A Beginners Guide to AVR

Technical Report · June 2014

CITATION
1

READS
23,813

1 author:

Aravind E Vijayan
ETH Zurich
8 PUBLICATIONS **66** CITATIONS

SEE PROFILE

A Beginner's Guide to AVR

Aravind E Vijayan

Contents

AVR Microcontroller Series

The AVR architecture was developed by two graduate students of Norwegian Institute of Technology, Alf-Egil and Vegard Wollan as part of their master's degree thesis. AVR is a Harvard architecture 8 bit RISC microcontroller. This is one of the first microcontrollers to use an on chip flash memory to store the program code. Anyone would wonder what **AVR** stands for. Well stands for **A**lf egil Bogen **V**egard Wollan **R**ISC microcontroller, also known as **A**dvanced **V**irtual **R**ISC.

You might be wondering what this RISC means?

RISC stands for Reduced Instruction Set Computing. A microcontroller instruction set may follow either of the following:

1) RISC - Reduced Instruction Set Computing
2) CISC - Complex Instruction Set Computing

RISC simply means that each instruction designed for the microcontroller can perform only a single task. For example

ADD R1, R2

This is an assembly instruction which is there in the AVR instruction set. This instruction performs only one single task. Add two values. If all the instructions are like this, then it is called RISC instruction set.

So what is Instruction Set?

Yes it is the set of (or let's say it's a list of) all the instructions (for e.g. ADD, SUB) designed for the microcontroller architecture.

CISC instructions can perform more than one operation per instruction. You will understand it more clearly as the tutorial progresses. CISC is mostly used in DSP processors. For example, let's say we have a DSP processor which has an instruction **XYZ.** If XYZ can performs more than one operations, for example add two values and then find the square of that value, then it is called a CISC instruction.

Having understood the difference between the RISC and CISC instructions, let's go back to AVR and check the variants of microcontrollers which use the AVR architecture.

There are three kind of microcontroller series available in the market which are based on the AVR architecture:

1) Tiny AVR
2) Mega AVR
3) Xmega AVR

The following table compares the above mentioned AVR microcontroller series:

uC Series	Pins	Memory	Remarks
TinyAVR	**6-32**	**0.5-8 KB**	**Small size**
MegaAVR	**28-100**	**4-256 KB**	**More peripherals**
XmegaAVR	**44-100**	**16-384 KB**	**DMA**

Introduction to Embedded Coding

Embedded coding in C is more or less like normal C/C++ coding, except for the libraries and library functions that we use in embedded coding are different. But why use AVR GCC when Arduino gets the same thing done with less effort. Arduino constructs and header files compromised the efficiency of code for simplicity of coding. Often, we find that codes written in Arduino are slower than those written in C using AVR GCC. Therefore Arduino is not suitable when efficiency is crucial for our application.

So how simple is Embedded C coding? Think I had to suddenly change the microcontroller that I was using. For example, from ATmega 16 to ATmega 32. Do I have to write the code from the very beginning?

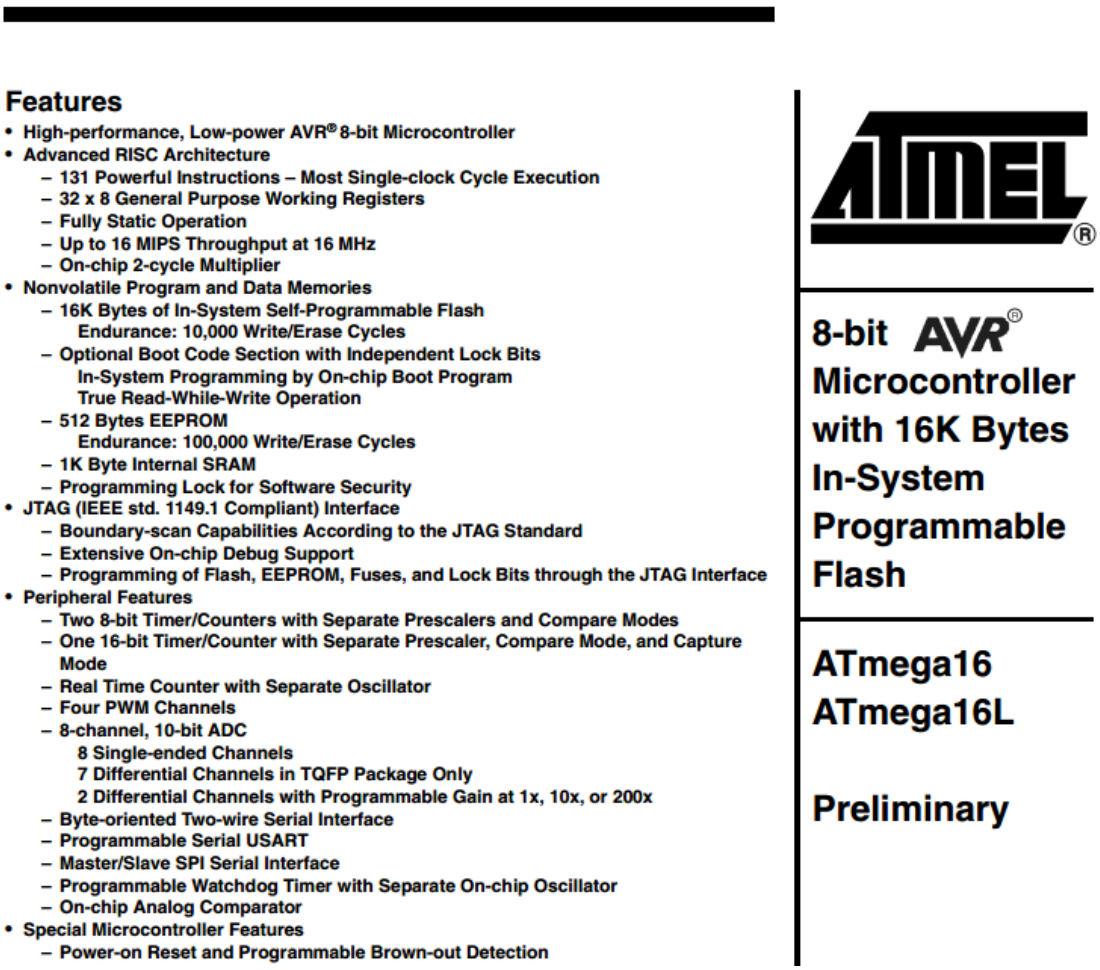
The Answer is No. Provided the microcontrollers belong to the same architecture (in our case AVR), significant changes in code won't be there. If header files are available, then AVR coding is easy, hassle free and much more efficient than Arduino. In this tutorial series I will be using ATMega16 for implementations.

What is a datasheet?

Every now and then, when you work with electronics components you hear about specifications. You would wonder if your circuit blow up. So how to ensure that your design is suited for the electrical and mechanical characteristics of the microcontroller?

This is when a datasheet becomes useful for you. A datasheet is an official document published by the device manufacturer which contains all the information a developer would need like the electrical characteristics say maximum voltage, maximum output current, no of input pins, other features of the microcontroller etc.

Any electronic component available in the market has a datasheet. Even LED and buzzer has their datasheets.



Features

- High-performance, Low-power AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- Nonvolatile Program and Data Memories
  - 16K Bytes of In-System Self-Programmable Flash
    Endurance: 10,000 Write/Erase Cycles
  - Optional Boot Code Section with Independent Lock Bits
    In-System Programming by On-chip Boot Program
    True Read-While-Write Operation
  - 512 Bytes EEPROM
    Endurance: 100,000 Write/Erase Cycles
  - 1K Byte Internal SRAM
  - Programming Lock for Software Security
- JTAG (IEEE std. 1149.1 Compliant) Interface
  - Boundary-scan Capabilities According to the JTAG Standard
  - Extensive On-chip Debug Support
  - Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Four PWM Channels
  - 8-channel, 10-bit ADC
    8 Single-ended Channels
    7 Differential Channels in TQFP Package Only
    2 Differential Channels with Programmable Gain at 1x, 10x, or 200x
  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection

ATMEL®

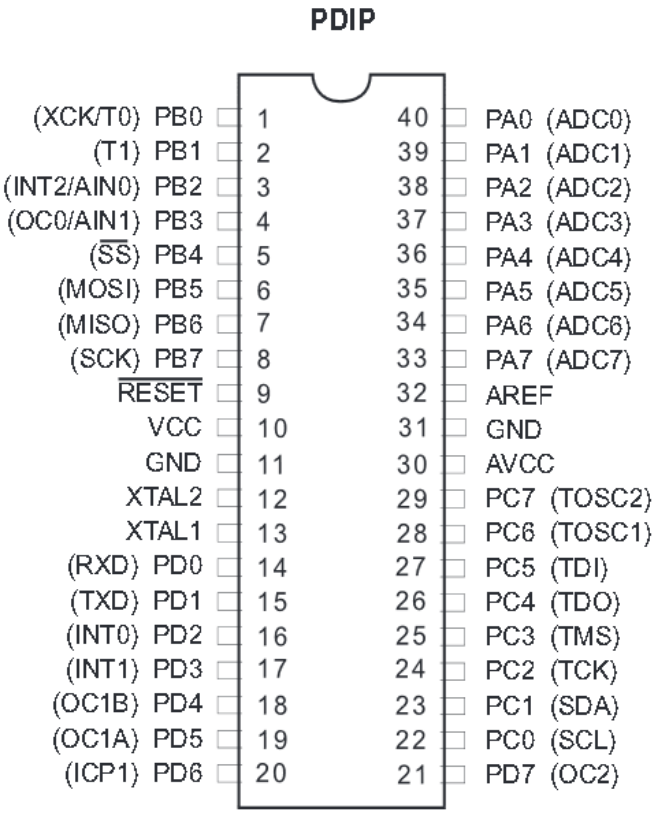
8-bit AVR® Microcontroller with 16K Bytes In-System Programmable Flash

ATmega16
ATmega16L

Preliminary

Atmel AVR ATmega 16

As discussed earlier, ATmega 16 is an AVR microcontroller manufactured by Atmel. It follows 8 bit RISC Harvard architecture. The pin out of the microcontroller is given below:



This is the pin diagram of the Dual Inline Package of ATmega 16. The microcontroller is also available in TQFP and QFN packages. Mentioned just for your information.

Now take a glimpse at the pinout. Let's understand the pinout of this microcontroller in detail. ATmega 16 is a 40 pin IC. The pins are numbered from 1 to 40. Now consider this

(Additional Feature) Primary Function – Pin number

This is the fashion in which the pins are named here. For example pin no 40 primarily function as PA0(we will discuss what this shortly), whereas it also has the additional capability to function as ADC0.Also we note that pin 10 is VCC, pin 11 and 31 are GND, whereas Pin 30 is AVCC. AVCC is the supply for the ADC module and must be either shorted to VCC or connected to the separate analog supply line if any. Still the AVCC should be within VCC +/- 3V.

Electrical Characteristics

The table shown below has been snipped from the 291st page of ATmega 16 datasheet:

Electrical Characteristics

Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin except $\overline{\text{RESET}}$ with respect to Ground	-0.5V to V_{CC}+0.5V
Voltage on $\overline{\text{RESET}}$ with respect to Ground	-0.5V to +13.0V
Maximum Operating Voltage	6.0V
DC Current per I/O Pin	40.0 mA
DC Current V_{CC} and GND Pins	200.0 mA PDIP and 400.0 mA TQFP/MLF

The table is self-explanatory. The voltage supply may range from **2.7V – 5.5V** normally in case of ATmega 16L while it is **4.5V – 5.5V** in the case of ATmega 16.

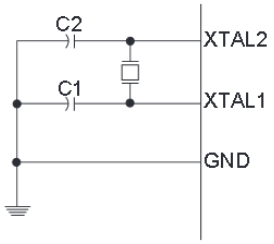
Features of ATmega 16

Now let's try to understand the features of ATmega 16. Some of the important features of the microcontroller are listed below:

1) 4 8-bit **GPIO** ports
2) 1 **UART** (serial communication) module
3) 1 **SPI** module (used to interface SD card, Ethernet module etc.)
4) **I2C** module (used to communicate with temperature sensors, accelerometers etc.)
5) 512 Bytes **EEPROM**
6) Two 8-bit and One 16-bit **Timer/Counter** module (used for wave generation)
7) Four **PWM** channels (for servo motor control)
8) 8-channel 10 bit **ADC**

Clock Source and various options available

A microcontroller is a synchronous device which means that it need a clock so that it can time its functioning. By default ATmega 16 has a 1 MHz RC oscillator which provides the square wave clock signal which the microcontroller uses as clock source. The problem with the RC oscillator is that it is not accurate. If we are not satisfied with the operating frequency, we have an option to connect a crystal externally to clock the microcontroller. A crystal is an oscillator with two pins. The figure shows how to connect your crystal to the microcontroller.



We cannot connect any arbitrarily fast crystal to the microcontroller. For the ATmega 16, the maximum crystal frequency is 16 MHz whereas it is 8 MHz for AT mega 16L.

So what about the capacitors shown in the figure?

The recommended capacitance values are in the range of 12 – 22 pF. If you do not connect the capacitors, the crystal won't generate stable clock signal and therefore, the microcontroller won't turn on. But connecting a crystal doesn't mean that the microcontroller will work at that frequency. There is little programming involved to configure the microcontroller to work using the crystal which will be dealt at a later stage in this tutorial.

General Purpose Registers

The general purpose register is used to store the local variables and the temporary data used by the program. Each of the I/O register configures the control registers of the various I/O peripherals of the uC. Each of the registers can have a value of either 0 or 1. The value given to these registers determine whether the various special features of the uC is activated or not. ATmega is an 8 bit microcontroller which means that each register in this microcontroller will be of 8 bit width. We cannot access each bit of the register individually, but can only edit the value of the register as a whole. We will learn how to tackle this drawback shortly. Given below is a register in the ADC module.

ADC Multiplexer Selection Register – ADMUX

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	

So if you want to put a '1' into REFS1 you have to do the following

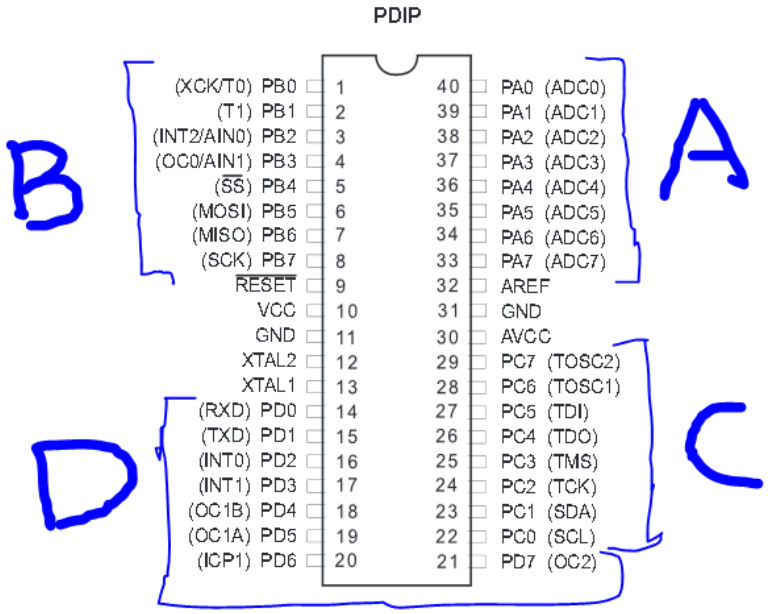
ADMUX = 0b10000000;

Note that REFS1 is the MSB of the register ADMUX. So when we assign the value to any register we write the value from MSB→LSB.

Now it's time for us to get into the real stuffs. Let's start with the basic peripheral available on any microcontroller or processor whether it is a simple ATmega 16 or one of the cutting edge processors presently available like the Intel i7. It is the GPIOs.

General Purpose Input Output (GPIO)

GPIOs are peripherals which are meant to serve the purpose of normal input output. These pins can be, for example, used to turn on or off a motor, a fan, a lamp. These pins have some inherent configurations which we need to take care of. Before going into all those details let's find the GPIOs available on ATmega 16. There are 4 8-bit GPIO ports available on ATmega 16 namely PORTA, PORTB, PORTC and PORTD.



It is clear that the each of these ports contain 8 pins each. For example, PORTA contains pins PA0…PA7. They are clubbed into 8 because the microcontroller is 8-bit. Each port is byte addressable. We will come to that later. So in total there are 32 GPIO pins in an ATmega 16.

Now let's try to understand how to control GPIOs. The following is the sequence in which a GPIO need to be configured.

- **Set a GPIO as Input/ Output**

  This is managed by configuring the register **DDRx**. Where x may be A, B, C or D. A value of '1' means that the pin is output whereas '0' means that the pin is input. All the pins are by default input pins. I have already mentioned that each port is 8 bit in size and are byte addressable. Byte addressable means that the register value can be configured only as a block. That is, we cannot access a particular pin individually. For example if I need to change the PA2 to input pin, I need to set the new value for the whole 8 pins in the PORTA.

For example consider

DDRA = 0b10110100;

The '0b' denotes that the value written is binary. This is C syntax and has nothing to do with embedded programming. The above line will configure the PORTA input/ output configuration as shown in the table below:

PIN	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
DDR value	1	0	1	1	0	1	0	0
Input / Output	Output	Input	Output	Output	Input	Output	Input	Input

I hope that setting up the DDR is clear by this time. So whether you need to change a single pin value or completely change the configuration of the port, it's the same. You have to access the whole register and edit the required parts.

Depending on whether the port/pin is configured as an input / output pin, there are two separate registers to take care of input and output.

- **What if a pin is configured as an output pin?**

  Now consider that we have set a particular pin as output pin. Is it over? Don't we have to set whether the pin should be HIGH (+5V) or LOW (0V)? This is where another register of ATmega 16, **PORTx** comes into picture. This is also a byte addressable register like the others. **PORT register is used to set any of the I/O pins to HIGH (1) or LOW (0) as required**. Now consider we write a line of code

  PORTC = 0b11000010;

  Here we wrote the number in binary. But we can write the same using hexadecimal representation also. The same definition can be written as

  PORTC = 0xC2;

  Here the '0x' represents that it is a hexadecimal representation. This can be written in the decimals also. In order to write in decimal, just convert the hexadecimal to corresponding decimal and assign it to the port. For example,

  0xC2 = 194

  PORTC = 194;

So if you do a declaration of any of the above mentioned three forms, its effect on the PORTC is shown in the table given below:

PIN	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
PORT value	1	1	0	0	0	0	1	0
Output voltage	HIGH	HIGH	LOW	LOW	LOW	LOW	HIGH	LOW

The above table also signifies that the MSB of the assigned value is assigned to PC7. So till now we discussed, what to do if a pin is set as output. What is a pin is set to input? How do we read it? Are inputs too read using some registers?

- **What if a pin is configured as an input pin?**

  Yes, there are registers to read the inputs too. Obviously these are also going to be 8-bit in width, and yes they are byte addressable. The **PINx** register serves this purpose. So how do we read it?

  char x = PINB;

  The eight bit value of the PORTB will be written into the register. This can be used to check a particular configuration or can be used to do something else. Any pin can be read regardless of whether it is set as an input pin or not in its DDR register. Till now we considered how to configure a pin as input/ output, how to read it if it is an input pin and how to set the pin HIGH/ LOW if it is an output pin. So what happens if a pin is not configured as an output pin, but we try to set it high using PORT register?

- **What if we try to use an input pin as output?**

  Well, this is not a hazard. If you try to set an input pin HIGH then that pin the pull up resistor connected to the pin gets activated. This is a special condition called Pull Up mode. A resistance of 1K comes in pull up configuration with the pin. So why do we need that configuration?

  Consider that we need to read a value from a particular pin. If the value to be read is not “driven” all the way to 5v, the pull up resistor makes sure that the value reached 5v. When the value to be read is 0v, it remains as such due to the high resistance value of 1K. So if we try to check the voltage of the pin using a multi meter, it will be 5V but it doesn’t mean that the pin is output pin. It is still an input pin which is in pull up mode.

Having discussed all the different configurations, let’s try out this problem.

Consider that the following configuration has been programmed into the microcontroller.

DDRD = 0xAA;

PORTD = 0x7E;

Now let us analyze the various states of the pins in PORTD.

PIN	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
DDR	1	0	1	0	1	0	1	0
PORT	0	1	1	1	1	1	1	0
State	Output LOW	Input Pull Up	Output HIGH	Input Pull Up	Output HIGH	Input Pull Up	Output HIGH	Input Hi impedance

With this we conclude our discussion on GPIOs. GPIOs are the simplest to understand of all the modules. Also, they are the simplest to configure. Now let's get our feet wet.

Basic Code

The code given below shows the basic structure of an embedded C code.

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

//main funtion
int main()
   {
      //declarations and initializations are written here
      //if we do a comparison between arduino code and embedded C code
      //the code you write here will be what you write in the "void setup()" in arduino IDE

      while(1)
      {
         //This is the place where you write all the code that needs to be iterated.
         //This is where whatever code you write in the "void loop()" needs to be written
      }
   }
```

#include <avr/ io.h> and **#include <util/ delay.h>** are two basic header files you need to add in all your embedded codes. The first on has all the register information whereas the second header file contains the delay function. We will discuss about them shortly.

Now let's write a code to implement the configuration given in the following table.

PIN	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Input or output	O/P	O/P	I/P	O/P	I/P	O/P	I/P	I/P
O/P VOLTAGE	0v	5v	HiZ	5v	HiZ	0v	HiZ	HiZ

Where HiZ means high impedance which is when both DDR and PORT is set to '0'

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

int main()
    {
        DDRA  = 0b11010100; //or DDRA  = 0xD4;
        PORTA = 0b01010000; //or PORTA = 0x50;
        while(1);
    }
```

Now consider that we have a sensor connected to PORTA. We need to write a code such that it waits until all eight pins are high and does something when this condition is satisfied. The code goes like this.

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

int main()
    {
        DDRA = 0x0; //we need not even clear the DDRA as
                    //any PORT can be read irresprctive of
                    //the configuration of DDRA
        while(PINA != 0xFF); //infinite loop
        //do something here
        while(1);
    }
```

I hope you understood how configure the GPIOs by this time. Now let's do an interesting real world problem. Before that I have to explain you the various delay functions available.

As it was pointed out earlier **#include <util/ delay.h>** includes the header file which contains all the delay function declarations. They are:

1) **_delay_ms**(int **delay_in_milliseconds**);
2) **_delay_us**(int **delay_in_microseconds**);

Now consider the following problem.

Lakshmi wants to implement an automatic door system which closes the door when it is found to be open. A motor is connected through a motor driver to PA7.when a high voltage is applied to PA7, the motor closes the door. A switch is connected to PC5(with all other pins in PORTC grounded) and it holds a high value whenever the door is closed. Write a code to help Lakshmi.

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

int main()
    {
        DDRA = 0x80;                    //set PA7 to output configuration
        DDRC = 0x00;                    //set the PORTC as input port
        while(1)
            {
                while(PINC != 0x20)     //checks if the door is still open
                    {
                        PORTA = 0x80;   //keeps the motor turned on
                    }
                PORTA = 0x00;           //turn the motor off
            }

    }
```

In all these codes we change the whole 8 bits. Is there any other method to edit each bit? Obviously yes, because we can do that in Arduino. So how this is done? This is done using the bitwise operations in C/C++.

Consider the following line

PORTA |= (1<<n);

This will set the n^{th} bit of PORTA HIGH without affecting the other bits. If you want to do the same for multiple pins say if you want to set pins 'n_1' and 'n_2' HIGH. This can be done as follows:

PORTA |= ((1<<n_1)|(1<<n_2));

This set pins high individually, but how to clear them. This is done as follows:

PORTA &= ~(1<<n);

PORTA &= ~((1<<n_1)| (1<<n_2));

The working can be easily understood if you know bitwise operations in C. Our objective is to make coding similar to Arduino. Simple, make these macros and keep on the top of our code. Generally we name it as sbi() for setting bit and cbi() for clearing bit. You will understand the usage once you read the following code.

```
#include <avr/io.h>
#define sbi(x,y)  x|=(1<<y)
#define cbi(x,y)  x&=~(1<<y)

int main()
    {
        sbi(DDRA, 1);   //set PA1 as output pin
        sbi(PORTA, 1);  //set the ouput value of PA1 to +5V

    }
```

AVR GCC has many macro statements in the io.h header files makes value of PA0 = 0, PD6 = 6, PC5 = 5 etc. This makes coding much simpler and obvious. Consider the following code which is highly readable.

```
#include <avr/io.h>
#define sbi(x,y)  x|=(1<<y)
#define cbi(x,y)  x&=~(1<<y)

int main()
    {
        DDRA |= (1<<PA0)|(1<<PA7); //set PA0 and PA7 as output pins
        sbi(PORTA, PA7);    //set the ouput value of PA7 to +5V

    }
```

We will use this kind of construct throughout our discussions now onwards.

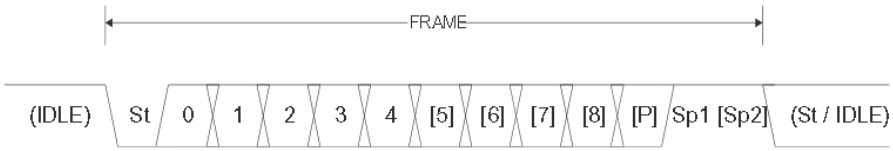
Serial monitor in the Arduino is a boon. So how to get this feature in AVR. Well, it's a bit of coding to get all the functions in place. Once we create our header file, it will be really simple.

Serial Communication using ATmega 16

Serial communication is the common term coined to USART which stands for Universal Synchronous and Asynchronous Receiver and Transmitter. This as the name indicates, send out or receives in data serially. It enables us to communicate in full duplex asynchronous mode with our computer or another hardware. The communication is done using the Tx and Rx pins on the ATmega 16.

It is evident that there is no clock line between the connected hardware. So how do we ensure that the microcontroller can serially communicate while it is a synchronous device? This is done under a mutual understanding. Both the receiver and transmitter use the same clock frequency for their UART module. This doesn't mean that the CPU frequency of the microcontrollers need to be same, but that the UART must somehow divide and generate the same frequency so that the transmitted sequence of data can be reliably received at the receiver side.

So how is the data sent? What should I do so that the data will be sent? Well even though



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.

it is tricky, once a header file is made things are simple. In order to do this we need to understand how the UART transmission is done by the hardware. The data we need to communicate must be written into an 8 bit register. This data is called data frame. The hardware adds a START BIT to the beginning of the sequence in order to synchronize the communication. The data might get corrupted during transmission and reception if the two are not in synchrony. Therefore USART module provides us an option to decide whether to use error detection algorithm called parity checking. To this frame, we add a STOP signal to denote end of transmission. The number of STOP BITS can be either one or two. The normal data frame configuration commonly used is called 8-N-1, which means 8-bit transmission with no parity and one stop bit. In this tutorial we assume the data frame that we need is of this configuration.

UART initialization

In order to send and receive data, we need to set up the hardware. This includes setting the data frame configuration, which in our case is 8-N-1 and also the baud rate. Baud rate is the rate at which data is transmitted or received. In other words, this is the clock that we just discussed about. Only two hardware set with the same baud rate can communicate. So setting the baud rate is also done during UART initialization. We need to tweak configurations in a couple of registers related to the UART module in order to get this done. The register of the UART module are discussed starting from page 163 in the ATmega 16 datasheet.

USART Baud Rate Registers – UBRRL and UBRRH

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	

USART Control and Status Register B – UCSRB

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	

USART Control and Status Register C – UCSRC

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	

The above figure shows the three registers that need to be configured to properly initialize the UART module.

- **Baud Rate generation**

  The UBRR (USART Baud Rate Register) is a 16 bit register which contains the baud rate information. How come 16 bit when ATmega 16 is 8 bit. Well, this is done by cascading two 8 bit registers namely UBRRH and UBRRL as shown in the figure given above. The normal Baud rates used for serial communication are 4800, 9600, 19200, 38400, 57600, 76800, 115200 etc. But how to find the UBRR value provided we have already decided upon the baud rate value? Well Atmel has provided the relation in the datasheet. The equation goes as follows:

$$UBRR = \frac{f_{OSC}}{16BAUD} - 1$$

So we should put the lower byte of UBRR into UBRRL and the upper byte into UBRRH. i.e.

```
UBRRL = ubrr;
UBRRH = ubrr>>8;
```

- **Enabling the transmitter and receiver module**

  The registers for this purpose can be found in the UCSRB register. Setting the TXEN HIGH will enable the transmitter, whereas setting the RXEN HIGH will turn on the receiver module. RXCIE stands for Receiver Interrupt Enable whereas TXCIE stands for Transmitter Interrupt Enable. This will help us make the code efficient by using interrupt to transmit and receive data rather than poling. So for the time being the following line will do the job for us.

```
UCSRB = 1<<TXEN | 1<< RXEN;
```

- **Setting the data frame configuration**

  According to the datasheet, we need to set the URSEL when writing into the UCSRC register. The data frame configuration is dealt with by the UCSRC register.

  UPM1 and UPM0 determines the parity information in the data. The following table shows the various possibilities.

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

  USBS is the bit that determines the number of stop bits. It is done as follows:

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

  The bits UCSZ1:0 determines the number of data bits in the data frame being transmitted. The details are given below:

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

Having all these tables in hand, it's so easy to decide the configuration that needs to be programmed into the UCSRB register. The configuration for 8-N-1 is given below:

```
UCSRC = 1<<URSEL|3<<UCSZ0;
```

Putting all these into a function will get the UART initialization function set up.

```
void usart_init(uint16_t baud)
    {
    uint16_t ubrr = F_CPU/(16*baud)-1;
    UBRRL = ubrr;
    UBRRH = ubrr>>8;
    UCSRC = 1<<URSEL|3<<UCSZ0;
    UCSRB = 1<<TXEN | 1<< RXEN;
    sei();
    }
```

Writing functions to send and receive data

Having successfully made the function to initialize the UART module, the next task is to write functions to send and receive data. For that purpose we need to discuss about some more registers.

USART Control and Status Register A – UCSRA

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	

USART I/O Data Register – UDR

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
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	

Serial communication is a full duplex communication which means that data can be transmitted and received simultaneously. We store the data that needs to be transmitted

in the UDR register. The received data is also stored in the same register UDR. But how?

Actually we have two registers with the same name UDR. When we try to read on register is used, whereas the other register is accessed when we try to write into it. This is evident from the image of the register given above. Now how do we come to know whether some data has been received? In the case of transmission of a stream of data, how do we know whether the transmitter has completed sending the previous data and that the transmitter is free and ready to send the next byte of data?

There are some bits dedicated in the UCSRA register to act as status registers for the transmitter and receiver. In the case of receiver, if a new byte of data has been completely received, then the bit RXC is set to 1. The bit is automatically cleared once the data is read from the receive buffer.

In the case of transmitter, we have another bit which acts as the status bit, namely UDRE. UDRE is set HIGH when the transmit buffer is ready to receive new data that needs to be transmitted. Now having gone through the different registers that needs to be checked during transmission and reception of data, we can write the required functions. The following are the functions that will serve our purpose:

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

void UART_writestring(char* StringPtr)
{

    while(*StringPtr != 0x00)
        {
            uart_writechar(*StringPtr);
            StringPtr++;
        }

}
unsigned char uart_readchar(void)
    {
        while( !(UCSRA & (1<<RXC)) );
        return UDR;
    }
```

With this we conclude our discussion on serial communication. But it's evident from the above code that it is not efficient. We need to wait in a loop until the status is set. This is called polling. We cannot perform any other task during this. Is there any way out? Yes. Use the Interrupt modules in the microcontroller. The implementation of receiver and transmitter using interrupt is left to you. We will discuss about the interrupt pins in the coming section.

Interrupts

This is the easiest module available in ATmega 16 to configure. Interesting isn't it? So what is this Interrupt? In the previous section when we made the functions to send and receive data, I mentioned that looping in a while loop until a particular condition is satisfied is not a good idea. We need to find an alternative for polling, and the answer is Interrupts.

When an interrupt occurs, the microcontroller pauses the normal flow of the code, and goes into a special function which contains some special code which needs to be run when an interrupt occurs. Once this special function is completely run, the normal flow of the code is resumed. That is, we no longer need to wait polling. Rather than that, we can enable interrupt somehow, so that the hardware triggers the execution of the required set of code on itself when the condition is satisfied.

Let's look at this from a different angle to make the use of interrupts much more obvious to you. Microcontrollers executes the instructions in the sequence in which the program is written. But sometimes it might be needed to handle high priority events for example power supply failure. In this case we need to take precautions to make sure all the unsaved data is stored to the memory. An interrupt can help us detect the power supply failure. This will interrupt the normal execution of the program and will execute a special function. This function is known as Interrupt Service Routine. So the Interrupt Service Routine (ISR) will take care that the data is saved and then the normal execution of the program is resumed from where the program was interrupted.

Interrupts in an ATmega can be classified largely into two:

1) Internal Interrupts
2) External Interrupts

Internal interrupts are those interrupts which are generated by some of the modules inside the microcontroller. For example, the UART module can be configured to interrupt when a new byte of data is received. Or it can be configured to interrupt when the current transmission is over and the transmitter is ready for receiving new data to communicate. Like the UART module, most of the other modules have interrupts. These interrupts are called the internal interrupts and takes care of states inside the microcontroller that we need to take care of often. Now what if we need to check for a particular condition which is external to the microcontroller. Say the power supply failure that we considered in the example above. We need to take in an input which indicates presence of power supply and generate an input when it's gone.

External interrupts come to our help here. There are four pins on the microcontroller which can be used to detect external interrupts namely, RESET, INT0, INT1, and INT2. RESET is considered an interrupt even when the program is not resumed after a reset.

Now I will explain how to configure INT0 and INT1 for our purpose. Before that as usual we need to understand the registers that controls the interrupt module.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
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	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
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	

The two registers that needs to be configured to initialize the interrupt module are

1) MCU Control Register – MCUCR
2) General Interrupt Control Register – GICR

Yeah let's do it. The following table shows the different conditions that the interrupt can be configured to get triggered at.

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

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.

ISC00:11 are in the MCUCR register. So If you want to configure the interrupt 0 to get triggered at a rising edge, the MCUCR must be configured as below:

```
MCUCR |= 1<<ISC00|1<<ISC01;
```

Setting the MCUCR isn't enough. We need to turn on the interrupt, in this case INT0 in the GICR register by setting it so that the interrupt module for INT0 is enabled.

```
GICR |= 1<<INT0;
```

Setting the INT0 enables the INT0 inside the module, but in order that we can use any interrupt in the microcontroller, we need to make sure that the global interrupt is enabled. The global interrupt bit is the 'I' bit in the Status Register (SREG) of ATmega 16. Anyway we need not set the bit like that. Calling the function **sei ()** enables global interrupt. **cli ()** is the function that can be used to clear the global interrupt in case you want to. The function to initialize the interrupt module so that both INT0 and INT1 are triggered at any logic change is shown below:

```
void int_enable()
    {
    GICR |= 1<<INT0 | 1<<INT1;
    MCUCR |= 1<<ISC00|1<<ISC10;
    sei();
    }
```

Yes we configured the interrupt module. But where do we write the code that will get run when an interrupt is triggered? We write it in the Interrupt Service Routine (ISR). Every interrupt has an address which is the address in the memory where our interrupt code is written. This is INT0_vect for INT0 and INT1_vect for INT1. So the ISR would like this.

```
ISR(INT1_vect)
   {
    //your interrupt code should be written here
   }
```

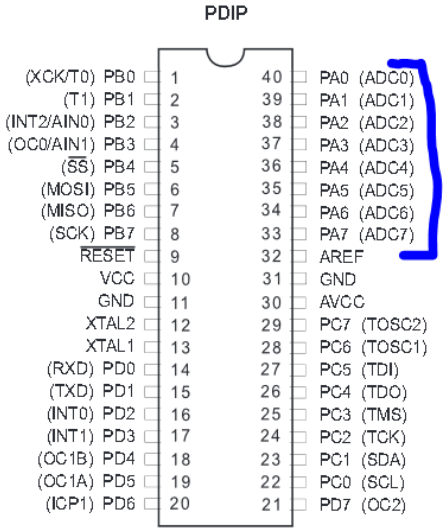
With this we conclude our discussion on interrupts. Using interrupts can really help us speed up our program. So next time when u need better performance you better try using interrupts.

Well, we discussed about how to implement serial communication, how to make our system efficient using interrupts, how to use the General Purpose Input Outputs etc. If somebody come to you and ask, "Hey do you know to read a potentiometer value using a microcontroller?" you will be like, O god this guy taught me bulls eye. In order to avoid such a comment we will discuss the ADC module in the coming section.

Analog to Digital Conversion (ADC)

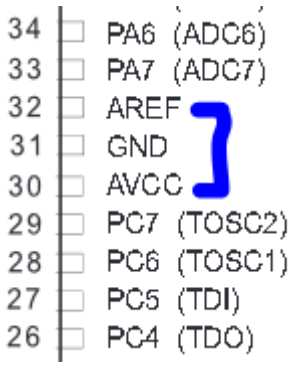
A microcontroller is a device which can only understand two logical states namely, HIGH (+5v) or LOW (0v). But all the information we wish to acquire in the real world scenario are analog, like the temperature, humidity, pressure etc. In other words, we are living in a world of analog signals. Even the ECG, EEG, all other similar bio potentials are analog in nature. So how do we acquire these signals? It is done as described below.

There are 8 pins to which ADC data can be inputted namely ADC7:0 which are multiplexed with the PA7:0 pins of ATmega 16.



Consider we have a 10 bit register. So the maximum value that we can store in a 10 bit register is $(2^{10} -1) = 1023$. We round the analog signal inputted to the microcontroller into the nearest 10 bit binary number as follows. VCC is converted to 1023, (VCC/2) to 511 and GND to 0. I hope that gives you a rough idea about the process. The ADC converter used in ATmega 16 is a 10-bit successive approximation ADC. The conversion process is worth reading. But it's not required to understand the working of ADC.

When I explained about the pin diagram of ATmega 16, I mentioned that there is a pin AVCC which is the supply for the ADC module and must be either shorted to VCC or connected to the separate analog supply line if any. Still the AVCC should be within VCC +/- 3V. Another pin which is of our interest is the AREF pin.



AREF is the analog reference pin for the A/D Converter. If you're not using the A/D, you don't need to connect AREF. So why do we need to discuss about the AREF? Because, this sets the maximum voltage that can be read using the ADC module. That is, AREF is the voltage which will be read as 1023. ATmega 16 gives us the flexibility to select the AREF. These are the possible configurations. We can either use the internal reference of AVCC or 2.56V (if we are using the internal reference, we need not connect the AREF to anything at all.) or the external reference voltage supplied at the AREF pin.

If you are not using the AREF pin, the best option is to connect it to ground through a 10nF or 100nF capacitor. In this case, we can still use the internal reference. Connecting the AREF to ground is the stupidest thing you can ever think of. If you do so, the ADC module cannot be used at all. If AREF is connected to VCC, then we cannot use the 2.56V internal reference, although the internal AVCC reference will still work.

ADC module has a clock of its own which is derived from the CPU frequency. The ADC module is slow because of the limitations of the converter. So it needs a slower clock. The ADC clock can therefore be selected by configuring the ADC prescaler. Setting the prescale value sets the division factor to be used (we will discuss it shortly). So the final ADC clock is given by

$$\text{ADC clock} = \frac{\text{CPU clock}}{\text{Division factor}}$$

Initializing the ADC

So the first thing we should do when initializing the ADC is to set the analog reference. This is configured using the ADMUX register.

ADC Multiplexer Selection Register – ADMUX

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	

REFS1:0 are the reference selection bits. The following table shows the possible states. Using these bits we can select the reference that we need, internal or external.

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

So if we need to configure the reference to AVCC, ADMUX should be configured as shown below:

```
ADMUX |= (1 << REFS0); // Set ADC reference to AVCC
```

The next thing to configure is the alignment of the result in the register. Since the data is a 10-bit number, it needs to be stored in a 16 bit register (a cascade of two 8 bit registers, in a fashion similar to the UBBR in USART module). This means that the data can be either adjusted to the left or right as per our wish. Setting the ADLAR to '1' left adjusts the result, whereas the data is right adjusted by default. I will explain this pictorially when we discuss how to read ADC value.

In this tutorial we will be using right adjusted data frame format.

MUX4:0 is used to select the analog input type and gain. The analog input can be either a single ended input or a differential input. While single ended input is the normal input that we give, differential input means that the difference in voltage between two pins will be used as the analog input voltage. This is really useful when you want to remove power supply noise from the signal. The internal amplifier can be used to amplify the input signal if the ADC is configured to differential input configuration. Why do we need a differential input? This is really helpful in case you need to read a bio potential into the microcontroller. In this case, the signal input from both the acquisition electrodes connected to our body will be having similar noise component. So if we

subtract the two we can remove the inherent noise. Wow. Well that's interesting, at least for me.

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10×
01001		ADC1	ADC0	10×
01010		ADC0	ADC0	200×
01011		ADC1	ADC0	200×
01100		ADC2	ADC2	10×
01101		ADC3	ADC2	10×
01110		ADC2	ADC2	200×
01111		ADC3	ADC2	200×
10000		ADC0	ADC1	1×
10001		ADC1	ADC1	1×
10010		ADC2	ADC1	1×
10011		ADC3	ADC1	1×
10100		ADC4	ADC1	1×
10101		ADC5	ADC1	1×
10110		ADC6	ADC1	1×
10111		ADC7	ADC1	1×
11000		ADC0	ADC2	1×
11001		ADC1	ADC2	1×
11010		ADC2	ADC2	1×
11011		ADC3	ADC2	1×
11100		ADC4	ADC2	1×
11101		ADC5	ADC2	1×
11110	1.22V (V_{BG})	N/A		
11111	0 V (GND)			

But the REFS4:0 need to be configured only when we need to select the ADC channel to read. That is while we try to read a particular pin. Therefore, we will configure it at a later stage in this tutorial while we discuss about reading the analog data.

There is one more register ADC Control and Status Register A which needs to be configured. But what's left to be configured? We haven't yet enabled the ADC module. We didn't start the ADC conversion. And also we didn't set the ADC clock frequency using the ADC prescaler.

ADC Control and Status Register A – ADCSRA

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	

Writing '1' to ADEN will enable the ADC module. By default ADC module is turned off, i.e. the ADEN is '0'. Likewise, setting the ADSC to '1' will start the ADC conversion. Please note that the converter should be explicitly turned on whenever we need to read a data. Therefore we should also set the ADSC in the adc_read () function that we are about to write.

ADIF and ADIE are needed only if we need to trigger some function when the ADC conversion is completed. Oh, I forgot to mention, the ADC conversion will take some time and therefore we need to wait until the value is converted completely by the successive approximation ADC hardware in ATmega 16. So interrupt will help us make the code much more efficient by automatically triggering the reading once the conversion is completed.

ADPS2:0 are the ADC Prescaler bits which are used to set the clock division factor which we discussed about in the introduction. It must be noted that there is a tradeoff between faster conversion and precision. This means that faster the ADC clock we use, less precise the conversion will be. So the ADC prescaler must be set depending on the precision we need for the converter.

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

We have almost completely discussed about the ADCSRA register. Now it's time to configure it for our purpose. I wish to use a division factor of 8 for my ADC.

```
ADCSRA |= 1<<ADEN|1<<ADSC|1<<ADPS1|1<<ADPS0;
```

ADIF bit in the ADCSRA register will be set when the ADC conversion is completed. Once the settings have been set, we need to wait until one conversion is successfully

completed. This is done as follows. This step in the initialization function cannot be avoided even if interrupt is used.

```
while(!(ADCSRA & 0x10));
```

Or it can be written as

```
while(!(ADCSRA & 1<<ADIF));
```

So the initialization function will look like this.

```
void adc_init(void)
    {
      ADMUX |= 1<<REFS0;
      ADCSRA |= 1<<ADEN|1<<ADSC|3<<ADPS0;
      while(!(ADCSRA & 1<<ADIF));
    }
```

It is up to you to add more function arguments so that the prescaler can be changed without editing the ADC_init () function every time you need to. Now we'll discuss about how to read the ADC value.

Reading the ADC value

You might be wondering what all we have to do in order to read from an analog value. By this time, you must have understood that a single ADC module is dealing with the 8 ADC channels. So in order to read an analog value, we must first specify which ADC channel you want to read from. This is done by configuring the MUX4:0 bits of ADMUX register. For example if you want to read from the 6th channel, ie, ADC5, we should configure the ADMUX register as shown below:

```
ADMUX |= 5;
```

After selecting the channel we need to start ADC conversion. For this, the ADSC bit needs to be set to '1'.

```
ADCSRA |= 1<<ADSC;
```

Once the conversion is started we need to wait until the conversion is completed. This can be done in either one of the following ways:

```
while(!(ADCSRA & 0x10));
```

```
while(!(ADCSRA & 1<<ADIF));
```

I hope you remember that we are using right adjusted data frame format. The result of ADC sampling is stored in the register ADC. ADC is a 16 bit register but cannot be read at one since ATmega 16 is an 8-bit microcontroller. So we need to read it from the two 8-bit registers ADC register is made of, namely, ADCL and ADCH. The alignment of data depending on the ADLAR value is shown below:

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	

The data read from the ADCL and ADCH registers can be combined as shown below:

```
uint16_t adc_value = ADCL|(ADCH<<8);
```

Now we can write the adc_read () function which can read the channel we specify as function parameter. The ADC inputs can be given to the pins PA7:0. Therefore, passing the pin name as parameter will suffice. For example,

adc_read (PA1);

Will read the ADC value from the pin PA1. The function for this is given below:

```
uint16_t adc_read(unsigned char channel)
  {
    ADMUX |= channel;
    ADCSRA |= 1<<ADSC;
    while(!(ADCSRA & 1<<ADIF));
    return ADCL|(ADCH<<8);
  }
```

Next we will discuss about the analog comparator module. Even though the names look similar, they doesn't have anything in common. Analog comparator, as the name signifies can be used for comparing two analog signals. This is quite useful when we do not want to add comparator and other analog hardware into our design.

Analog Comparator

This is one of the simplest module to configure. Tweak a single register and it's over. Interesting isn't it? It is for me. So let's see how to initialize the comparator module. Analog comparator compares the analog values inputted on two pins namely AIN0 and AIN1. AIN0 is the positive pin, whereas AIN1 is the negative input. The working of this module is similar to any comparator IC. I.e. if the input on AIN0 is higher than the analog voltage inputted on the AIN1 pin, then the ACO pin is set high. The ACO pin can be set to trigger an interrupt. The output can be also set to trigger Timer/Counter 1 Input Capture (this will be discussed later). The user can select interrupt trigger on comparator output rise, fall or toggle.

Analog Comparator Control and Status Register – ACSR

Bit	7	6	5	4	3	2	1	0	
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

Surprisingly, there is no Enable bit like most of the other modules, but it has a disable bit. ACD if set to '1' will disable analog comparator. The analog comparator is enabled by default. As mentioned in the previous paragraph, ACO is the comparator output. ACI is the interrupt flag which is set when the comparator output triggers an interrupt. ACIE is the bit which if set to '1' will activate the comparator interrupt. ACIC bit if set will enable Input Capture function of Timer 1. ACIS1:0 are the bits which need to be configured in order to determine the interrupt mode. The various possible settings are shown below:

ACIS1	ACIS0	Interrupt Mode
0	0	Comparator Interrupt on Output Toggle
0	1	Reserved
1	0	Comparator Interrupt on Falling Output Edge
1	1	Comparator Interrupt on Rising Output Edge

Now let's create a function in order to configure the analog comparator module to interrupt on rising output edge i.e. the module triggers an interrupt as the input at AIN0 starts rising above AIN1. As discussed above, we need to configure only very few bits in the register Analog Comparator Control and Status Register (ACSR). Since we are trying to enable the interrupt feature of the module, we also have to make sure that the global interrupt enable is enabled by setting the 'I' bit of SREG register. This can also be enabled by calling the **sei ()** function.

```
void analog_comp_init()
    {
        ACSR |= ((1<<ACIE)|(1<<ACIS1)|(1<<ACIS0));
        sei();
    }
```

This function will initialize the analog comparator so as to transfer the control to an Interrupt Service Routine upon an interrupt trigger. But we didn't right any ISR till now. The vector address for analog comparator interrupt routine is **ANA_COMP_vect**. The ISR can be written as follows. Add the desired operation inside the ISR so that they will be executed whenever an analog comparator interrupt even occurs.

```
ISR(ANA_COMP_vect)
{

//write the code you want to
//excecute on an interrupt here

}
```

With this we conclude our discussion on analog comparator module.

In the next section we will start some serious discussion. The module that we are going to discuss in the coming section is the most flexible module in the ATmega 16. This makes it one of the most confusing modules to configure. Regardless of all these, this module finds its application in many systems.

Timers and Counters

In this section, we will discuss about how to configure the timer/counter modules of ATmega 16. Why do we need them? Well. Timers/counters are used to generate PWM signals, for generating accurately timed pulses and for counting external events. There are three timer modules in ATmega 16, two 8-bit timer/counter modules and one 16 bit timer/counter module.

But what is the basic difference between a timer and a counter? Some of you might already know this. Timer is a module which increments its count whenever the Timer clock ticks. A counter is that which increments its count each time an external event occurs. So what is the difference between this 16 bit and 8 bit modules? Well a 16-bit Timer/Counter can count up to a maximum of $(2^{16} -1) = 65535$ where as an 8-bit Timer/Counter can only count up to $(2^8 -1) = 255$. So why do I always call it a Timer/Counter? Because the module can be used either as a timer or a counter. The only difference between a timer and a counter is that the clock for the counter is generated based on the occurrence of some external event. In order to make this discussion as simple as possible, I would like to take this forward in a sequential manner. So by this time you must have understood that the timer is counting based on some clock. So let's discuss about that first.

The timer module, like the ADC module has its own clock. The timer clock frequency depends on the prescaler value. Setting prescaler sets the timer frequency as given below:

$$\text{Timer clock} = \frac{CPU\ clock}{prescaler}$$

CPU clock may be internally generated from 1 MHz internal RC oscillator or from the external crystal oscillator connected.

The timer counts, I mentioned it before. The counting always starts from '0'. The maximum value that the counter can reach is termed as MAX. So in the case of an 8-bit timer, MAX is 255. There is another value which is set by the programmer called TOP. The value of TOP can be anything in the range [0, MAX]. When the TOP value is attained by the timer counter, then a timer output compare match is said to have happened. Based on this some things happen, which we will discuss shortly.

You need to have a basic idea about the registers before I can explain anything. So the important registers are

- TCNTx (x = 0,1,2 depending on the timer which module you're configuring)

  This is the Timer/Counter register. The counter value is stored in this register and gets incremented on each timer clock tick.

- OCRx (Output Compare Register, x = 0,1,2)

  This register is called OCR0 for Timer/Counter 0, and OCR2 in the case of Timer/ Counter 2. But there are two Output Compare Registers in the case of Timer 1 since it is a 16 bit timer, namely OCR1A and OCR1B. The TOP value I mentioned before is stored in the OCRx register by the programmer. This value being compared with the TCNTx value after each TCNTx incrementation.

- TCCRx (Timer/ Counter Control Registers)

  Similar to the OCR registers, there are TCCR0, TCCR1A, TCCR1B and TCCR2. The Compare output mode and Wave generation mode are selected using this register. Also the prescaler setting is selected using this register. In the case of the 8-bit timer/counter modules all the above mentioned configurations are in a single register, while in the case of timer/counter 1 the configuration is determined by TCCR1A and TCCR1B.

- TIMSK

  This register is called the Timer/Counter Interrupt Mask Register. This register contains the interrupt settings of all the three timers. There are two conditions of the timer which can be used to trigger an interrupt.

  - Output compare match

    An output compare match is said to have happened when the Output Compare Register (OCRx) value matches the Timer/Counter (TCNTx) value.

  - Timer/Counter overflow

    The Timer/Counter count value is said to have overflown if the TCNT value reaches MAX, which is the maximum attainable value of the counter. In this case the overflow flag is set and the counter value is reset to zero. The counter starts counting again from '0'. The timer overflow flag may trigger an interrupt if it is configured to act so.

- TIFR

  This is the Timer/Counter Interrupt Flag register and stores the various interrupt flags which are generated. They may be output compare match flags or timer overflow flags. The interrupt flags of all the three Timer/Counter modules are stored in this register.

We discussed about all the required registers. Now we are in a position to understand the various modes of operation. Due to the high flexibility in the timer configurations, there are a couple of timer modes which we can use. Depending on the timer mode, the output of the timer may also vary. The final behavior of the Timer/Counter depends on the Waveform Generation mode and Compare Output mode that we set in the TCCRx register. The various modes of operation are:

1) Normal mode
2) Clear Timer on Compare Match mode (CTC)
3) Fast PWM mode
4) Phase Correct PWM mode
5) Phase and Frequency Correct PWM mode (in Timer/Counter 1 only).

I will give you a simplified explanation about all these different modes of operation. So let's start up with the normal mode.

Normal Mode

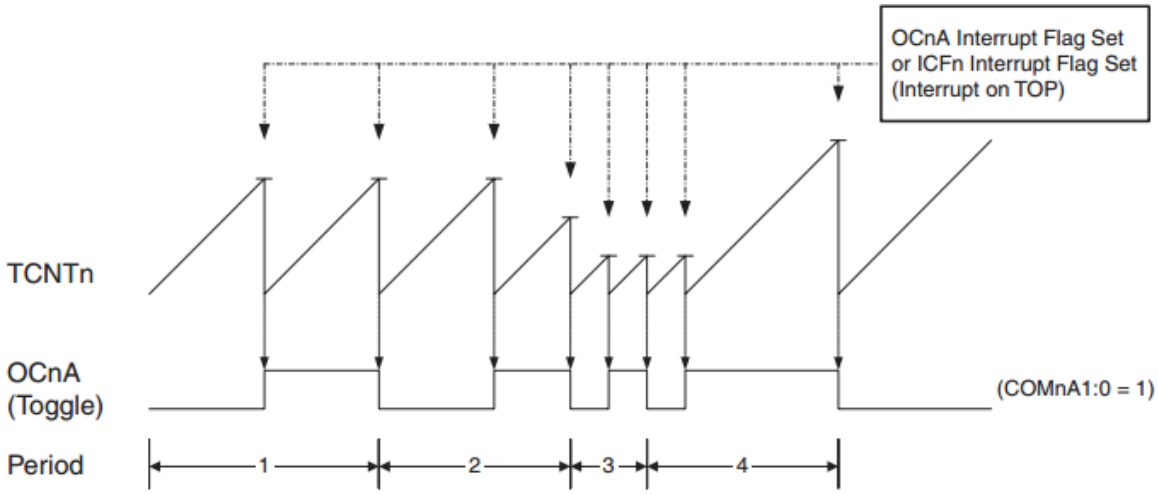
This is the simplest of all the timer modes. In this mode the TCNT simply keeps on incrementing its value at each timer clock tick until it reaches the TOP value. Once the TOP value is reached, the Timer Overflow Flag (TOVx) is set and can be used to trigger an interrupt if you wish so.

This mode can be used to trigger periodic events.

Clear Timer on Compare Match (CTC) mode

In this mode, the TCNTx value gets reset to zero when the TCNTx value becomes equal to the OCRx value. This event is called an Output Compare Match. When an output compare match occurs the OCx (output compare register) value gets changed depending on what setting you've set in the Compare Output mode bits(COMx1:0) of the TCCR register. Assume that the Compare Output mode is configured to toggle OCx value when a compare match occurs (i.e. COMx1:0 = 1). In this case a square wave is generated in the OCx pin (which is an output of ATmega 16). This means that setting the Timer/Counter to CTC mode outputs a square wave from the OCx pin of ATmega. In the case of Timer/Counter 0, the square wave is produces on OC0 pin, and so is it for Timer/Counter 2 also. But since Timer/Counter 1 is a 16-bit

timer, there are two OCR registers namely, OCR1A and OCR1B which independently can be configured to produce two square waves on the pins OC1A and OC1B of ATmega 16. The following figure shows the Timer/Counter behavior assuming that Compare Match mode (COMx1:0 = 1) is set to Toggle.



It is clear from the timing diagram that the OCx value gets toggles whenever the TCNTx value equals the OCRx value. At that instant itself the TCNTx value is reset to zero. “Oh god, are you telling me that I have to sit and find the frequency of the generated square wave? I am lazy enough to leave this job undone.” Well, no need to worry. You can find the timer output frequency using the following equation:

$$f_{OCnA} = \frac{f_{clk\_I/O}}{2 \cdot N \cdot (1 + OCRnA)}$$

Where N is the prescaler value. It can be 1, 8, 64, 256 or 1024.

Remarks: If the OCRx value is kept constant, then we can generate a 50% duty cycle square wave of desired frequency. Or in other words, this can be used as a square generator.

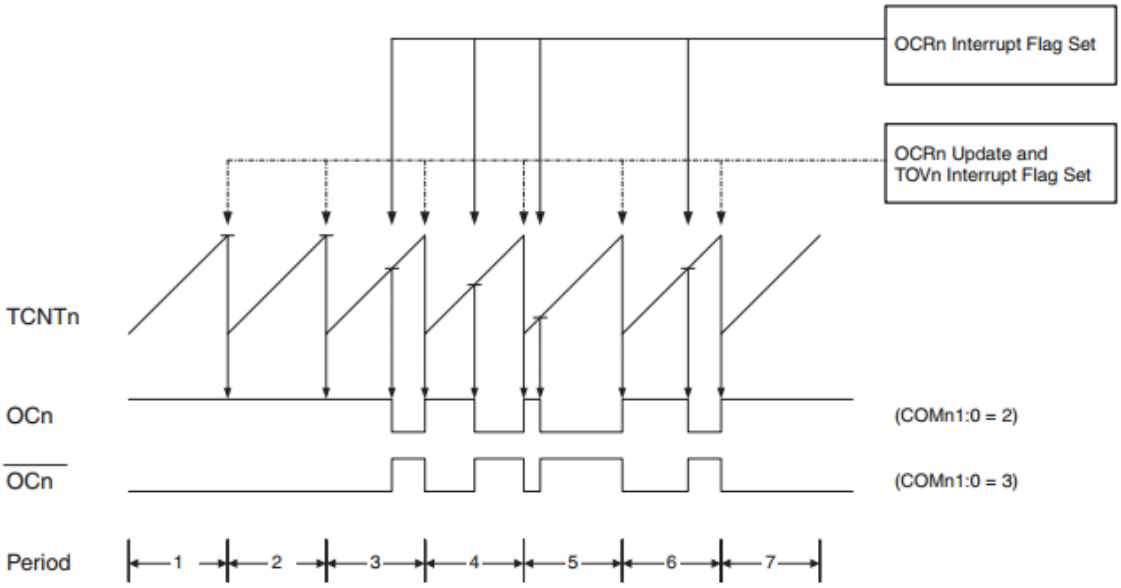
Fast PWM mode

This mode can be thought to be a combined form of Normal mode and CTC mode with slight modifications. Note some important things before we proceed. 1) TOP is no longer equal to OCRx. TOP is equal to MAX for Timer 0 and 2. Whereas we can set the TOP value in the case of Timer 1. The value that you want to set as TOP must be written to register ICR1.

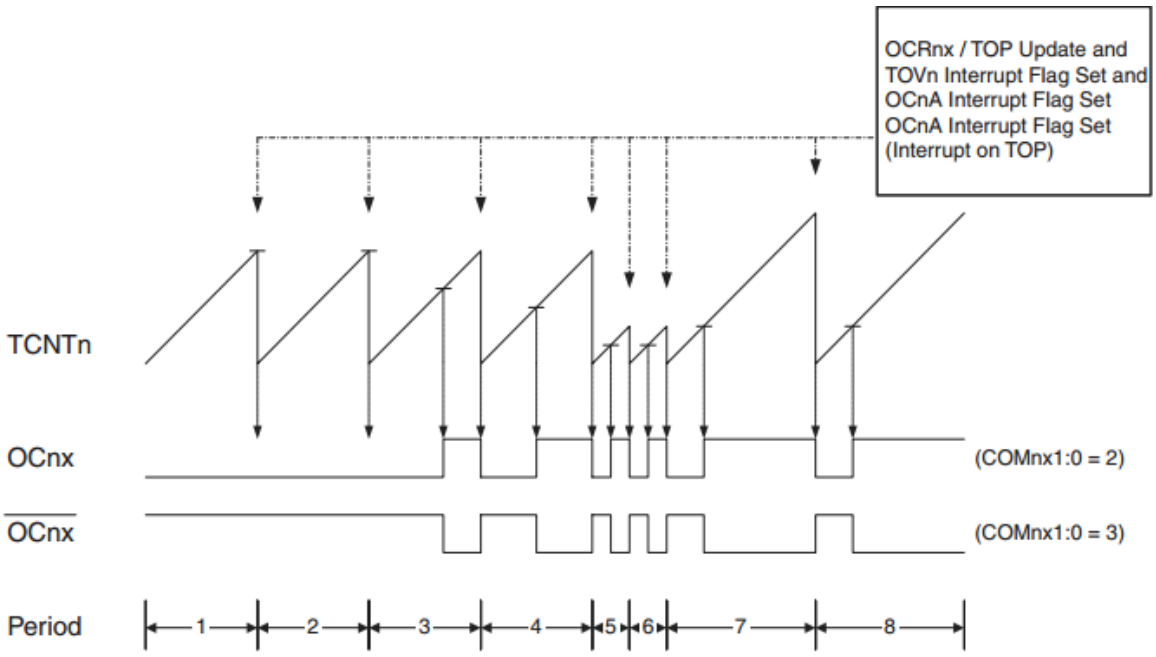
2) The TCNTx keeps on counting up to TOP, i.e. it doesn't get reset when OCRx equals TCNTx value.

3) The OCx value is changed not only during a compare match, but also when the value reaches TOP. We can decide whether the OCx value gets set/cleared when the TCNTx value reaches TOP value.

In this we can configure the Compare Output mode (COMx1:0) to toggle/set/clear OCx pin or to trigger an interrupt as you wish. In Fast PWM mode, the TCNTx do not get reset when it equals OCRx value, but it keeps on incrementing although the OCx value gets altered according to the configuration you set in the TCCRx register. The following figure shows the functioning of the Timer/Counter 0 and 2 module in Fast PWM mode:



In the above picture, Compare Output mode (COMx1:0) is set in such a way that the OCx gets set whenever an overflow occurs, and to clear the OCx pin whenever an output compare match occurs (i.e. COMx1:0 = 2). The same timing diagram in the case of Timer 1 is shown below. Please note that the OCn and its compliment shown in the figure below are wrong. You can see that in the case of Timer/Counter 1, the value till which TCNTx counts is determined by the TOP value.



Oh God, this guy is messing up everything that I understood till now (if any). Don't worry. Let's scrutinize what I just explained to you. Note that the OCx gets set whenever the TCNTx overflows. Remember that the overflow occurs periodically, only after a fixed interval of time as the MAX for a Timer/Counter is constant. This means that the OCx value is periodically 'set' in the above figure. Now depending on the TOP value, the location in the wave at which the OCx gets reset can be adjusted. In other words, we can adjust the duty cycle of the generated square wave by adjusting the TOP value. The frequency of the square wave is determined by the prescaler only. The following equations:

For Timer/Counter 0 and 2:

$$f_{OCnPWM} = \frac{f_{clk\_I/O}}{N \cdot 256}$$

For Timer/Counter 1:

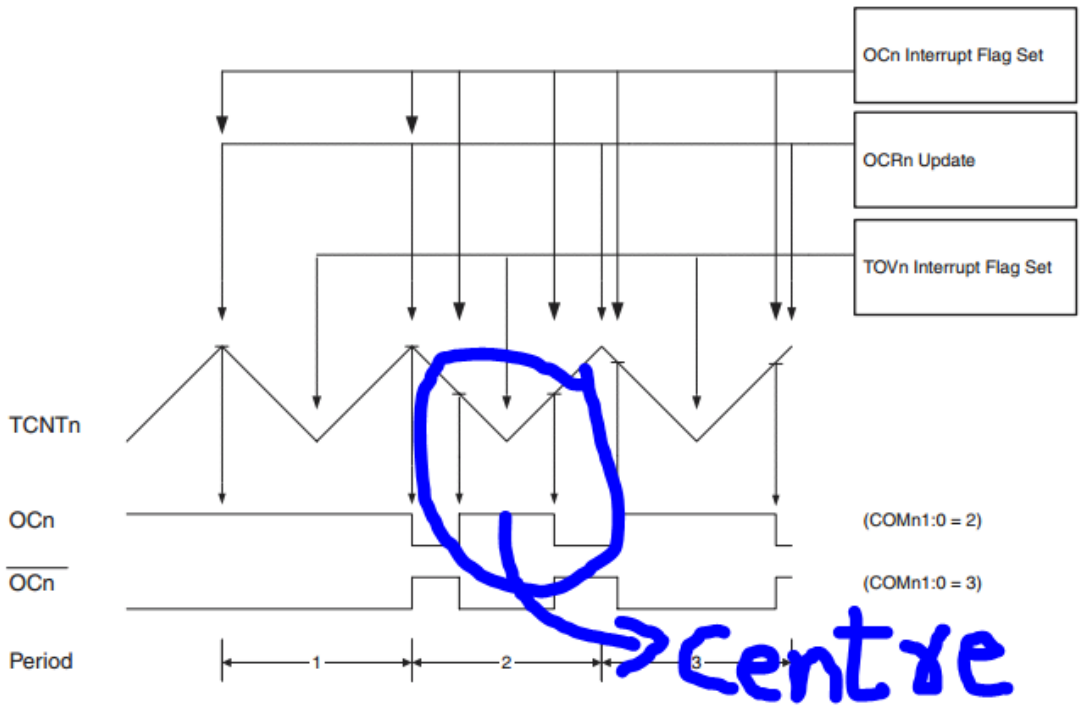
$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \cdot (1 + TOP)}$$

It is obvious why the denominator changed. In the case of Timer/Counter 0 and 2, the OCRx value is in range [0, MAX], whereas for Timer/Counter 1 it is in the range of [0, TOP] or [0, ICR1], since TOP is set using the ICR1 register for Timer 1.

Phase correct PWM mode

I know what you would be feeling now. I felt the same way when I read about it for the first time. But I hope this explanation is much simpler than the way it is presented in the Atmel datasheet. Till now we discussed about operation modes in which the TCNTx increments. Unlike the other modes, in phase correct PWM, the TCNTx counts from BOTTOM to TOP and then decrements from TOP to BOTTOM. The TOP value is set in a similar fashion to that in Fast PWM mode. Similar to the Fast PWM mode, both TOP and compare match due to OCRx alters the output OCx value. But here, unlike the case of Fast PWM, the output compare happens both during increment and decrement.

But why is it called phase correct PWM? If you look carefully, you will find that the square wave being generated has the TOP value at its center. This is the reason why it is called a phase correct PWM. Therefore, in Phase Correct PWM, the PWM channels are going to turn on or off at exactly the same time for all duty cycles. When we create multiple PWM outputs through multiple channels, this means that all the outputs will be in phase which is not possible in the case of Fast PWM. The following is the timing diagrams for Phase correct PWM in the case of Timer/Counter 0 and 2:



From the timing diagram, it is evident that the TOP value is constant and equal to MAX in the case of Timer/Counter 0 and 2. The following is the timing diagram in the case of Timer/Counter 1. It should be noted that the TOP value in

the case of Timer/Counter 1 can be changed. In other words, the frequency of the PWM can be changed in Timer/Counter 1 whereas it is a constant (not exactly as it can be varied using the prescaler) in the case of Timer/Counter 0 and 2.

The frequency of PWM can be found out using the following formulae:

For Timer/Counter 0 and 2:

$$f_{OCnPCPWM} = \frac{f_{clk\_I/O}}{N \cdot 510}$$

For Timer/Counter 1:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{2 \cdot N \cdot TOP}$$

Where N is the prescaler divider, N = 1, 8, 64, 256 or 1024.

I leave Phase and Frequency correct PWM to you as an assignment. Or in other words I am tired talking about the modes available.

Now let's take a closer look at the registers. I would advise you to go back a couple of sections and brush up whatever we discussed till now. We will proceed in the same order in which we discussed the topic. So, First the prescaler settings. I will explain the configuration for Timer/Counter 1. Configuring the same for Timer 0 and 2 are much simpler but similar. The prescaler is set using the Clock Select bits (CS12:0) for Timer/Counter 1. The same bits in Timer/Counter are be called (CS02:0). The Clock select bits are in the Timer/Counter 1 Control Register B (TCCR1B) in the case of Timer 1.

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.

The register in which this configuration needs to be set is shown below:

Timer/Counter1 Control Register B – TCCR1B

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	

WGM 13:2 along with the WGM11:0 in the register TCCR1A determines the mode of operation. The Register in which the counter value is stored is shown below (shown to make the description complete).

Timer/Counter1 – TCNT1H and TCNT1L

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	

The output Compare Registers A and B in the case of Timer 1 are shown below:

Output Compare Register 1 A – OCR1AH and OCR1AL

Bit	7	6	5	4	3	2	1	0	
	OCR1A[15:8]								OCR1AH
	OCR1A[7:0]								OCR1AL
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	

Output Compare Register 1 B – OCR1BH and OCR1BL

Bit	7	6	5	4	3	2	1	0	
	OCR1B[15:8]								OCR1BH
	OCR1B[7:0]								OCR1BL
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 explained earlier, the OCR1A and OCR1B are capable of making two independent square waves on the OC1A and OC1B respectively.

Now we'll see the register TCCR1A which holds the Compare Output mode and Waveform generation mode settings. Waveform generation mode is nothing but the different modes that we discussed before.

Timer/Counter1 Control Register A – TCCR1A

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

COM1A1:0 determines the Output Compare modes for OC1A whereas COM1B1:0 determines that for OC1B as shown below. Please note that depending on the waveform generation mode that we select, the Compare Output mode configuration changes.

Compare Output Mode, non-PWM

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)

Compare Output Mode, Fast PWM

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OCnA/OCnB disconnected.
1	0	Clear OC1A/OC1B on compare match, set OC1A/OC1B at TOP
1	1	Set OC1A/OC1B on compare match, clear OC1A/OC1B at TOP

Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 9 or 14: Toggle OCnA on Compare Match, OCnB disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on compare match when up-counting. Set OC1A/OC1B on compare match when downcounting.
1	1	Set OC1A/OC1B on compare match when up-counting. Clear OC1A/OC1B on compare match when downcounting.

Now let's see how to select the mode of operation for the Timer/Counter 1.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1X	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	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	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	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Now we'll discuss how to set up timer interrupts. The timer interrupts are set using the register TIMSK. The timer can be set to trigger an interrupt on either of the following happenings:

1) Input Capture, i.e. when ICF1 flag is set '1'.
2) Output Compare Match on A and B
3) Timer/Counter overflow

The register TIMSK is shown below:

Timer/Counter Interrupt Mask Register – TIMSK

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	

The vector name which needs to be used as parameter when an interrupt is enabled is different for each of the three interrupt configurations. They are"

TIMER2 COMP	Timer/Counter2 Compare Match
TIMER2 OVF	Timer/Counter2 Overflow
TIMER1 CAPT	Timer/Counter1 Capture Event
TIMER1 COMPA	Timer/Counter1 Compare Match A
TIMER1 COMPB	Timer/Counter1 Compare Match B
TIMER1 OVF	Timer/Counter1 Overflow
TIMER0 OVF	Timer/Counter0 Overflow

Let's do some coding before concluding this discussion.

- CTC mode

  We will make a 50% duty cycle square wave of 1Hz frequency using the CTC mode of Timer/Counter 1. We will make this output available on the 0C1A pin of the microcontroller. So we'll use 1024 prescaler to create the timer clock. Note that the clock frequency is 1MHz. So, we can find the OCR1A using the equation given in the discussion about CTC mode. For generating a 1Hz square wave, OCR1A = 487.

```
TCCR1B = 1<<CS10|1<<CS12;
OCR1A = F_CPU/(2*1024*F_WAVE)-1;
```

  Now we will configure the wave generation mode.

```
TCCR1B = 1<<WGM12;
```

  There is one more setting left, Compare Match mode. We will toggle the OC1A pin in this example.

```
TCCR1A = 1<<COM1A0;
```

  Don't forget to set the OC1A pin to output. OC1A pin is multiplexed with PD5. So let's make a function which can be used hereafter to create a square wave.

```
void sq_wave(unit16_t F_WAVE)
{
    DDRD |= 1<<PD5;
    TCCR1A = 1<<COM1A0;
    TCCR1B = 1<<CS10|1<<CS12|1<<WGM12;
    OCR1A = F_CPU/(2*1024*F_WAVE)-1;
}
```

In order to use interrupt, we have to enable the interrupt mask in TIMSK. Also don't forget to enable the global interrupt using **sei ()**.

```
TIMSK |= 1<<OCIE1A;
sei();
```

In this case, the timer function will look like this:

```
void sq_wave(unit16_t F_WAVE)
{
    DDRD |= 1<<PD5;
    TCCR1A = 1<<COM1A0;
    TCCR1B = 1<<CS10|1<<CS12|1<<WGM12;
    OCR1A = F_CPU/(2*1024*F_WAVE)-1;
    TIMSK |= 1<<OCIE1A;
    sei();
}
```

The ISR routine will look like this.

```
ISR(TIMER1_COMPA_vect)
    {
        //write your Interrupt code here
    }
```

- PWM mode

  In a similar fashion as explained for CTC mode,

```
void pwm(int dutyA, dutyB)
{
    TCCR1A = (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1);
    TCCR1B = (1 << CS10) | (1 << WGM12);
        OCR1A = dutyA*255/100;
        OCR1B = dutyB*255/100;
}
```

With this we will conclude our discussion on timers. Configuring the others are similar to this and can be done easily if you understood the tutorial.

Bibliography

1. www.atmel.in