter divides the baud rate generator clock output by 2, 8, or 16 depending on mode. The baud rate generator output is used directly by the receiver's clock and data recovery units.

Below are the equations for calculating baud rate and UBRR value:



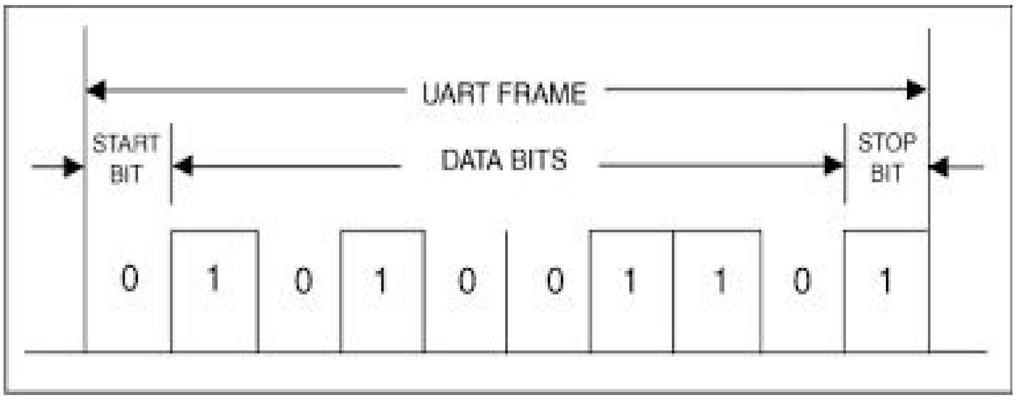
Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR + 1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR + 1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(UBRR + 1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

1. BAUD = Baud Rate in Bits/Second (bps) (Always remember, Bps = Bytes/Second, whereas bps = Bits/Second)
2. FOSC = System Clock Frequency (1MHz) (or as per use in case of external oscillator)
3. UBRR = Contents of UBRRL and UBRRH registers

Frame Formats

A frame refers to the entire data packet which is being sent/received during a communication. Depending upon the communication protocol, the formats of the frame might vary. For example, TCP/IP has a particular frame format, whereas UDP has another frame format. Similarly in our case, RS232 has a typical frame format as well. This is nothing but the selection of a frame format!

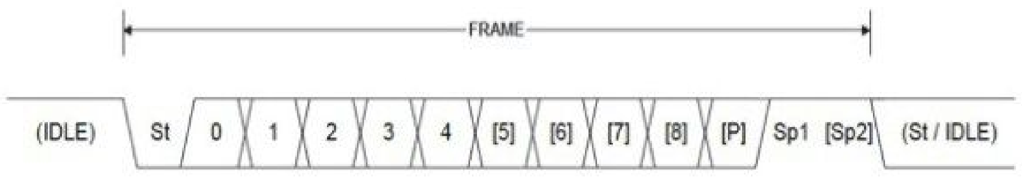
A typical frame for USART/RS232 is usually 10 bits long: 1 start bit, 8 data bits, and a stop bit. However a vast number of configurations are available… 30 to be precise!



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-



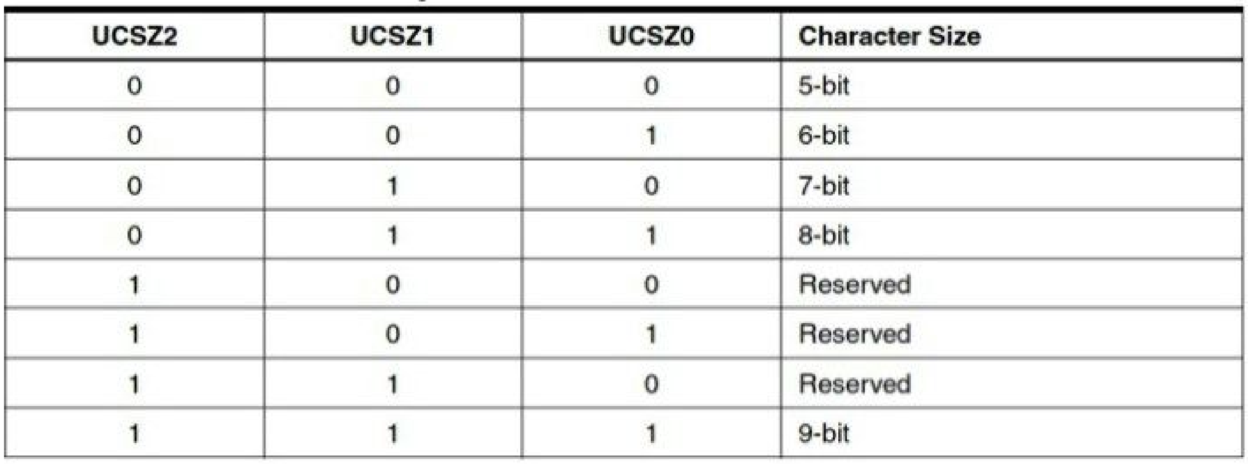
St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high.

Note: The previous image (not the above one, the one before that) of Frame Format has a flaw in it! If you can find it.

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.

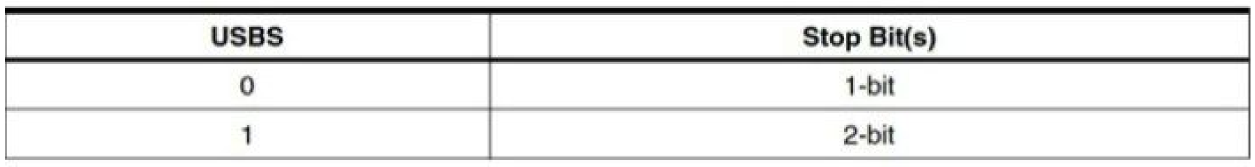


UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Data Bit Settings (Click to Enlarge)

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 (Click to Enlarge)

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.

Limitations of using single parity bit is that it can detect only single bit errors. If two bits are changed simultaneously, it fails.

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.

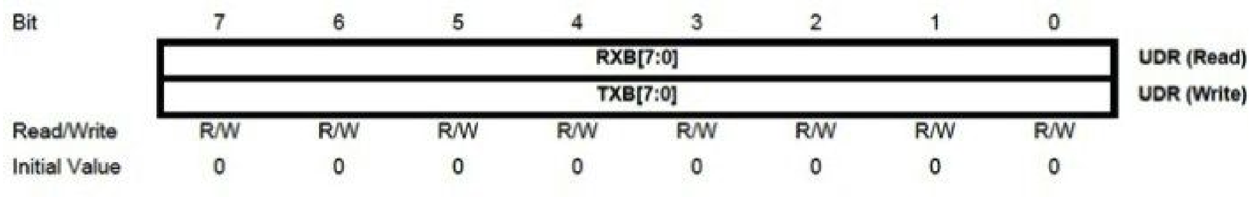
UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

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

Register Description

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.

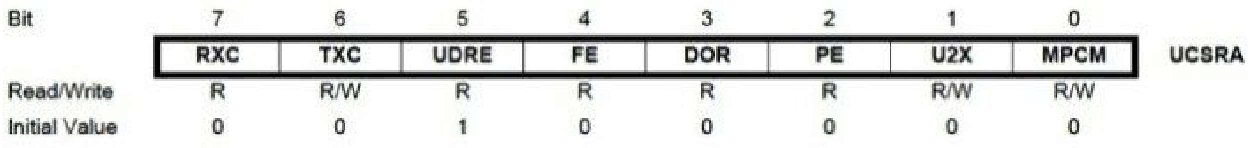
UDR: USART Data Register (16-bit)



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.

UCSRA: USART Control and Status Register A (8-bit)



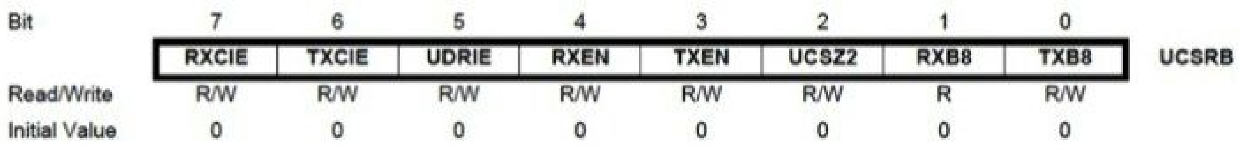
Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- Bit 7: RxC – USART Receive Complete Flag: This flag bit is set by the CPU when there are unread data in the Receive buffer and is cleared by the CPU when the receive buffer is empty. This can also be used to generate a Receive Complete Interrupt (see description of the RXCIE bit in UCSRB register).
- Bit 6: TxC – USART Transmit Complete Flag: This flag bit is set by the CPU when the entire frame in the Transmit Shift Register has been shifted out and there is no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a Transmit Complete Interrupt is executed, or it can be cleared by writing a *one (yes, one and NOT zero)* to its bit location. The TXC Flag can generate a Transmit Complete Interrupt (see description of the TXCIE bit in UCSRB register).
- Bit 5: UDRE – USART Data Register Empty: The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty Interrupt (see description of the UDRIE bit in UCSRB register). UDRE is set after a reset to indicate that the Transmitter is ready.
- Bit 4: FE – Frame Error: This bit is set if the next character in the receive buffer had a Frame Error when received (i.e. when the first stop bit of the next character in the receive buffer is zero). This bit is valid until the receive buffer

(UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

- Bit 3: DOR – Data Overrun Error: This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.
- Bit 2: PE – Parity Error: This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.
- Bit 1: U2X – Double Transmission Speed: This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.
- Bit 0: MPCM – Multi-Processor Communication Mode: This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting. This is essential when the receiver is exposed to more than one transmitter, and hence must use the address information to extract the correct information.

UCSRB: USART Control and Status Register B (8-bit)



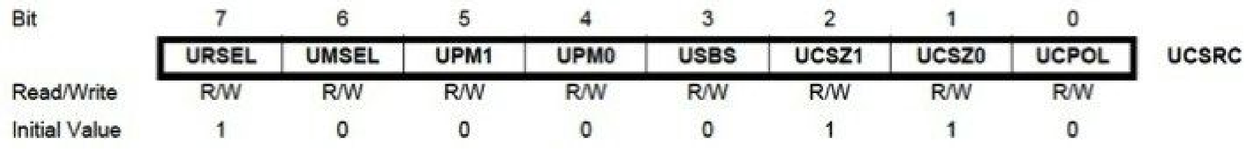
Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7: RXCIE – RX Complete Interrupt Enable: Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set. The result is that whenever any data is received, an interrupt will be fired by the CPU.
- Bit 6: TXCIE – TX Complete Interrupt Enable: Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set. The result is that whenever any data is sent, an interrupt will be fired by the CPU.
- Bit 5: UDRIE – USART Data Register Empty Interrupt Enable: Writing this bit to one enables interrupt on the UDRE Flag (remember – bit 5 in UCSRA?). A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set. The result is that whenever the transmit buffer is empty, an interrupt will be fired by the CPU.
- Bit 4: RXEN – Receiver Enable: Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled.
- Bit 3: TXEN – Transmitter Enable: Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled.

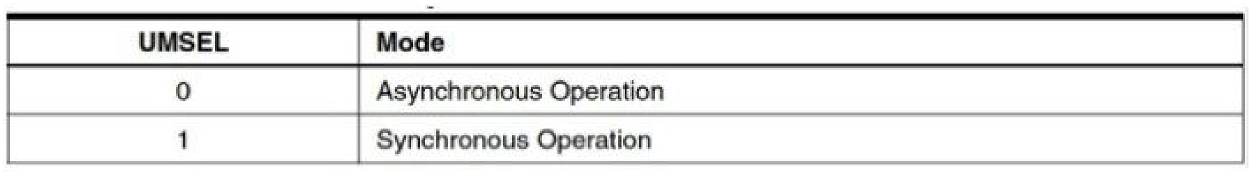
- Bit 2: UCSZ2 – Character Size: The UCSZ2 bits combined with the UCSZ1:0 bits in UCSRC register sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use. More information given along with UCSZ1:0 bits in UCSRC register.
- Bit 1: RXB8 – Receive Data Bit 8: RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. It must be read before reading the low bits from UDR.
- Bit 0: TXB8 – Transmit Data Bit 8: TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. It must be written before writing the low bits to UDR.

UCSRC: USART Control and Status Register C (8-bit)

The UCSRC register can be used as either UCSRC, or as UBRRH register. This is done using the URSEL bit.

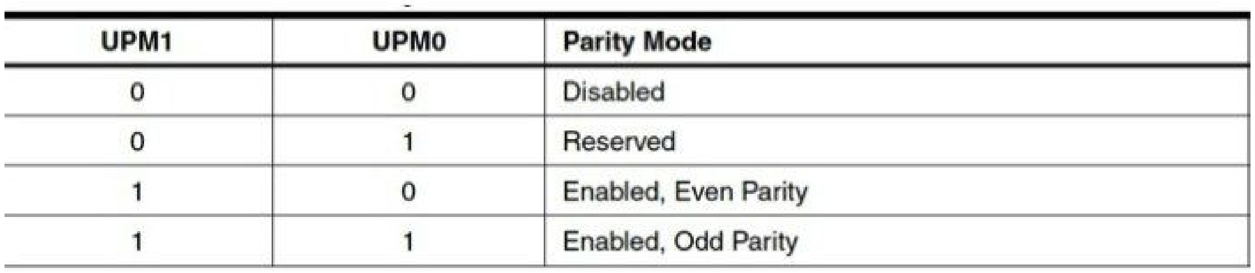


Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

- Bit 7: URSEL – USART Register Select: This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.
- Bit 6: UMSEL – USART Mode Select: This bit selects between Asynchronous and Synchronous mode of operation.
- 

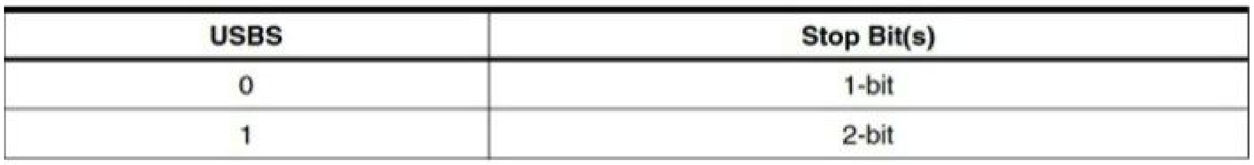
UMSEL	Mode
0	Asynchronous Operation
1	Synchronous Operation

- Bit 5:4: UPM1:0 – Parity Mode: This bit helps you enable/disable/choose the type of parity.

- 

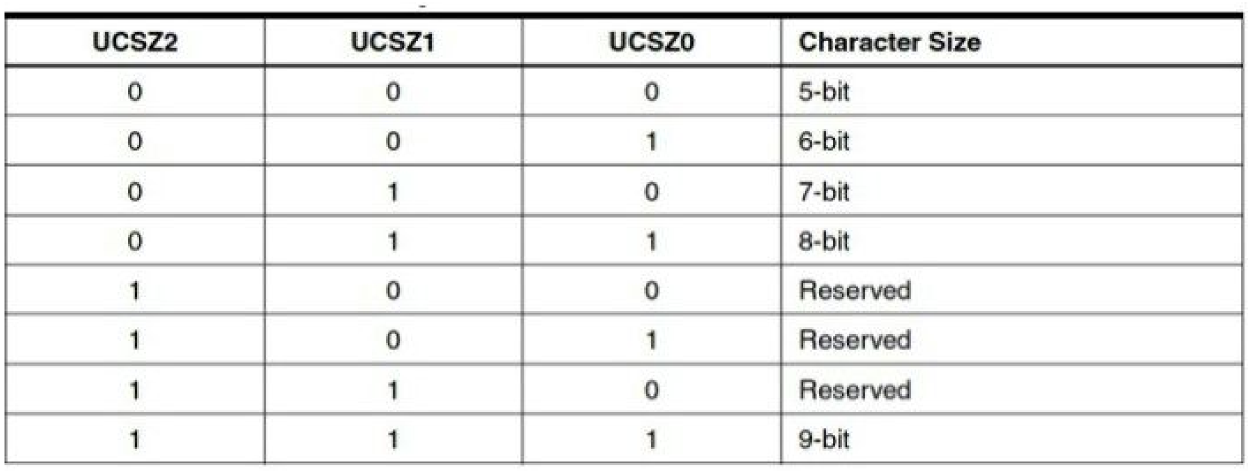
UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

Bit 3: USBS – Stop Bit Select: This bit helps you choose the number of stop bits for your frame.



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

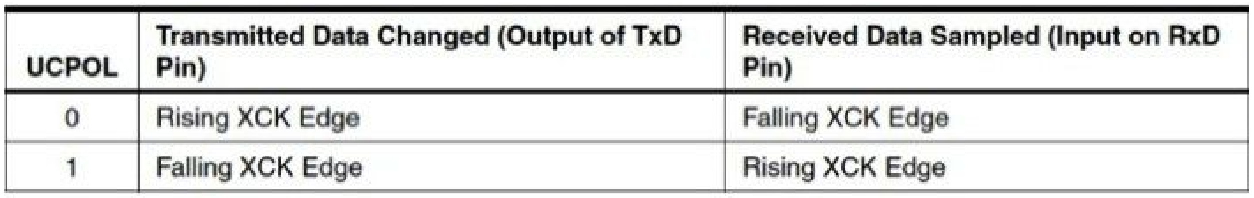
- Bit 2:1: UCSZ1:0 – Character Size: These two bits in combination with the UCSZ2 bit in UCSRB register helps choosing the number of data bits in your frame.

- 

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

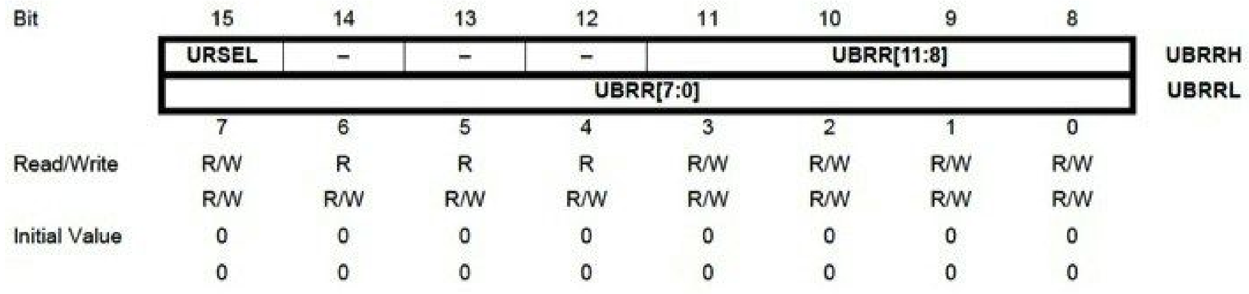
Bit 0: UCPOL – Clock Polarity: This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the

relationship between data output change and data input sample, and the synchronous clock (XCK).



UCPOL	Transmitted Data Changed (Output of TxD Pin)	Received Data Sampled (Input on RxD Pin)
0	Rising XCK Edge	Falling XCK Edge
1	Falling XCK Edge	Rising XCK Edge

UBRR: USART Baud Rate Register (16-bit)



Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

We have already seen this register, except the URSEL bit.

- Bit 15: URSEL: This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

Let’s code it!

For Transmitter :

```
/*
* Day4_4_1.c
*
* Created: 03-12-2015 10.38.20 PM
* Author : cc
*/

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

void USARTInit(uint16_t ubrr_value)
{
        UBRRL=ubrr_value;
        UBRRH=(ubrr_value>>8);

        //asynchronous mode;no parity;1 stop bit; char size 8
```

```
          UCSRC=(1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1);

          UCSRB=(1<<RXEN)|(1<<TXEN);
}



void USARTWriteChar(char data)
{
          while(!(UCSRA &(1<<UDRE)))
          {

          }
          UDR=data;
}

int main(void)
{
  char data1=21;
  char data2=5;
  USARTInit(51);

   while (1)
   {
                    USARTWriteChar(data1);
                    _delay_ms(500);
                    //USARTWriteChar(data2);
   }
}
```

For Receiver

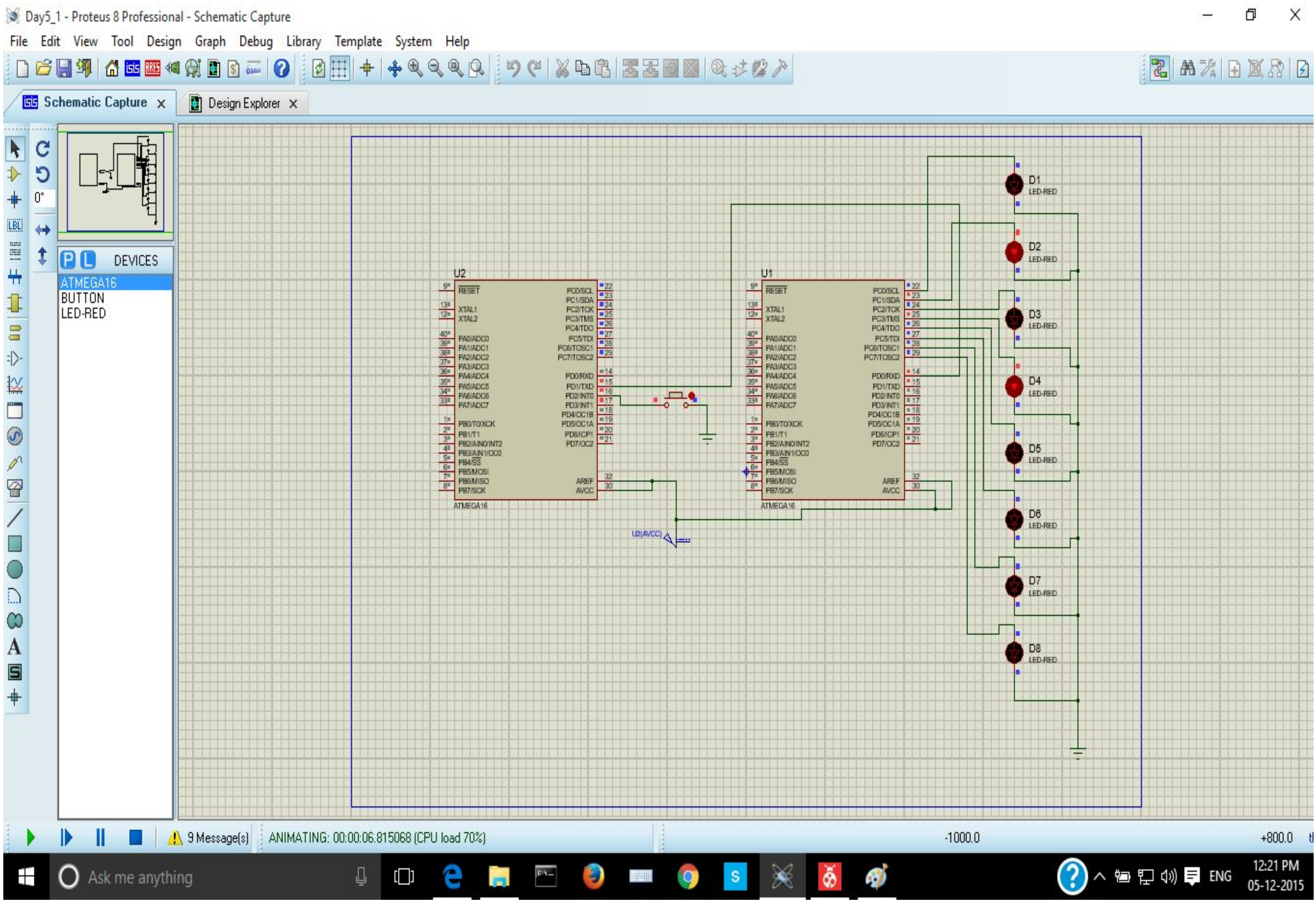
```
/*
* Day4_4_1.c
*
* Created: 03-12-2015 10.38.20 PM
* Author : cc
*/

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


void USARTInit(uint16_t ubrr_value)
{
        UBRRL=ubrr_value;
        UBRRH=(ubrr_value>>8);

        //asynchronous mode;no parity;1 stop bit; char size 8

        UCSRC=(1<<URSEL)|(3<<UCSZ0);

        UCSRB=(1<<RXEN)|(1<<TXEN);
}

char USARTReadChar()
{
        while(!(UCSRA &(1<<RXC)))
        {

        }

        return UDR;
}



int main(void)
{
        DDRC=0b11111111;

  char data;
  USARTInit(51);
   while (1)
   {
                data=USARTReadChar();
                /*if (data==10)
                {
                        PORTC=0b00000001;
                        _delay_ms(1000);
                        PORTC=0b00000000;
                        _delay_ms(300);
                }
                if (data==5)
                {
                        PORTC=0b00000010;
                        _delay_ms(300);
                        PORTC=0b00000000;
                        _delay_ms(100);
                }*/
                PORTC=data;
   }
```

}

Day 5

We start implementing code on USART for different problem statements.

1)Using interrupt we have to transmite data to other uC .


Day5_1 - Proteus 8 Professional - Schematic Capture
File Edit View Tool Design Graph Debug Library Template System Help
Schematic Capture
Design Explorer
DEVICES
ATMEGA16
BUTTON
LED-RED
U2
U1
ATMEGA16
D1
D2
D3
D4
D5
D6
D7
D8
LED-RED
9 Message(s)
ANIMATING: 00:00:06.815068 (CPU load 70%)
-1000.0
+800.0
Ask me anything
12:21 PM
05-12-2015

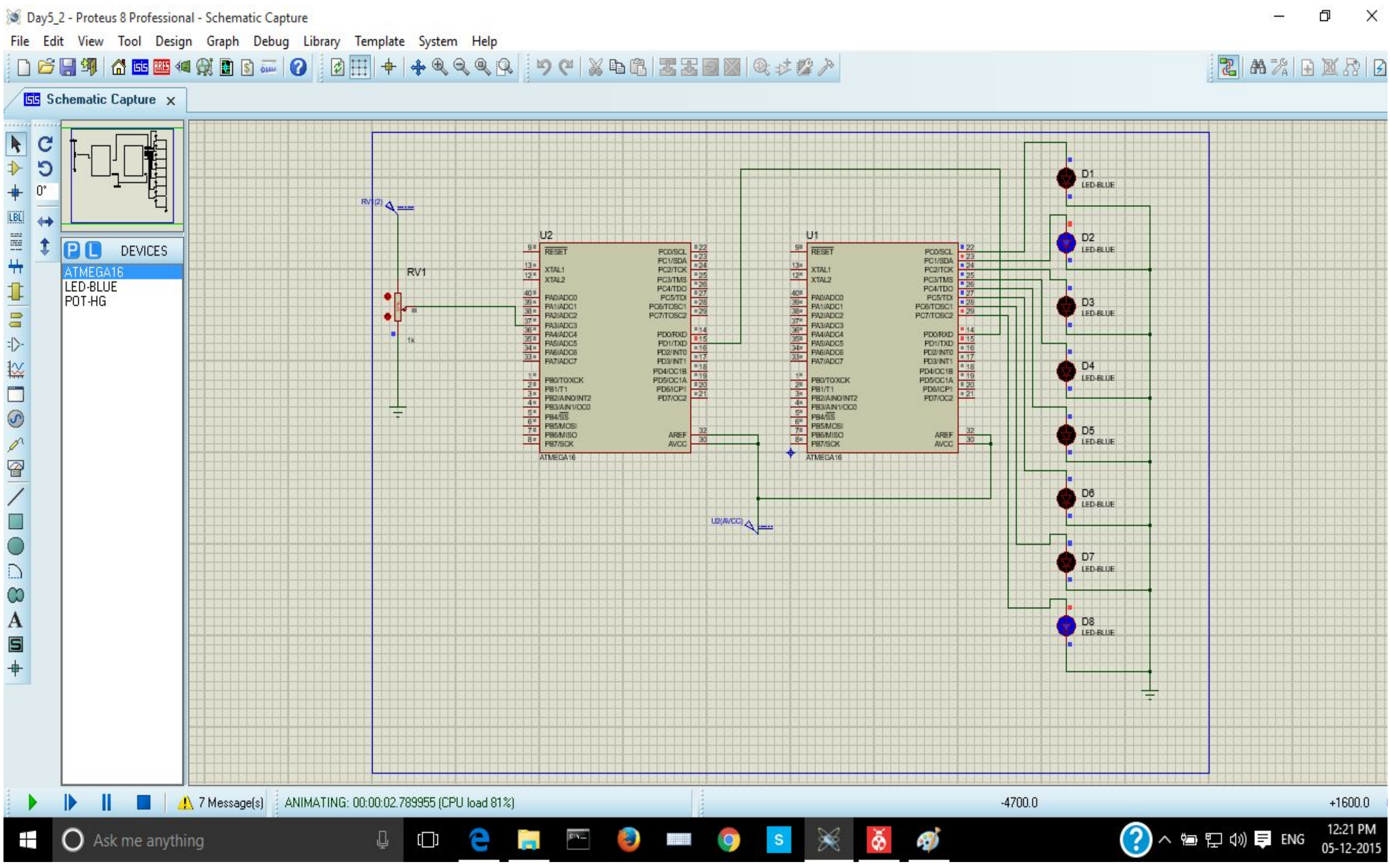
Code for transmitter:-

```
/*
 * Day5_1_1.c
 *
 * Created: 03-12-2015 12.49.05 PM
 * Author : Surya Jeet Singh
 */

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

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

void USARTInit(uint16_t ubrr_value)
{
    UBRRL=ubrr_value;
    UBRRH=(ubrr_value>>8);

    //asynchronous mode;no parity;1 stop bit; char size 8

    UCSRC=(1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1);

    UCSRB=(1<<RXEN)|(1<<TXEN);
}


void USARTWriteChar(char data)
{
    while(!(UCSRA &(1<<UDRE)))
    {
    }
    UDR=data;
}


ISR(INT0_vect)
{
    char data1=10;
    USARTInit(51);
         USARTWriteChar(data1);

}

ISR(INT1_vect)
{

    PORTC=0b00000010;
    _delay_ms(1000);
    PORTC=0b00000000;
}
int main(void)
{
    DDRD=1<<PD2|1<<PD3;
    PORTD=1<<PD2|1<<PD3;

    DDRC=0b11111111;



    GICR=1<<INT0 | 1<<INT1;
    MCUCR=(1<<ISC01)|(1<<ISC00)|(1<<ISC10)|(1<<ISC11);

    sei();

         while (1)
         {
        }
}
```

Code for receiver:-

```
/*
 * Day5_1_2.c
 *
 * Created: 03-12-2015 10.38.20 PM
 * Author : Surya Jeet Singh
 */

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


void USARTInit(uint16_t ubrr_value)
{
   UBRRL=ubrr_value;
   UBRRH=(ubrr_value>>8);

   //asynchronous mode;no parity;1 stop bit; char size 8

   UCSRC=(1<<URSEL)|(3<<UCSZ0);

   UCSRB=(1<<RXEN)|(1<<TXEN);
}

char USARTReadChar()
{

   while(!(UCSRA &(1<<RXC)))
   {
    }

    return UDR;
}



int main(void)
{
    DDRC=0b11111111;

   char data;
   USARTInit(51);
         while (1)
   {
          data=USARTReadChar();
          PORTC=data;
          _delay_ms(1000);
          PORTC=0b00000000;
         }
}
```

2)We have to transmitte ADC value to other uC.


Day5_2 - Proteus 8 Professional - Schematic Capture
File Edit View Tool Design Graph Debug Library Template System Help
Schematic Capture
DEVICES
ATMEGA16
LED-BLUE
POT-HG
RV1
U2
U1
ATMEGA16
D1
D2
D3
D4
D5
D6
D7
D8
LED-BLUE
7 Message(s)
ANIMATING: 00:00:02.789955 (CPU load 81%)
-4700.0
+1600.0
Ask me anything
ENG
12:21 PM
05-12-2015

Transmitter code:-

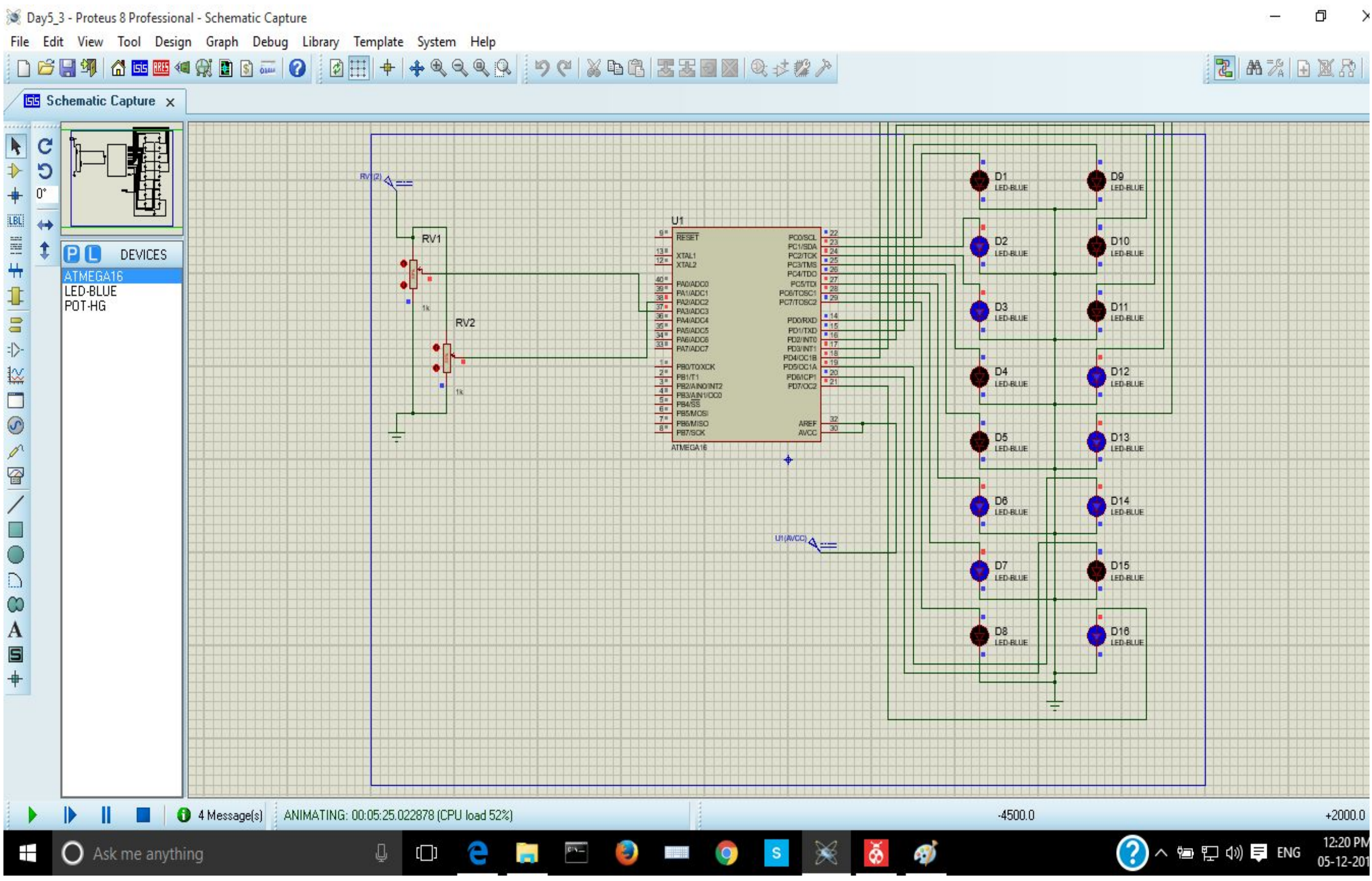
```
/*
 * Day5_2_1.c
 *
 * Created: 04-12-2015 3.57.55 PM
 * Author : Surya Jeet Singh
 */

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

void USARTInit(uint16_t ubrr_value)
{
   UBRRL=ubrr_value;
   UBRRH=(ubrr_value>>8);
```

```
    //asynchronous mode;no parity;1 stop bit; char size 8

    UCSRC=(1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1);

    UCSRB=(1<<RXEN)|(1<<TXEN);
}



void USARTWriteChar(char data)
{
    while(!(UCSRA &(1<<UDRE)))
    {

    }
    UDR=data;
}

int main(void)
{


    DDRA=0b00000000;
    ADMUX=0b00000011;
    ADCSRA|=(1<<ADEN)|(7<<ADPS0);
    USARTInit(51);

    while (1)
    {
        ADCSRA|=(1<<ADSC);
        while(!(ADCSRA & (1<<ADSC)));
        if(ADC<256)
        {
            USARTWriteChar(ADC);
        }
    }
}
```

Receiver code:-

```
 * Day5_2_2.c
 *
 * Created: 03-12-2015 10.38.20 PM
 * Author : Surya Jeet Singh
 */

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


void USARTInit(uint16_t ubrr_value)
{
   UBRRL=ubrr_value;
   UBRRH=(ubrr_value>>8);

   //asynchronous mode;no parity;1 stop bit; char size 8

   UCSRC=(1<<URSEL)|(3<<UCSZ0);

   UCSRB=(1<<RXEN)|(1<<TXEN);
}

char USARTReadChar()
{

   while(!(UCSRA &(1<<RXC)))
   {

   }

   return UDR;
}



int main(void)
{
    DDRC=0b11111111;

   char data;
   USARTInit(51);
          while (1)
   {
           data=USARTReadChar();
           PORTC=data;

          }
}
```

3) we have to take multiple ADC value.



Code:-

```
/*
 * Day5_3.c
 *
 * Created: 02-12-2015 12.03.02 AM
 * Author : Surya Jeet Singh
 */

#include <avr/io.h>

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

volatile uint8_t x;
volatile uint8_t y;
```

```
int main(void)
{

    uint8_t y,z;
    ADCSRA|=(1<<ADEN)|(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2);
    ADMUX|=(1<<REFS0);
        DDRA=0b00000000;
    DDRC=0b11111111;
    DDRD=0b11111111;

    while(1)
    {

        ADMUX=(1<<MUX1);
        ADCSRA=0b11000111;
        while(((ADCSRA)&(1<<ADSC)));


        //y=readadc(1);
        //z=readadc(2);
        y=ADC;
        PORTC=y;
        //_delay_ms(500);

        ADMUX=1<<MUX1|1<<MUX0;
        ADCSRA=0b11000111;
        while(((ADCSRA)&(1<<ADSC)));

        z=ADC;
        PORTD=z;
        //_delay_ms(500);
    }
}
```

After all this we set threshold value for accelerometer.

Day 6

WE that day work on sonar and threshold the accelerometer.

Code For Sonar:-

```
//We are calculating the distane of object in cm
//#include "sonar.h"//Including the initialized functions, ports and
variables

#define F_CPU 16000000UL        // CPU Frequency


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

/*
 * Define Ports and Pins as required
 * Modify Maximum response time and delay as required
 * MAX_RESP_TIME : default: 300
 * DELAY_BETWEEN_TESTS : default: 50
*/
#define TRIG_DDR     DDRD                  // Trigger Port
#define TRIG_PORT    PORTD
#define TRIG_PIN     PIND
#define TRIG_BIT     PD0                   // Trigger Pin************
```

```
#define ECHO_DDR    DDRD                        // Echo Port
#define ECHO_PORT  PORTD
#define ECHO_PIN     PIND
#define ECHO_BIT     PD1                         // Echo Pin***************

// Speed of sound
// Default: 343 meters per second in dry air at room temperature (~20C)
#define SPEED_OF_SOUND        343
#define MAX_SONAR_RANGE 4                   // This is trigger + echo
range (in meters) for SR04*******needs thresholding
#define DELAY_BETWEEN_TESTS       500             // Echo canceling
time between sampling. Default: 500us
#define TIMER_MAX 65535                         // 65535 for 16 bit timer
and we are using here TIMER1 which is 16 bit

/*
 * Do not change anything further unless you know what you're doing
 Please refer data sheet for changes
 * */
#define TRIG_ERROR -1
#define ECHO_ERROR -2    //let

#define CYCLES_PER_US (F_CPU/16000000)// instructions per
microsecond
#define CYCLES_PER_MS (F_CPU/1000)              // instructions per
millisecond
// Timeout. Decreasing this decreases measuring distance
// but gives faster sampling
#define SONAR_TIMEOUT
((F_CPU*MAX_SONAR_RANGE)/SPEED_OF_SOUND)

#define TRIG_INPUT_MODE() TRIG_DDR &= ~(1<<TRIG_BIT)
#define TRIG_OUTPUT_MODE() TRIG_DDR |= (1<<TRIG_BIT)
#define TRIG_LOW() TRIG_PORT &= ~(1<<TRIG_BIT)
#define TRIG_HIGH() TRIG_PORT |=(1<<TRIG_BIT)
```

```
#define ECHO_INPUT_MODE() ECHO_DDR &= ~(1<<ECHO_BIT)
#define ECHO_OUTPUT_MODE() ECHO_DDR |= (1<<ECHO_BIT)
#define ECHO_LOW() ECHO_PORT &= ~(1<<ECHO_BIT)
#define ECHO_HIGH() ECHO_PORT |=(1<<ECHO_BIT)

#define CONVERT_TO_CM ((10000*2)/SPEED_OF_SOUND)  // or
simply 58

/**
 * Initiate Ports for Trigger and Echo pins
*/
void init_sonar();

/**
 * Send 10us pulse on Ultrasonic Trigger pin
*/
void trigger_sonar();

/**
 * Calculate and store echo time and return distance
 * parameter:      void
 * return type:    unsigned int
 * Usage:   int foo = read_sonar(); - basically syntax
*/
unsigned int read_sonar();



volatile uint32_t overFlowCounter = 0;
volatile uint32_t trig_counter = 0;
volatile uint32_t no_of_ticks = 0;
//To be on the safe side using 32 bit binary data
/*******************************************************************
 * Initiate Ultrasonic Module Ports and Pins
 * Input:   none
 * Returns: none
***************************************************************************/
void init_sonar(){
        TRIG_OUTPUT_MODE();             // Set Trigger pin as output
        ECHO_INPUT_MODE();              // Set Echo pin as input
}

/***************************************************************************
 * Send 10us pulse on Sonar Trigger pin
 * 1.   Clear trigger pin before sending a pulse
 * 2.   Send high pulse to trigger pin for 10us
 * 3.   Clear trigger pin to pull it trigger pin low
 *      Input:   none
 *      Returns: none
***************************************************************************/
void trigger_sonar(){
        TRIG_LOW();                             // Clear pin before setting it high
    _delay_us(1);                   // Clear to zero and give time for electronics
to set
    TRIG_HIGH();                    // Set pin high
    _delay_us(12);                          // Send high pulse for minimum 10us
    TRIG_LOW();                             // Clear pin
    _delay_us(1);                   // Delay not required, but just in case...
}

/***************************************************************************
 * Increment timer on each overflow
 * Input:   none
 * Returns: none
***************************************************************************/
ISR(TIMER1_OVF_vect){      // Timer1 overflow interrupt
        overFlowCounter++;
        TCNT1=0;
}
```

```
/***********************************************************************
 * Calculate and store echo time and return distance
 * Input:   none
 * Returns: 1. -1           :       Indicates trigger error. Could not pull trigger
high
 *                      2. -2           :       Indicates echo error. No echo
received within range
 *                      3. Distance :       Sonar calculated distance in cm.
***********************************************************************/
unsigned int read_sonar()
{
        int dist_in_cm = 0;
        init_sonar();                                           // Setup pins and
ports
        trigger_sonar();                                // send a 10us high pulse

        while(!(ECHO_PIN & (1<<ECHO_BIT)))
        {       // while echo pin is still low
                trig_counter++;
                 uint32_t max_response_time = SONAR_TIMEOUT;
                if (trig_counter > max_response_time)
                {       // SONAR_TIMEOUT
                        return TRIG_ERROR;
                }
        }

        TCNT1=0;                                                // reset timer
        TCCR1B |= (1<<CS10);                                    // start 16 bit timer
with no prescaler ie taking x=1
        TIMSK |= (1<<TOIE1);                            // enable overflow
interrupt on timer1
        overFlowCounter=0;                                      // reset overflow
counter
```

```
        sei();                                                  // enable global
interrupts

        while((ECHO_PIN & (1<<ECHO_BIT)))
        {       // while echo pin is still high
                if (((overFlowCounter*TIMER_MAX)+TCNT1) >
SONAR_TIMEOUT)
                {
                        return ECHO_ERROR;              // No echo within
sonar range
                }
        }

        TCCR1B = 0x00;                                          // stop 16 bit timer
with no prescalier
        cli();                                                  // disable global
interrupts
        no_of_ticks = ((overFlowCounter*TIMER_MAX)+TCNT1); //
counter count
        dist_in_cm =
(no_of_ticks/(CONVERT_TO_CM*CYCLES_PER_US));   // distance in
cm
        return (dist_in_cm );
}

int main(void)
{
        DDRB=0b11111111;
        PORTB=read_sonar();
        _delay_ms(50);
}
```

Day-7

Triple Axis calculation:

Introducing the third axis, the orientation of the sensor determines the complete sphere. Thus, classical method of rectangular(x,y,z) to spherical (ρ,$\boldsymbol{\Theta}$,ϕ) conversion can be used to relate the angle of tilt in XY plane.

This implies that,

`

$$\boldsymbol{\Theta}= tan^{-1}(Ax.out/Ay.out)$$

$$\phi= cos^{-1}(Az.out/(\sqrt{A^2.x.out+}A^2y.out+A^2.z.out))$$

Similarly,

$$\boldsymbol{\Theta}= tan^{-1}(Ax.out/\sqrt{A^2y.out+A^2z.out}\ )$$

$$\boldsymbol{\Psi}= tan^{-1}(Ay.out/\sqrt{A^2x.out+A^2z.out}\ )$$

$$\boldsymbol{\Phi}= tan^{-1}(\sqrt{A^2y.out+A^2x.out}\ )/Az.out$$

Sonar receiver code:

```
//Fuse Bits   :   l:EE, h:D9, E:07
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include "sonar.h"
#include "sonar.c"

//This function is used to read the available data
//from USART. This function will wait until data is
//available.
uint8_t USARTReadChar()
{
	//Wait until data is available
	while(!(UCSRA & (1<<RXC)))
	{
		//Do nothing
```

```
        }

        //Now when USART receives data from host
        //and it is available in the buffer
        return UDR;
}

//This function is used to initialize the USART
//at a given UBRR value
void USARTInit(uint16_t ubrr_value)
{
        //Set Baud rate
        UBRRL = ubrr_value;
        UBRRH = (ubrr_value>>8);

        UCSRC=(1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1);

        //Enable The receiver and transmitter
        UCSRB=(1<<RXEN)|(1<<TXEN);
}

int main(){
        int distance_in_cm=0;
        DDRB=255;
        char data;
        USARTInit(103);
        while(1)
        {
                 DDRB=0xFF;//for setting led to output

                 //Read data
                 data=USARTReadChar();

                 //Use the ASCII value to light up the LEDs
                 PORTB=data;
                distance_in_cm=read_sonar();
                if(distance_in_cm<32)
                PORTB=0b00001111;
                //_delay_ms(100);
                //break;
                //_delay_ms(100);


                /*else{
                        ADMUX=0b00000000;
                        ADCSRA=0b11000111;
                        //while(ADCSRA&(1<<ADSC));
                        x=ADC;
```

```
				if(x>=200) PORTB=0b00001010;
				if(x<200 && x>=50) PORTB=0b00000000;
				if(x<50) PORTB=0b00000101;
			}

			if (distance_in_cm == TRIG_ERROR)
			{
				//ERROR!!!
				PORTB=0b11111111;
				_delay_ms(DELAY_BETWEEN_TESTS/2);
				PORTB=0;
				_delay_ms(DELAY_BETWEEN_TESTS/2);

			}
			else if (distance_in_cm == ECHO_ERROR)
			{
				//CLEAR!!....No obstacle
				PORTB=0b01010101;
				_delay_ms(DELAY_BETWEEN_TESTS);
				PORTB=0;
			}
			else
			{
				//obstacle detected
				PORTB=0b11110000;
				_delay_ms(DELAY_BETWEEN_TESTS);
				PORTB=0;
			}*/
		}
		return 0;
}
```

The bot after doing the final demonstration of the problem statement detecting the gesture controlled obstacle avoider