# WHAT IS A MICROCONTROLLER? AND HOW DOES IT DIFFER FROM A MICROPROCESSOR?


**Microcontrollers vs Microprocessors**

A **microcontroller** (sometimes abbreviated **µC**, **uC** or **MCU**) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals.

A **microprocessor**(sometimes abbreviated **µP**, **uP** or **MPU**) incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC, or microchip).
Well, confused? In simple words, microcontroller is a full fledged PC in a single chip! On the other hand, microprocessor is the CPU of the PC in a single chip!

Okay, if someone asks you the configuration of your PC/laptop, how are you gonna tell them? Something like this (well, these are the configurations of my laptop… I know that it's outdated and obsolete now, but it still serves my purpose

- *Intel core2duo T6600 processor*
- *2.2GHz processing speed*
- *2 MB cache*
- *320 GB 7200 rpm hard disk*
- *4 GB DDR2 RAM*
- *512 MB dedicated ATI graphics card*
- *15" WLED screen with HD 720p resolution*
- *and other stuffs*

So, these are configurations of a full fledged laptop/PC. Now, a MCU has all these things! Surprised? Well, you must be wondering that earlier I said that MCU is a single chip, but laptop is such a huge thing. It contains so many chips, so many devices, peripherals, etc. How can a MCU accommodate all these? Well, it has all these features, but on a *lower/limited/restricted scale*. Let's take an Atmel ATMEGA32 MCU. It has the following specifications:

- *32 KB Flash memory*
- *2 KB RAM*
- *1 KB EEPROM*
- *16MHz maximum speed*
- *40 pin IC*
- *and many others that we will come across later*

So, you see, a small IC can easily accommodate these features. You can consider microprocessor to be built up of separate units, but having higher resources.

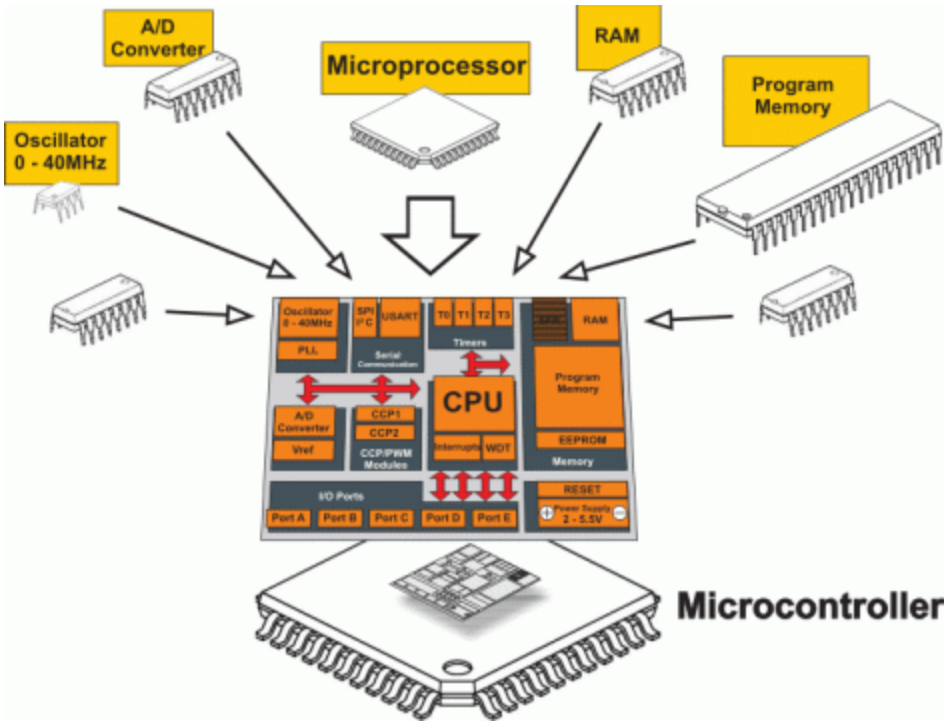
In simplest terms,

***MPU = CPU***
***MCU = MPU + Peripherals + Memory***
***Peripherals = Ports + Clock + Timers + UART/USART + ADC + DAC + LCD Drivers + Other Stuffs***
***Memory = Flash + SRAM + EPROM + EEPROM***
(Don't worry about these terms, we will discuss them when needed)

Since a PC has higher resources (from higher resources, I mean 4GHz processor instead of a 16MHz one, 4GB RAM instead of a 2KB RAM, etc), it is difficult to squeeze them into a single IC, thus we have separated units.

Thus it's all about putting more in small space. As the technology gets improved, more stuffs can be squeezed into a single IC. It all depends upon the latest VLSI Design technology.

The following diagram explains it all!




Microcontroller and Microprocessor

Now you must be thinking that if a MCU *is* a PC, then why use use MPU? At this point, I would like to bring up the following point. A MCU is usually ***application specific*** or in other words, specific to applications that don't require higher resources. For example, in a LED lighting system. Here, its absolutely worthless to implement a 2.2GHz core2duo processor, 320GB HDD, 4GB RAM, etc. A 16MHz single core processor, 16KB Flash, 1KB RAM, etc is more than enough for this application. So why waste money and resources on a MPU when we can manage it easily with a low cost MCU? But in case of a MPU, it can handle huge loads and features multitasking. You can play games in your PC, as well as browse the internet from the same PC.

Thus, we have discussed another important point that microcontrollers are usually designed to perform a small set of specific functions, for example as in the case of a Digital Signal Processor (like TMS320 by Texas Instruments) which performs a small set of signal processing functions, whereas microprocessors tend to be designed to perform a wider set of general purpose functions.

For those whom it might concern, the most MPUs are based upon von Neumann architecture (same memory is for both, program and data) whereas most MCUs are based upon Harvard architecture (separate memories for program and data). However, exceptions do exist!

**Examples** of MPU include the Intel processors (Pentium series, Dual Core, Core i7, etc), AMD processors, etc and examples of MCU include those made by Atmel (AVR), Microchip (PIC), Intel (8051), Texas Instruments (MSP430), etc.

Car Tail Light: Application of MCU

Let's have a look at few more **microcontroller applications**:

- *alarm clocks, electric toothbrush*
- *microwave ovens, automatic electric heaters*
- *traffic signals*
- *phones and mobile phones*
- *automotive – lighting, braking, speed control*
- *harddisks, pendrives, printers, mouse, cameras, small electronic machines*
- *day to day applications like washing machines, photocoping machine, elevators, etc*
- *life saving equipments like pacemaker, etc*
- *and the list is endless!*

MCU Application: Robotics & Control

Hence, we conclude that our world is too much wired, and too much full of microcontrollers! Try spending a day without microcontrollers… I bet you can't! And if you do, kindly share your experiences with me!

**Summary**

- MPU = CPU (single chip)
- MCU = MPU + Peripherals + Memory (single chip)
- MCUs are application specific whereas MPUs are designed for varied applications
- MPUs and MCUs have different architectures
- Life is incomplete without MCUs!

# BASICS OF MICROCONTROLLERS

Hello everyone! In my previous post, we analyzed the difference between a microcontroller (MCU) and a microprocessor (MPU) and some of the applications of microcontrollers. We concluded that our life would suck without microcontrollers!

So, interested in controlling these microcontrollers? And make some really cool stuff out of it? Well, for that you should be familiar with their basics and how a microcontroller development process takes place.

## Elementary Microcontroller Concepts

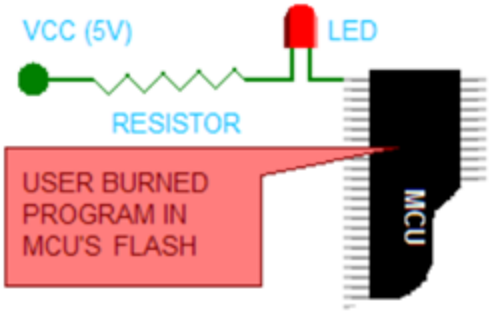
Before we start off, it's essential for you to know some elementary concepts related to MCU. To make it quick, let's codify them as follows:

- A MCU is an Integrated Circuit (IC). The number of pins, size, structure and architecture may vary depending upon the manufacturer and model.
- A combination of pins in a MCU is known as a *port*. The configuration of ports may vary from manufacturer to manufacturer.
- Every MCU has a flash memory. It is equivalent to the hard drive of a PC.
- Data can be erased and rewritten as many times as you want.
- Every MCU has a lifespan of around 1000-1500 read/write cycles. After that it becomes dead and stops working. In fact every IC has a lifetime in terms of read/write cycles. For example, pendrives usually have 10000 read/write cycles. Taking an average pendrive, it should stop working after 10-12 years.

Now that you are familiar with some basic concepts, let’s design a simple MCU based application.

# Simple MCU Application

The simplest possible MCU based solution could be blinking an LED continuously at a specified time interval. Okay, let’s design a solution for it. Don’t worry; we will be designing a theoretical solution for it. A practical design will be made later (not in this post).




Simplest MCU Process

*Problem Statement: Design a microcontroller based solution to blink an LED every 250ms.*
*Solution:*

- Let us assume our supply voltage to be 5 volts i.e. Vcc = 5V
- Connect the LED to a pin (not port, mind it! View next section for more details on ports and pins of a microcontroller) of the MCU as shown.
- The resistor is provided to prevent the LED from blowing off.
- Now, when we make it voltage of the corresponding pin to zero (0 volts) (LOW), the LED glows due to the generated potential difference.
- When we make the voltage of the pin to Vcc (5 volts) (HIGH), the LED switches off due to insufficient potential difference.
- The timing can be easily altered by introducing appropriate delays.
- The following pseudo code includes this feature. Yes, let’s code it! It’s not tough!

```
void main()
{
    InitializePort();         //Initialization
    while(1)                  //Run infinitely
    {
        PORTB = 0b00000001;  //HIGH
        Wait(0.25);          //Wait 250ms
        PORTB = 0b00000000;  //LOW
        Wait(0.25);          //Wait 250ms
    }
}
```

**Code explained:**

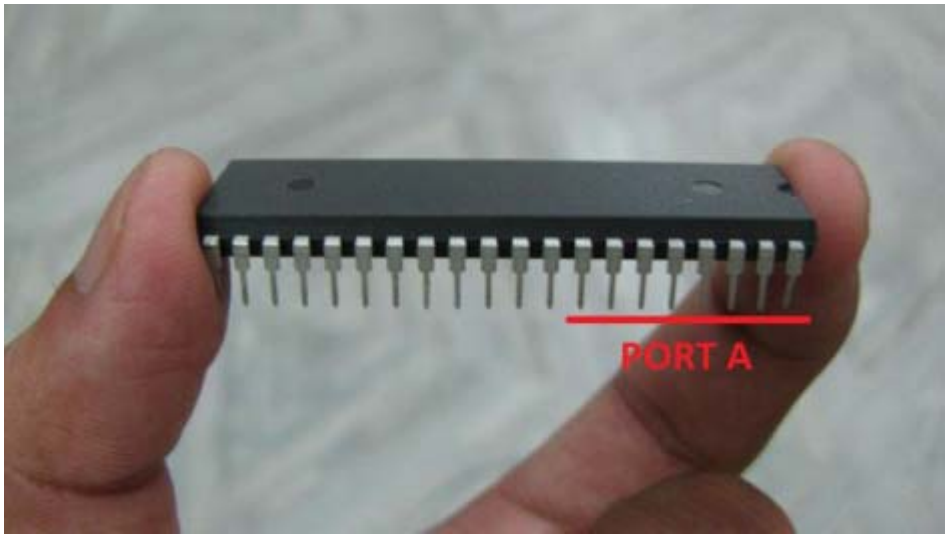
- Starting with the best part, it's written in C
- InitializePort() – this function call initializes the values of the *pins of the port*
- while(1) – '1' implies TRUE condition. While(condition) continues iterating until a false condition is met. Since the condition is *always* true, the loop is an infinite loop. Now, in C, an infinite loop is treated as a runtime error, but not in this case. This is because we want the LED to blink continuously as long as the circuit is powered. We don't want it to stop blinking after blinking for some particular number of times.
- PORTB = 0b00000001 – don't worry about this line much! We will discuss about this syntax later on. All you need to notice is the '1' in the end. Here, '1' implies HIGH i.e. 5 volts, which will switch off the LED.
- Wait(0.25) – again here, don't worry about the syntax. Just know that it introduces a delay of 250ms.
- PORTB = 0b00000000 – notice that the '1' is changed to '0' here, which means LOW i.e. 0 volts, which turns on the LED.
- Wait(0.25) – once again a 250ms delay is introduced.

Congratulations! You have successfully designed an embedded solution!

**Please note that I have assumed that you have a prior knowledge of C programming.** If not, then go to the market and buy "Let Us C" by Yeshwant P Kanetkar and start reading it.

# Ports and Pins of a MCU

- As I already said, port contains the pins of a MCU. In the adjoining figure, PORT A of an AVR microcontroller is shown. Have a close look at it and you will come to know that it consistes of 8 digital GPIO pins.




Port comprises of several pins

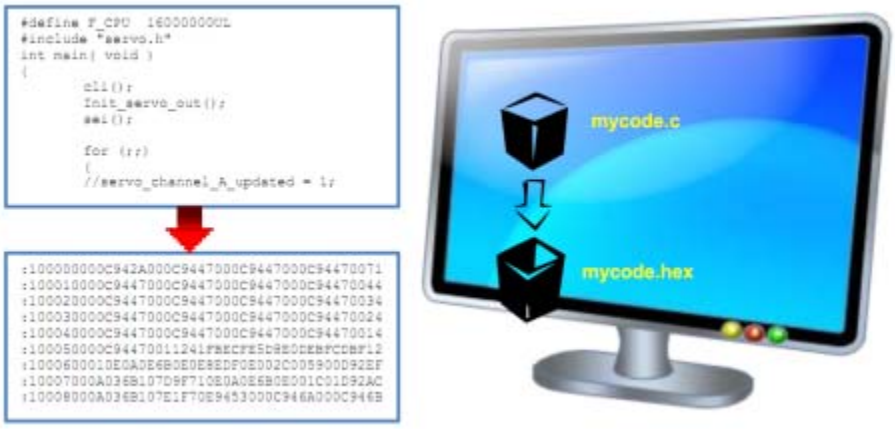
- In a MCU, most of the pins are digital GPIO pins (General Purpose Input Output pins). These digital pins can be turned on or off (or can be turned HIGH or LOW) as per the requirement.
- Not all pins are GPIO pins, there are some pins (like Vcc, GND, XTAL, etc) which are for other functions and cannot be turned on/off for interfacing.
- GPIO have two modes:
    - **Output mode:** It's quite simple. Setting it HIGH (1) gives an output of Vcc at that pin. Setting it LOW (0) gives an output of zero at that pin.
    - **Input mode:** In this mode, the MCU can read the values at the pins. Here, a threshold is defined. Threshold voltage is usually half of Vcc. If a voltage above the threshold is read, it reads it as HIGH (1) or else LOW (0). For example, suppose our supply voltage Vcc = 5V. Hence, threshold = 2.5V. When we apply a voltage 5V, it reads as HIGH (as 5 > 2.5). When we apply a voltage of 0V, it reads as LOW (as 0 < 2.5). Suppose 2V is applied. In this case, it will consider it as LOW (as 2 < 2.5).
    - The input/output mode of GPIO pins is set by DDR register (which we will discuss later).

Now that you are familiar with the basics, let's have a look at the MCU based Development Process.

# MCU Based Development Process

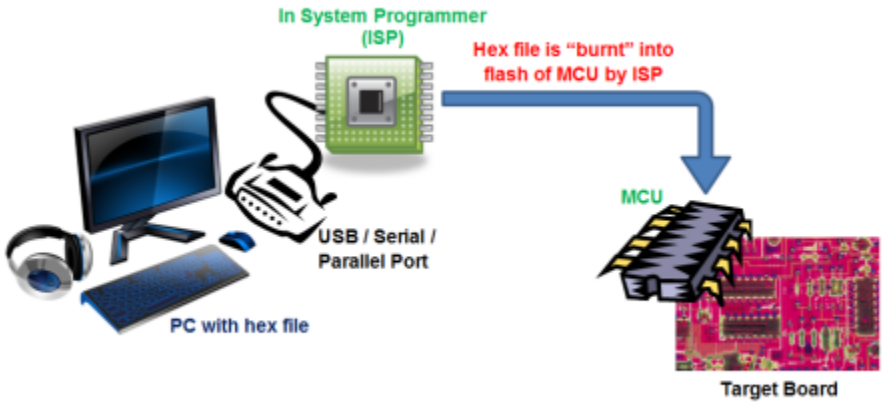
A MCU based development process consists of two simple steps:

- **Step 1:** Write a desired code for the given problem statement in a PC/laptop in an IDE, edit, compile and debug it. Say if you code in C, your source file will be something like mycode.c, which will be converted into a hex file mycode.hex.




MCU based Development Process Step 1

- **Step 2:** Open up the programmer software, locate hex file and click on “Program”. The software first reads the flash of the MCU to check the amount of data, then an erase cycle will be performed, and then your hex file will be *burnt* into the flash! But the process doesn’t end here… the file is verified so that you can be sure that your code has been transferred successfully! All this process happens with just one click!




MCU based Development Process Step 2

Don’t worry regarding the hardware shown in the diagram. We will come across them in my next post. Just concentrate on the process.

## Why use hex file?

Well, everything seems fine and perfect! But did you give it a thought why did we burn the hex file and not the C file? Well, it’s a natural fact that hex files execute much faster than c files. But let us consider the following facts:

- Suppose we use a 16MHz crystal in order to generate clock pulses. Even a substantially large C file will execute in a flash. So, why convert it to hex?
- Let’s say we program using BASIC. Still then it is converted to hex. If the BASIC files are executed faster than C files, do we need to convert it to hex?

Yes! We do! Here’s why:

- Sticking to the natural fact that hex files execute the fastest, they introduce very little delay. Using a C or BASIC file may introduce undesirable delays, which may add up and slow down the system in the end.
- Say, let us even consider the best case, where the execution time of hex file is nearly same as that of C or BASIC. This will do well for general applications. But suppose we design a solution to be installed in a car, where the MCU controls the airbag and releases it as soon as an accident happens. *This is where speed comes into play*. Your C and BASIC files will take too long to respond, and by the time they respond, the worst would have happened.

Manufacturers, at the time of manufacturing a MCU, have no idea whether that MCU will be used to blink LEDs, used to control satellites or used as target detection mechanisms in missiles. A hex code may seem useless to blink LEDs, but it’s quite valuable for the remaining two applications.

# WHAT DO WE NEED TO GET STARTED?

Hello and welcome! In my previous post Basics of Microcontrollers, we came across some of the elementary concepts and how a microcontroller based development process goes. In this post, we will see what are the things you need to get started with a MCU.

All the things you need can be categorized into two:

- Hardware
- Software

# Selecting Hardware

To get started, we must have something upon which we can work, in other words, hardware. We will be requiring the following hardware for our purpose:

- PC / Mac (preferably PC)
- In-System Programmer (ISP)
- Target Board (MCU Development Board)
- Lastly, a MCU!

## PC / Mac

We need a PC / Mac as we will be writing, editing, compiling and debugging the code here! The PC requirements are not too much. Any decent PC / laptop will do.

## Target Board (MCU Development Board)

A MCU cannot function on its own. It's not like you go to the market, buy a MCU and it starts working! You need a PCB along with an appropriate circuit to make it work. There will be slot where the MCU will fit in. It also has slots for other accessory drivers, ICs, pins' outlet of GPIO pins, etc. These PCBs are called Target Boards (or MCU Development Boards). Don't worry too much about it, there are many readymade boards available. But don't go and buy any random board. You need to consider the type of ISP and MCU (discussed next) and then choose one according to your requirement and budget.

Here, I am providing few links from where you can get a MCU development board:

- Google Shopping Results
- Thinklabs
- Embedded Market
- SparkFun
- Extreme Electronics
- Arduino

## In-System Programmer (ISP)

You have your PC at one end (with your hex file ready to be transferred to the MCU) and you have your MCU Development Board at another end. How will you transfer the hex file? For this we need another hardware called the In-System Programmer (ISP). This device *programs* the MCU by *burning* the hex file into the flash memory of the MCU. ISP is of three types:

- **Parallel Programmer**– Here, data is transferred through the parallel port (printer port) of the PC. Technically, the parallel port is called DB25 port. A DB25 connector is shown below.

DB25 Parallel Port

- **Serial Programmer**– Here, data is transferred through the serial port of the PC. Technically, the serial port is called DB9 port. A DB9 connector is shown below.


DB9 Serial Port

- **USB Programmer**– Here, data is transferred through the USB 2.0 port of the PC. A USB connector is shown below.

USB Port

In some MCU Development boards, there is an inbuilt ISP. In this case, you won't need any extra hardware for that. You can also get ISP from the links mentioned in the previous topic.

## Microcontroller


What to choose?

Selecting a MCU to work upon may be the biggest hurdle that you might face while getting started. There are more than 150 IC manufacturers which manufacture MCUs, and each one of them have more than 100 different models in their account. So, which one to choose?

On the basis of popularity, there are three series of MCUs. They are 8051 (by Intel), PIC (by Microchip) and AVR (by Atmel). 8051 is the oldest of the three, and hence quite popular, but lacks several new features and registers. And AVR is the newest of the three with the most recent architecture.
The following factors may help you choose the MCU you want to work upon.

- **Cost** – A cheap MCU is always preferred. PIC microcontrollers are comparatively cheaper than AVR microcontrollers.
- **Speed** – Speed of execution also matters. Typically, an AVR is much faster than a PIC, about 3-4 times faster!
- **Ease of use** – AVR is much more user friendly and easier to use than PIC.
- **RISC / CISC** – There are two different categories of architecture. Let's talk in layman's terms. In Reduced Intruction Chip Set (RISC) architecture, less amount of instructions will result in more output, whereas in Complex Instruction Chip Set (CISC) architecture, more amount of instructions will result in less output. In other words, *to do a particular job, RISC requires lesser number of instructions than CISC*. AVR, PIC, etc are based on RISC architecture whereas Intel 8051, Motorola 68000, etc are based on CISC architecture.
- **Free 'C' Compiler** – The best part of an AVR is that the C compiler is available for free, and that too by Atmel! Many other manufacturers do not provide with a compliler or a proprietary premium software, which we need to buy.
- **Durability** – Practically, it has been found that a PIC MCU is more durable than AVR MCU.

So, choose any MCU you want. I work with AVR and MSP430 (by Texas Instruments) microcontrollers, whereas you may prefer something else. It all depends upon your choice. But still, **most of the posts in this blog will be covered using AVR ATMEGA32 (datasheet) MCU. However, they can easily be transformed for any AVR device.**
Some of the popular AVR MCUs are:

- ATTiny 85/2313
- ATMega 8/16/32/168

P.S. For your information, MSP430 is a Mixed Signal Processing MCU by Texas Instruments. It is famous for low cost and low power applications. MSP430 series consume the lowest power of all the MCUs!

# Selecting Software

AVR Studio 4 Logo

- Talking about AVR MCUs, any platform (Windows/Linux/Mac) will do. There are different compilers available for different OS.
- Next, choose the language in which you want to program your MCU. C is preferred over BASIC as it is more organized and gives you the power to do substantially more! Apart from this, C is a language which (ideally speaking) every engineer knows! And its ubiquitous. However, you can also use BASIC, I won't force you to use C .
- Next, choose a compiler. For BASIC programmers, the best compiler is BASCOM. For C programmers, best compiler is AVR Studio. CodeVisionAVR is another popular software, but it's not free.
- And at last, you need a programmer software. Well, to speak of it, most of the compilers (including AVR Studio, CodeVisionAVR and BASCOM) have an inbuilt programmer. But I prefer to use a separate one for greater flexibility. Some popular programmers are PonyProg (serial), avrdude and FreeISP (GUI for avrdude).

# AVR Basics

Hello everyone! From now on, we will be working with AVR microcontrollers. I hope you have an AVR MCU development board and an ISP with you. And I also assume that you have the elementary concepts of a MCU. If not, then I suggest you to go through the following posts:

- Difference between a microcontroller and microprocessor
- Basics of microcontroller
- What do we need to get started?

Now that you have a development board and an ISP with you, I also hope that you have a tested AVR MCU in that. This is because new AVR MCUs need to be initialised. In technical terms, we need to *set* the *fuse bits*, or else it won't work. We will discuss this thing later as it requires some skill which you will be getting now. For now, let's get into the basics of AVR.

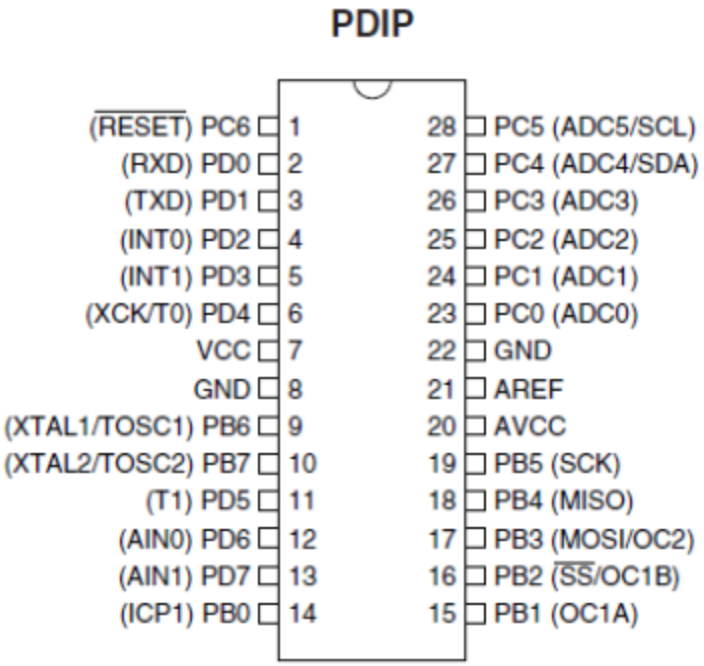
# Datasheet

A datasheet of any electronic item/component/device is a document in which the manufacturer gives information regarding the product to its users. The data given in it is in detail. All the features, technical specs, design, register Summary, expected usage, troubleshooting, pin details, etc, everything is given in detail. It is the best source of info for that particular device!! All electronic components have a datasheet published by the manufacturers. You can Google them out!! They are free. Check out the datasheet of the following AVR MCUs:

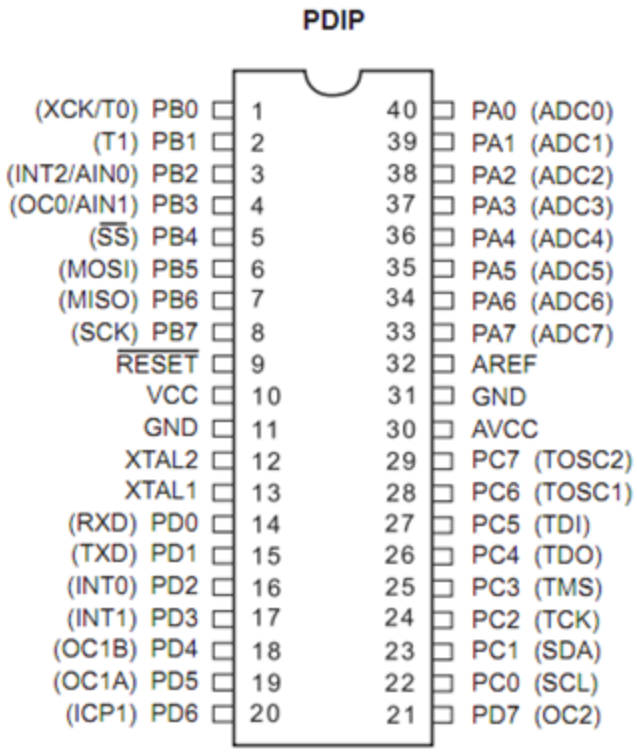
- ATTINY2313
- ATMEGA8
- ATMEGA16
- ATMEGA32
- ATMEGA168

# Pin Configuration

The pin configurations of an ATMEGA8 and ATMEGA16/32 MCU are shown below.




ATMEGA8 Pin Configuration

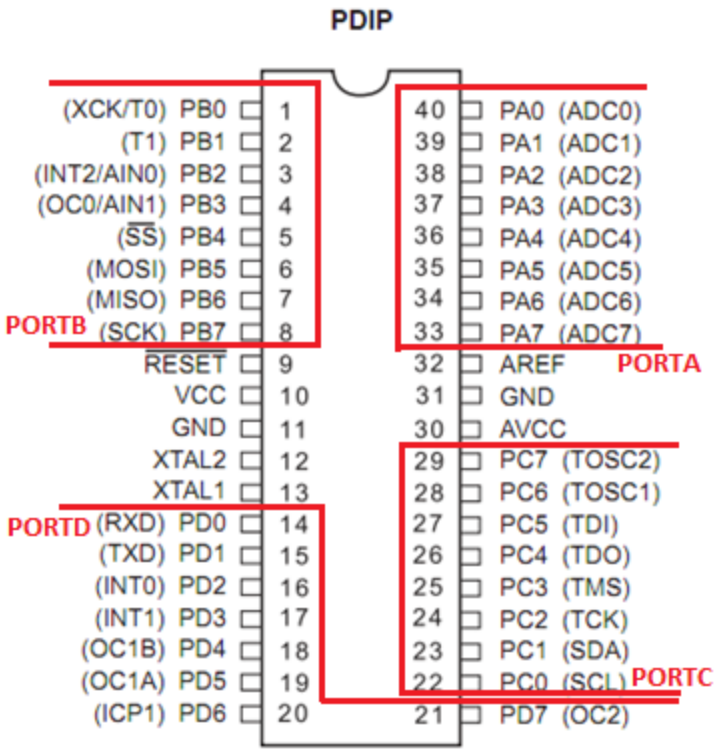



ATMEGA16/32 Pin Configuration

Please note that we will be dealing with DIP packages. For more information of types of IC Packages, visit this site.

Now, let's analyze the pin configuration of ATMEGA16/32. First of all, note that the ATMEGA16 and ATMEGA32 MCUs are completely similar, except the fact that ATMEGA16 has 16KB Flash, 1KB RAM and 512B EEPROM whereas ATMEGA32 has 32KB Flash, 2KB RAM and 1KB EEPROM.

Hey, don't get scared after looking at the pin configurations. The MCU has 40 pins. It's not that complicated. Here, let me simplify it as follows.




ATMEGA32 Pin Configuration Simplified

So here, as you can see, I have split the pins into four major parts. Remember about ports? I said that ports contain the pins of a MCU. ATMEGA32 has four ports, PORTA, PORTB, PORTC and PORTD, each one having 8 pins. Don't worry about the other details, we will discuss them as and when needed. Right now, concentrate on PA0…PA7, PB0…PB7, PC0…PC7 and PD0…PD7. These are the four ports that make up the GPIO pins (General Purpose Input Output). These concepts are discusses in my previous post, Basics of Microcontroller.

So, how many pins over? 4 × 8 = 32. How many remain? 40 – 32 = 8! The remaining 8 are mostly consumed by supply pins (VCC, AVCC), ground (GND), reset, XTAL pins, and other minor stuffs. So, all the 40 pins over, isn't it? 😉

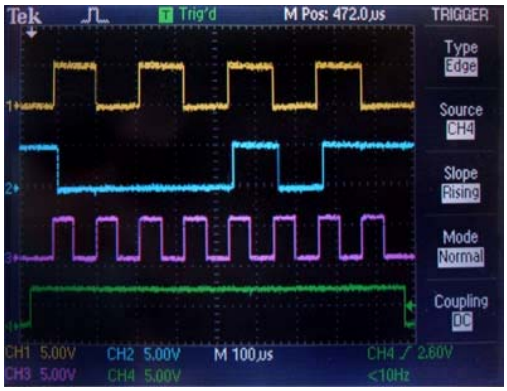
P.S. Every MCU has an internal oscillator which determines its frequency of oscillator. But we need not stick to it. We can connect an external crystal oscillator to generate higher frequencies

and clock pulses. This external oscillator is connected across the XTAL pins (XTAL1 and XTAL2).

# AVR Peripherals

Okay, now have a look again at the pin configuration of ATMEGA32. Have a look at all the GPIO pins of all the ports. Can you see some things written in the brackets? Like PA0 (ADC0), PB5 (MOSI), PC2 (TCK), PD1 (TXD), etc. Well, these are the *extra* features that the MCU can offer you apart from GPIO. In other words, these pins show dual behavior. If nothing is specified, they act as GPIO pins. The secondary features of these pins become active only if you enable certain bits of some registers. These are called peripherals. There are several peripherals that AVR offers in ATMEGA32, some are as follows:

- **ADC** – Analog to Digital Converter – usually 10-12 bit
- **Timers** – 8 bit and 16 bit timers
- **JTAG** – Joint Test Action Group
- **TWI** – Two Wire Interface (or) **I2C** – Inter-Integrated Circuit
- **USART** – Universal Synchronous Asynchronous Receiver Transmitter
- **UART** – Universal Asynchronous Receiver Transmitter
- **SPI** – Serial Peripheral Interface
- **WDT**– Watchdog Timer …and many more!




SPI Communication using UART

You can get a complete list from the datasheet. Let me give you brief idea regarding them. Detailed information will be given in later posts. For now, the following is enough.

ADC stands for Analog to Digital Conversion. The term is self-explaining. This feature converts Analog signals into Digital signals.

Timers are something which are a consequence of clock frequency. We can manipulate the clock pulses in order to generate timers of required resolution.

JTAG corresponds to testing of the circuit. When we make a circuit and fix the MCU onto it, we use JTAG to verify whether our connection, soldering, circuit design, etc is correct or not.

TWI/I2C (its actually I-square-C) is a revolutionary technology by Philips, in which two devices are connected and communicate by using two wires! USART/UART is related to serial communication in which the MCU sends and receives signals from another device based on serial protocol. SPI is something which helps to interface I2C, UART, etc. The adjoining figure showing SPI Communication using UART is just for representation purpose.

WDT (Watch Dog Timer) is something interesting it seems. The name is also quite interesting. Just like a watchdog always keeps an eye on it’s master to protect him from any harm, WDT keeps an eye on the execution of the code to protect the MCU from any harm. A WDT is a computer hardware or software timer that triggers a system reset or other corrective action if the main program, due to some fault condition, such as a hang, neglects to regularly service the watchdog (writing a “service pulse” to it, also referred to as “kicking the dog”, “petting the dog”, “feeding the watchdog” or “waking the watchdog”). The intention is to bring the system back from the unresponsive state into normal operation.

# I/O Port Operations in AVR

Hello friends! In this post, we will discuss about the port operations in AVR. Before going further, I suggest that you read my previous post regarding AVR Basics. **The examples discussed here are in accordance with ATMEGA16/32 MCU. However, the concepts are equally good for any AVR MCU**.

## Register

Okay, now I hope you are familiar with the term register. If not, then you must have heard of it. Basically, a processor register is a memory space within the CPU itself so that they can be accessed very frequently and fast. These registers are linked with the operation of the MCU. Let's consider the following memory space.


Register Memory Space

Here, you can see that I have represented 8 bits together to form a memory of 1 byte. Note the sequence in which the bits are numbered. They are numbered as 7, 6, 5, … , 1, 0. This is because the bits are numbered from the **Least Significant Bit (LSB)** to the **Most Significant Bit (MSB)**. From the knowledge of digital logic, we know that the last bit is the LSB whereas the first bit is the MSB. Hence, **Bit 0 = LSB** and **Bit 7 = MSB**.

**Register Concept Explained**

Let me explain you why LSB is the last bit. Let's take an example. Please note that 0b stands for binary and 0x stands for hexadecimal. If nothing is prefixed, it means that it is in decimal number system.

A = 0b 0110 0111 = 103

Now here, let's change the value of the last bit (orange colour bit) from 1 to 0. This makes

B = 0b 0110 0110 = 102

Now, once again in A, let's change the first bit (magenta colour bit) from 0 to 1. This makes

C = 0b 1110 0111 = 231

We see that by changing the last bit, the result (B) is very close to the original data (A), whereas by changing the first bit, the result (C) varies quite significantly from the original data (A). Hence, the last bit is the LSB (as the data doesn't change significantly) whereas the first bit is the MSB (as the data changes significantly).

Now, we also know that 1 nibble = 4 bits. Hence, bits 0,1,2,3 are called **lower nibble** whereas bits 4,5,6,7 are called **upper nibble**. So basically, a register is a memory allocated in the CPU, usually having a size of 1 byte (8 bits).

Next, every register has a name, and every bit of it also has a name. Take the following example.

ADMUX Register

| Bit No. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Name | REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Upper Nibble Lower Nibble

ADMUX Register

Here, the name of the register is ADMUX (don't worry about the name, we will discuss it later). Also, note that each bit also has a name and an initial value. Each bit of the register can have a value of either 0 or 1 (initially it is 0). Now suppose, I write

```
ADMUX = 0b01000111;
```

This means that the ADMUX register has been updated as follows:


ADMUX Register after setting values

This can also be achieved by the following codes:

```
ADMUX = (1<<REFS0)|(1<<MUX2)|(1<<MUX1)|(1<<MUX0);
ADMUX = 0x47;    //Hex form
```

So, got an idea of what registers are and how are they defined/initialized? We will discuss about various registers one by one as required. Right now, we are concerned with only three registers, DDR, PIN and PORT.

# Port Operation Registers

The following registers are related to the various port operations that we can perform with the GPIO pins.

- DDRx – Data Direction Register
- PORTx – Pin Output Register
- PINx – Pin Input Register

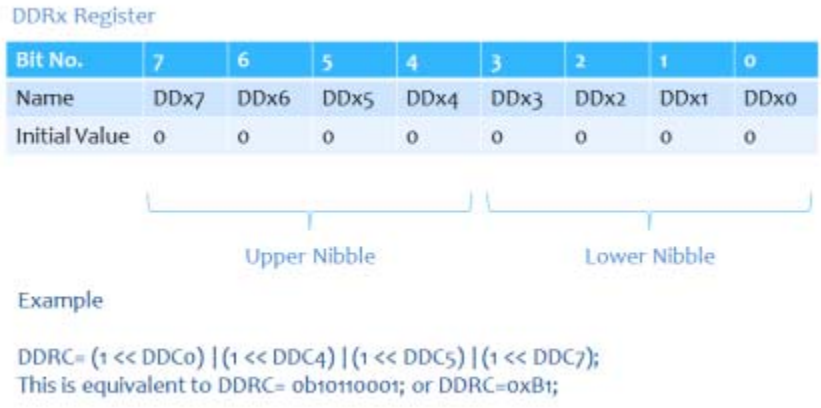
where x = GPIO port name (A, B, C or D)


Different Port Operations

**DDRx Register**

As I mentioned earlier, the GPIO pins are the digital I/O pins i.e. they can act as both input and output. Now, how do we know that the pin is an output pin or input? The DDRx (Data Direction Register) helps in it.

DDRx initializes the port. Have a look at it's bit structure.




DDRx Register

The ‘x’ in DDRx is just for representation. It is replaced by the corresponding port i.e. x = A, B, C, D. Say for the example shown in the diagram above

```
DDRC = (1<<DDC0)|(1<<DDC4)|(1<<DDC5)|(1<<DDC7);
```

This is equivalent to

```
DDRC = 0b10110001;
```

and as well as

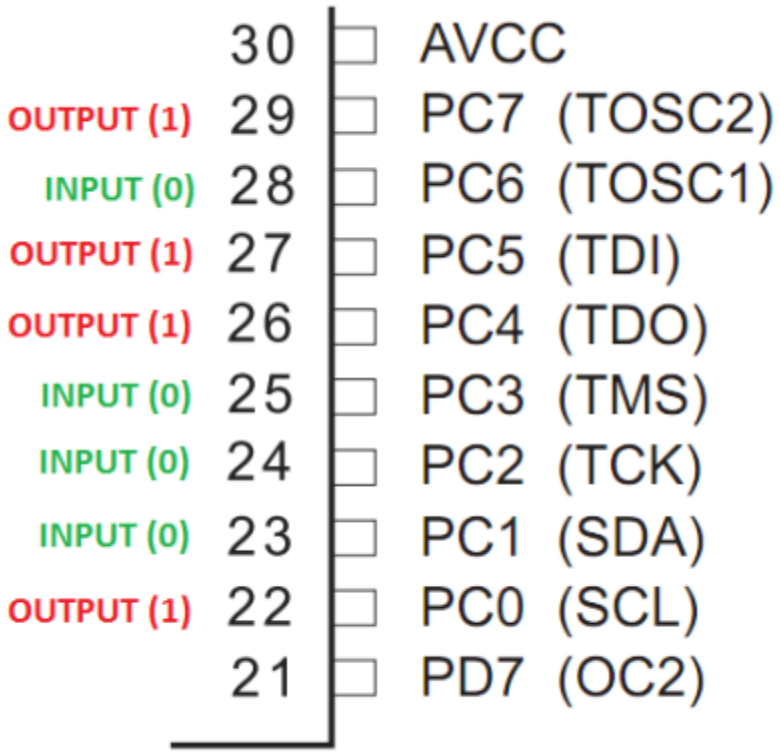
```
DDRC = 0xB1;
```

and even

```
DDRC = (1<<0)|(1<<4)|(1<<5)|(1<<7);
```

So, did you get how to declare it? Suppose I would like to initialize my port B, then I would have written

```
DDRB = (1<<DDB0)|(1<<DDB4)|(1<<DDB5)|(1<<DDB7);
```




DDRC Example

All right, now that we are done with the declaration, let me explain you what it does. Always remember, in the case of DDRx, **1 stands for output and 0 stands for input**. In the following statement (given below), port C is initialized such that the pins PC0, PC4, PC5 and PC7 are output pins whereas pins PC1, PC2, PC3 and PC6 are input pins.

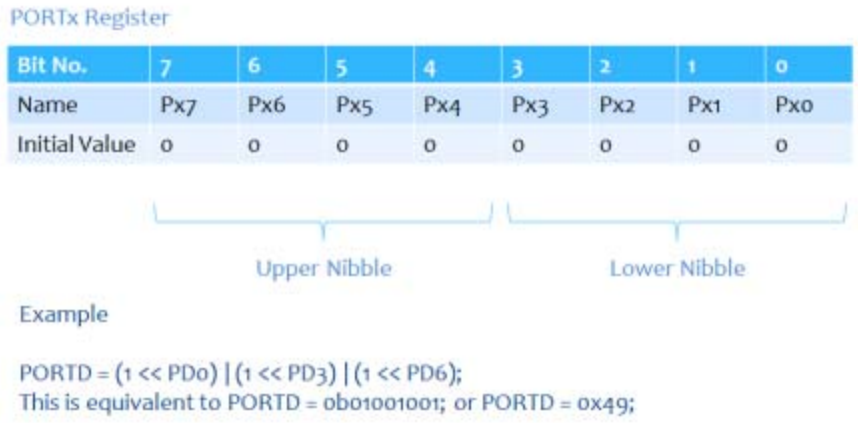
This is represented in the adjoining figure. The pins marked as input now has the capability to *read* the voltage level at that pin and then treat it as HIGH if it is above the threshold level, or else it will treat it as LOW. Generally, the threshold level is half the VCC.

Similarily, the pins marked as output have the capability to either *become* HIGH (voltage = VCC) or LOW (voltage = zero) as directed/written in the code.

```
DDRC = (1<<DDC0)|(1<<DDC4)|(1<<DDC5)|(1<<DDC7);
```

**PORTx Register**

The PORTx register determines whether the output should be HIGH or LOW of the output pins. In simplified terms, once the DDRx register has defined the output pins, we need to set them to give an output of HIGH or LOW. The PORTx register goes as follows.




PORTx Register

The register declaration is similar to the DDRx register, except that we change the names, that's all! One such example is given above in the diagram. ***The following declarations are one and the same***.

```
PORTD = (1 << PD0)|(1 << PD3)|(1 << PD6);
PORTD = (1 << 0)|(1 << 3)|(1 << 6);
PORTD = 0b01001001;
PORTD = 0x49;
```

Now, let's consider the following statements:

```
DDRC   = 0b10110001;
PORTC  = 0b10010001;
OUTPUT = 0b10010001;  /*This is not a C executable line, this line is just
for explanation*/
```

The port C is initialized using the DDRx register. The highlighted bits correspond to the output pins. Now, just concentrate on the highlighted bits only. Since they are output pins, wherever I state ’1′ in PORTC, that pin goes HIGH (1), giving an output voltage of VCC at that pin.

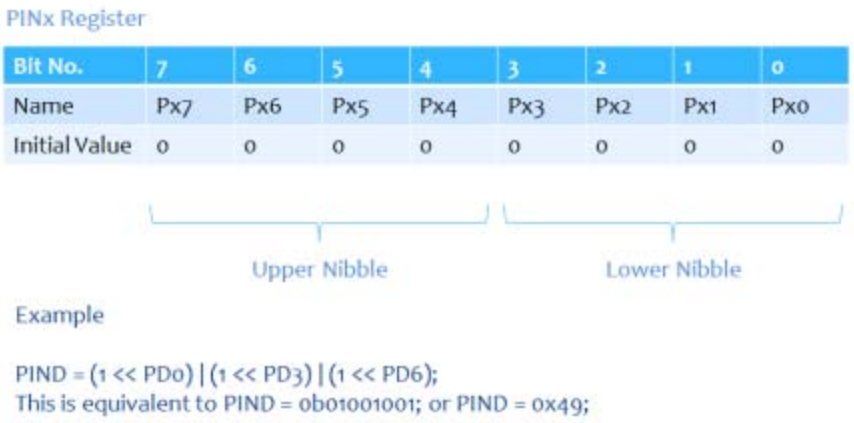
Now consider the following set of statements:

```
DDRC   = 0b10110001;
PORTC  = 0b10010101;
OUTPUT = 0b10010001;  /*This is not a C executable line, this line is just
for explanation*/
```

Once again, the bits highlighted in orange correspond to the output pins. So, whatever value (0 or 1) desired in the orange region is reflected in the output. Now, look at the magenta highlighted bit. Inspite of being set as HIGH in the PORTx register, the output is LOW. This is because that pin is initialized as an input pin by DDRx. **Hence, PORTx cannot change the properties of that pin. Hence, in general, PORTx cannot modify the properties of a pin which is initialized as input by DDRx**.

**PINx Register**

The PINx register gets the reading from the input pins of the MCU. The register goes as follows:



PINx Register

The register declaration procedure stands the same. Also note that the names of the bits of PORTx and PINx registers are same.

Now, let's consider the following statements:

```
DDRC   = 0b10110001;
PINC   = 0b01001011;
INPUT  = 0b01001011;  /*This is not a C executable line, this line is just
for explanation*/
```

Here, the highlighted bits correspond to the pins that are initialized as input by the DDRx. In the second line, the PINx register is defined. Well, this line is just to explain the concept, practically, we always use PINx as a condition (like in IF or in WHILE loop). As per the second statement, the PINx command reads the values only at the input pins.

Now, consider the following set of statements:

```
DDRC   = 0b10110001;
PINC   = 0b01011010;
INPUT  = 0b01001010;  /*This is not a C executable line, this line is just
for explanation*/
```

Here, you can compare it with the example I gave for PORTx. Since the magenta-highlighted bit is an output pin, PINx cannot change it's properties. **Hence, in general, PINx cannot modify the properties of a pin which is initialized as output by DDRx and vice versa**.

# Example Code Snippet

Let's demonstrate the use of the DDRx, PORTx and PINx registers from the following code snippet:

```
DDRC = 0x0F;
PORTC = 0x0C;

// lets assume a 4V supply comes to PORTC.6 and Vcc = 5V
if (PINC == 0b01000000)
PORTC = 0x0B;
else
PORTC = 0x00;
```

**Code Explained:**

- DDRC = 0x0F; is equivalent to DDRC = 0b00001111; This means that the pins PC0…PC3 are output pins (can be manipulated using PORTC) and pins PC4…PC7 are input pins (whose levels determine the value of PINC).
- PORTC = 0x0C; is equivalent to PORTC = 0b00001100; This means that the pins PC2 and PC3 have a HIGH voltage (Vcc = 5V) and pins PC0 and PC1 have LOW voltage (0V). The other pins have low voltage by default.
- if (PINC = 0b01000000) checks the input voltage at pin PC6. Since it is mentioned in the comment that a 4V is supplied to PORTC.6 (same as pin PC6), this condition is true (as 4 > 2.5, where 2.5V is the threshold, 5/2 = 2.5).
- Since the if condition is true, PORTC = 0x0B; is executed.
- If the if condition is not satisfied, PORTC = 0×00; will be executed.

We can also put it inside a while loop to run it continuously i.e. it always checks for the voltage at pin PC6, and the outputs at PC0, PC1 and PC3 go high only if the voltage at PC6 is greater than 4V.

```
DDRC = 0x0F;
while(1)
{
    // a condition of 4V supply to PORTC.6 and Vcc = 5V
    if (PINC == 0b01000000)
        PORTC = 0x0B;
    else
        PORTC = 0x00;
}
```

To learn how to code and simulate using AVR Studio 5, visit this page.

So, here we end up with the port operations that can be performed with an AVR MCU. Though the concept has been explained using ATMEGA16/32, the concepts are equally good for any AVR MCU! Just go through the datasheet in order to get an idea of the registers.

# AVR C Programming Basics

In this chapter you will explore some of the key concepts of C programming for AVR microcontrollers.



## 3.1 AVR Registers

A **register** is a special storage space in which the state of the bits in the register have some special meaning to the AVR microcontroller. Most of the registers are 8-bits wide (there are a few exceptions).

Each register has a name and individual bits may also have unique names. All of the register and bit names are described in the ATmega48/88/168/328 datasheet. The AVR Libc library will define identifiers for the register names and bit names so that they can be easily accessed in C.

Manipulating the bits in the various registers of the AVR is the basis for AVR programming. Everything from configuring the AVR device's built-in peripherals to using the AVR's pins for digital I/O is done by manipulating bits in these registers.

## 3.2 Bits and Bytes

A **bit** represents one of two possible states: 1 or 0 (aka: on/off, set/clear, high/low). Several bits together represent numerical values in binary, where each bit is one binary digit. An AVR microcontroller groups 8 bits together to form one **byte** wtih the **L**east **S**ignificant **B**it (LSB) on the right. Each bit is numbered, starting from 0, at the LSB.

Consider the decimal number `15` for example. `15` represented in 8-bit binary is `00001111`. The 4 least significant bits 0, 1, 2, and 3, are set.

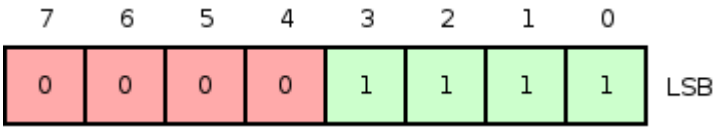



Figure 3.1 - The decimal number 15 represented in 8 bits

If you do not already know how to convert numbers between decimal, binary, and hexadecimal, you can find numerous lessons online or use my Printable Decimal-Binary-Hex Conversion Chart

Another example, the decimal number `40` is represented in binary as `00101000`. Bits 3 and 5 are set.

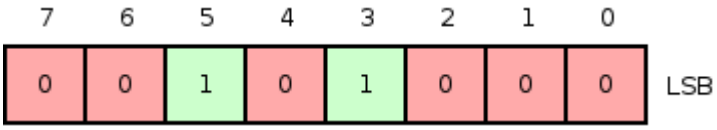



Figure 3.2 - The decimal number 40 represented in 8 bits

Numerical values in a C program for an AVR microcontroller may be defined using a decimal, hexadecimal, or binary notation depending on the context and the programmer's preference. A hexadecmial number is defined using the `0x` prefix and a binary number is defined using the `0b` prefix.

The following C code shows 3 ways in which a variable might be initialized to the decimal value of 15.

```
uint8_t a = 15;         /* decimal */
uint8_t b = 0x0F;       /* hexidecimal */
uint8_t c = 0b00001111; /* binary */
```

The `uint8_t` data type is one of the fixed width integer types from the C99 standard. It defines an 8-bit unsigned integer. The C99 style data types will be used throughout this tutorial series.

## 3.3 Bitwise Operations

Since individual bits often have a significant meaning when programming AVR microcontrollers, bitwise operations are very important.

A **bitwise AND** operation results in bits being set only if the same bits are set in both of the operands. In other words: bit *n* will be set in the result if bit *n* is set in the first operand **and** the second operand.

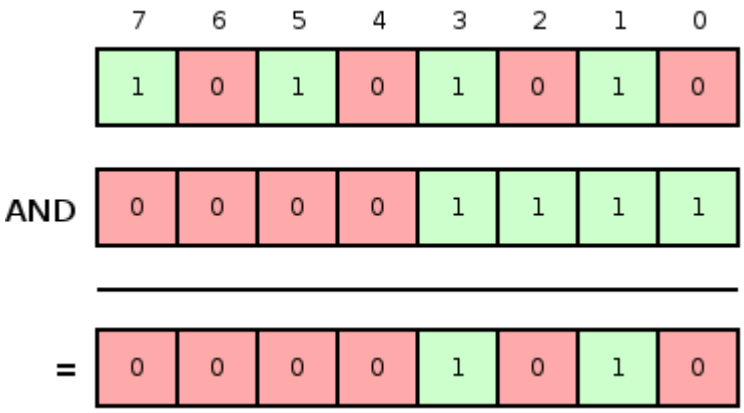



Figure 3.3 - Bitwise AND operation

A single ampersand character (`&`) is the bitwise AND operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a & b;  /* 00001010 */
```

A **bitwise OR** operation results in bits being set if the same bits are set in either of the operands. In other words: bit *n* will be set in the result if bit *n* is set in the first operand **or** the second operand.

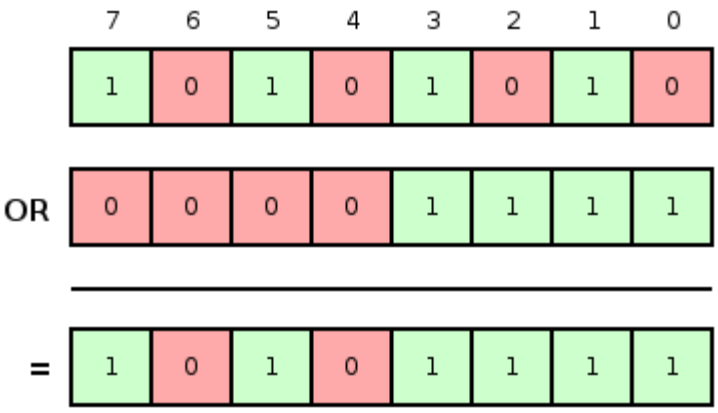



Figure 3.4 - Bitwise OR operation

A single pipe character (|) is the bitwise OR operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a | b;  /* 10101111 */
```

A **bitwise XOR** operation ("exclusive or") results in bits being set if, and only if, the same bit is set in one of the operands but not the other. In other words: bit *n* will be set in the result if bit *n* is exclusively set in only one of the operands.

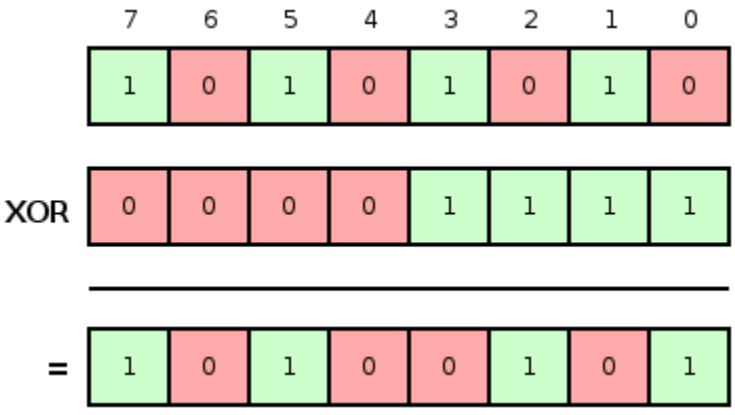



Figure 3.5 - Bitwise XOR operation

The caret character (^) is the bitwise XOR operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a ^ b;  /* 10100101 */
```

A **NOT** operation, also known as a **one's complement**, is a unary operation. That means the operation is performed on a single value instead of two. The NOT operation will simply negate each bit. Every `1` becomes `0` and every `0` becomes `1`.

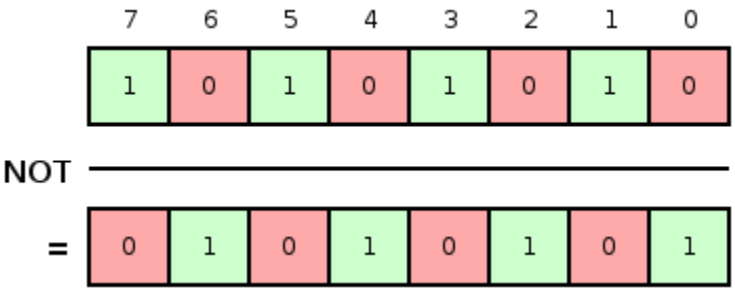



Figure 3.6 - Bitwise NOT operation

A tilde character (~) is the NOT operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = ~a;   /* 01010101 */
```

A **shift** operation shifts all of the bits to the left or the right. In a left shift, bits get "shifted out" on the left and `0` bits get "shifted in" on the right. The opposite goes for a right shift.

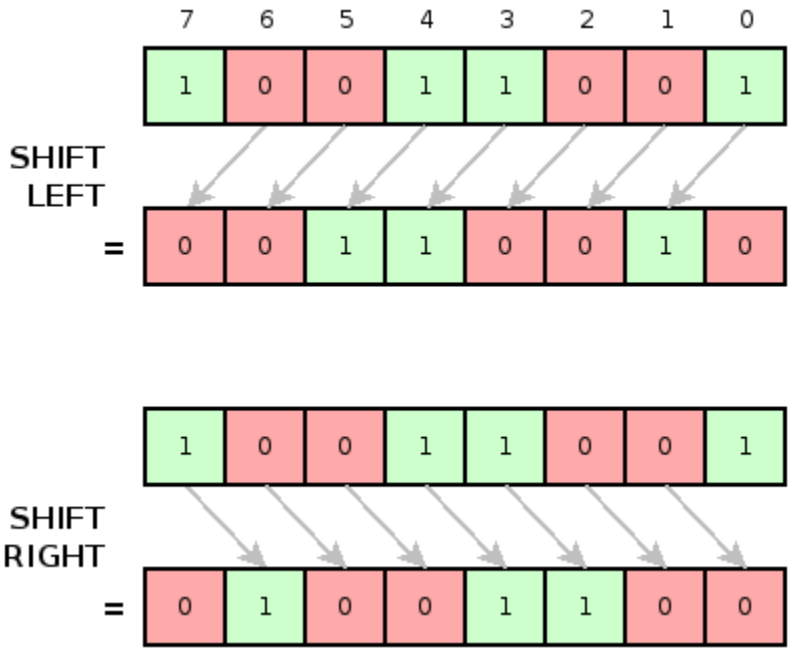



Figure 3.7 - Bitwise shift operation

Two less-than symbols (<<) is the left shift operator and two greater-than symbols (>>) is the right shift operator in C. The right side of the operator is the number of bits to shift.

```
uint8_t a = 0x99; /* 10011001 */
uint8_t b = a<<1; /* 00110010 */
uint8_t c = a>>3; /* 00010011 */
```

## 3.4 Clearing and Setting Bits

Setting and clearing a single bit, without changing any other bits, is a common task in AVR microcontroller programming. You will use these techniques over and over again.

When manipulating a single bit, it is often necessary to have a byte value in which only the bit of interest is set. This byte can then be used with bitwise operations to manipulate that one bit. Let's call this a **bit value mask**. For example, the bit value mask for bit 2 would be `00000100` and the bit value mask for bit 6 would be `01000000`.

Since the number `1` is represented in binary with only bit 0 set, you can get the bit value mask for a given bit by left shifting `1` by the bit number of interest. For example, to get the bit value mask for bit 2, left shift `1` by 2.

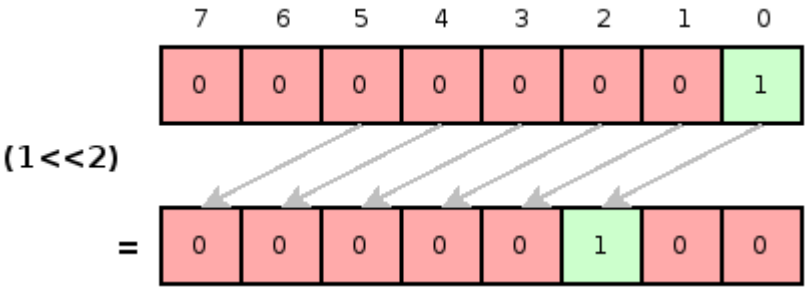



Figure 3.7 - Bit value mask

To **set a bit** in C, OR the value with the bit value mask.

```
uint8_t a = 0x08; /* 00001000 */
                  /* set bit 2 */
a |= (1<<2);      /* 00001100 */
```

Use multiple OR operators to set multiple bits.

```
uint8_t a = 0x08;   /* 00001000 */
                    /* set bits 1 and 2 */
a |= (1<<2)|(1<<1); /* 00001110 */
```

To **clear a bit** in C, NOT the bit value mask so that the bit of interest is the only bit cleared, and then AND that with the value.

```
uint8_t a = 0x0F; /* 00001111 */
                  /* clear bit 2 */
a &= ~(1<<2);     /* 00001011 */
```

Use multiple OR operators to clear multiple bits.

```
uint8_t a = 0x0F;      /* 00001111 */
                       /* clear bit 1 and 2 */
a &= ~((1<<2)|(1<<1)); /* 00001001 */
```

To **toggle a bit** in C, XOR the value with the bit value mask.

```
uint8_t a = 0x0F; /* 00001111 */
                  /* toggle bit 2 */
a ^= (1<<2);      /* 00001011 */
a ^= (1<<2);      /* 00001111 */
```

## 3.5 The Bit Value Macro _BV()

AVR Libc defines a the `_BV()` macro which stands for "bit value". It is a convenience macro to get the bit value mask for a given bit number. The idea is to make the code a little more readable over using a bitwise left shift. Using `_BV(n)` is functionally equivelent to using `(1<<n)`.

```
/* set bit 0 using _BV() */
a |= _BV(0);

/* set bit 0 using shift */
a |= (1<<0);
```

Which method you choose is entirely up to you. You will see both in wide use and should be comfortable seeing either. Since the `_BV(n)` macro is unique to `gcc`, it is not as portable to other compilers as the `(1<<n)` method. However, that is not usually a concern for a hobbyist and the `_BV(n)` is often more comfortable for a beginner.

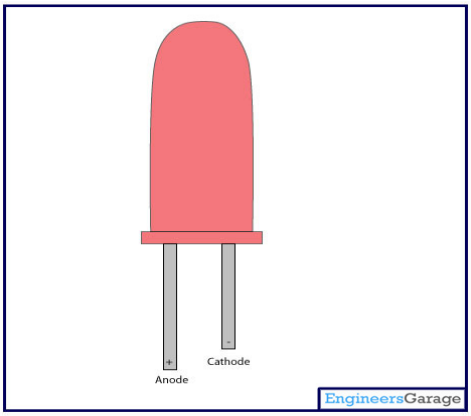
# LED

**Light emitting diodes** (**LEDs**) are semiconductor light sources. The light emitted from **LED**s varies from visible to infrared and ultraviolet regions. They operate on low voltage and power. LEDs are one of the most common electronic components and are mostly used as indicators in circuits. They are also used for luminance and optoelectronic applications.

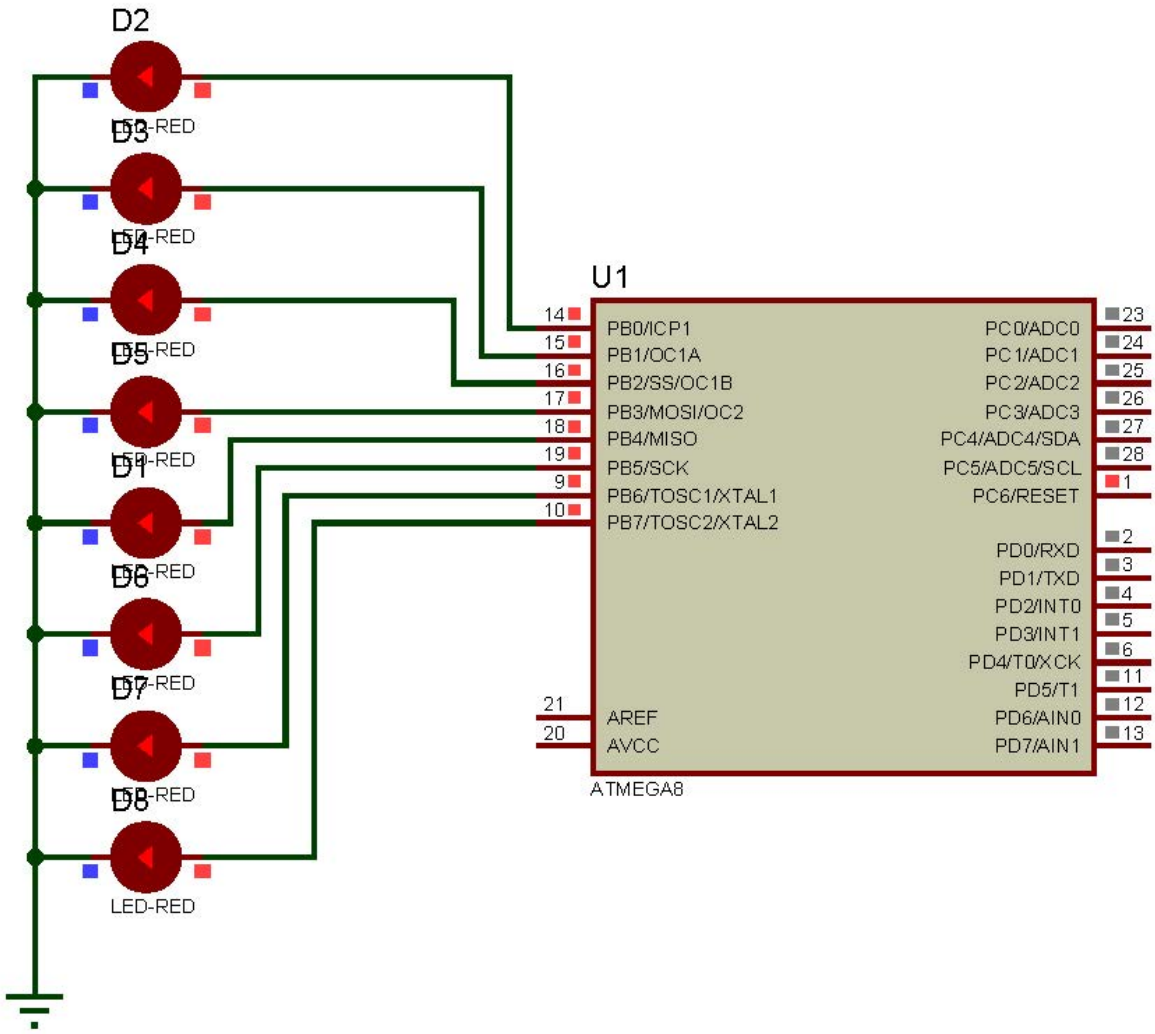
Based on semiconductor diode, **LED**s emit photons when electrons recombine with holes on forward biasing. The two terminals of LEDs are anode (+) and cathode (-) and can be identified by their size. The longer leg is the positive terminal or anode and shorter one is negative terminal.

The forward voltage of **LED** (1.7V-2.2V) is lower than the voltage supplied (5V) to drive it in a circuit. Using an LED as such would burn it because a high current would destroy its p-n gate. Therefore a current limiting resistor is used in series with LED. Without this resistor, either low input voltage (equal to forward voltage) or PWM (pulse width modulation) is used to drive the LED. Get details about internal structure of a LED.

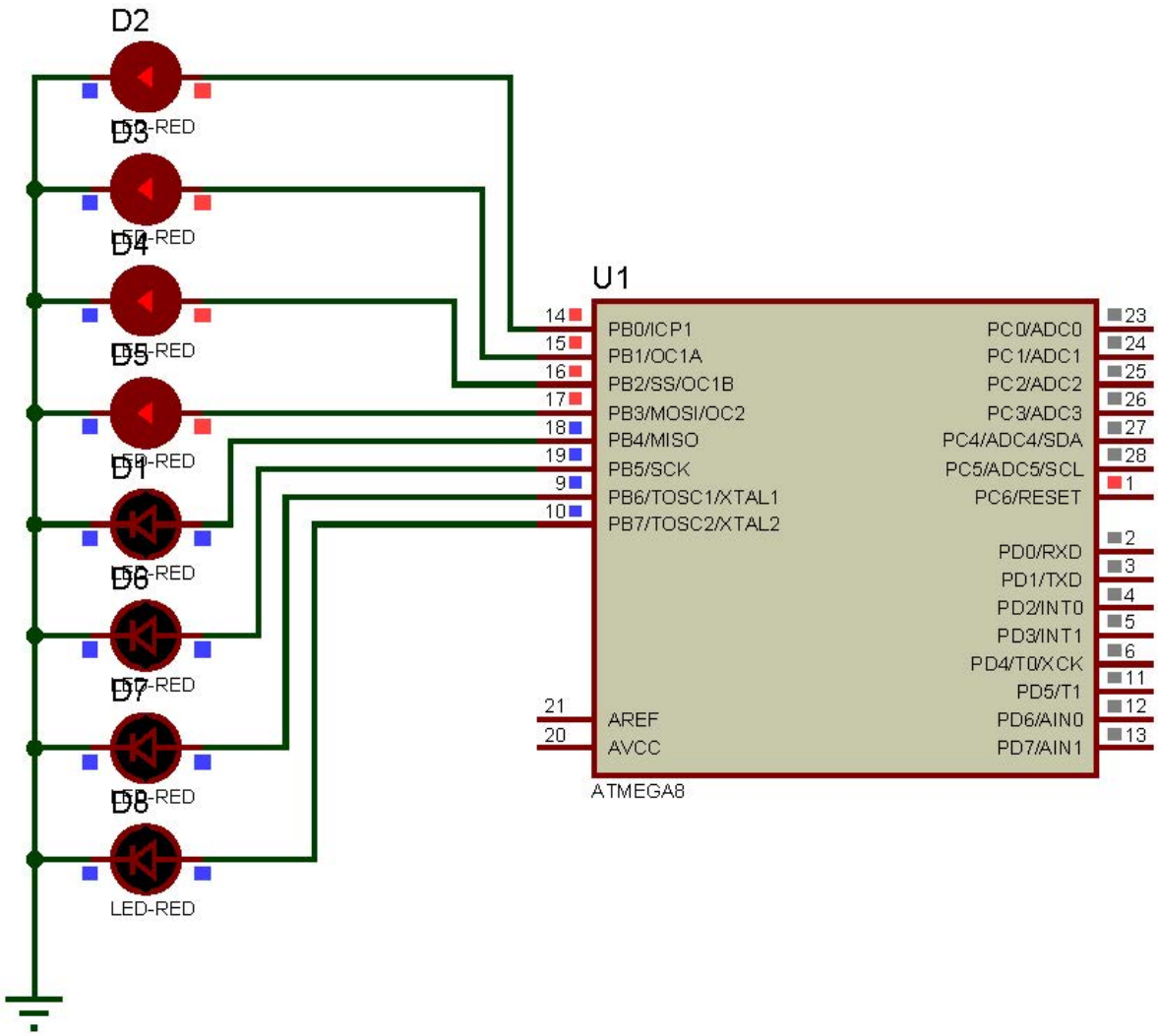
## Pin Diagram:



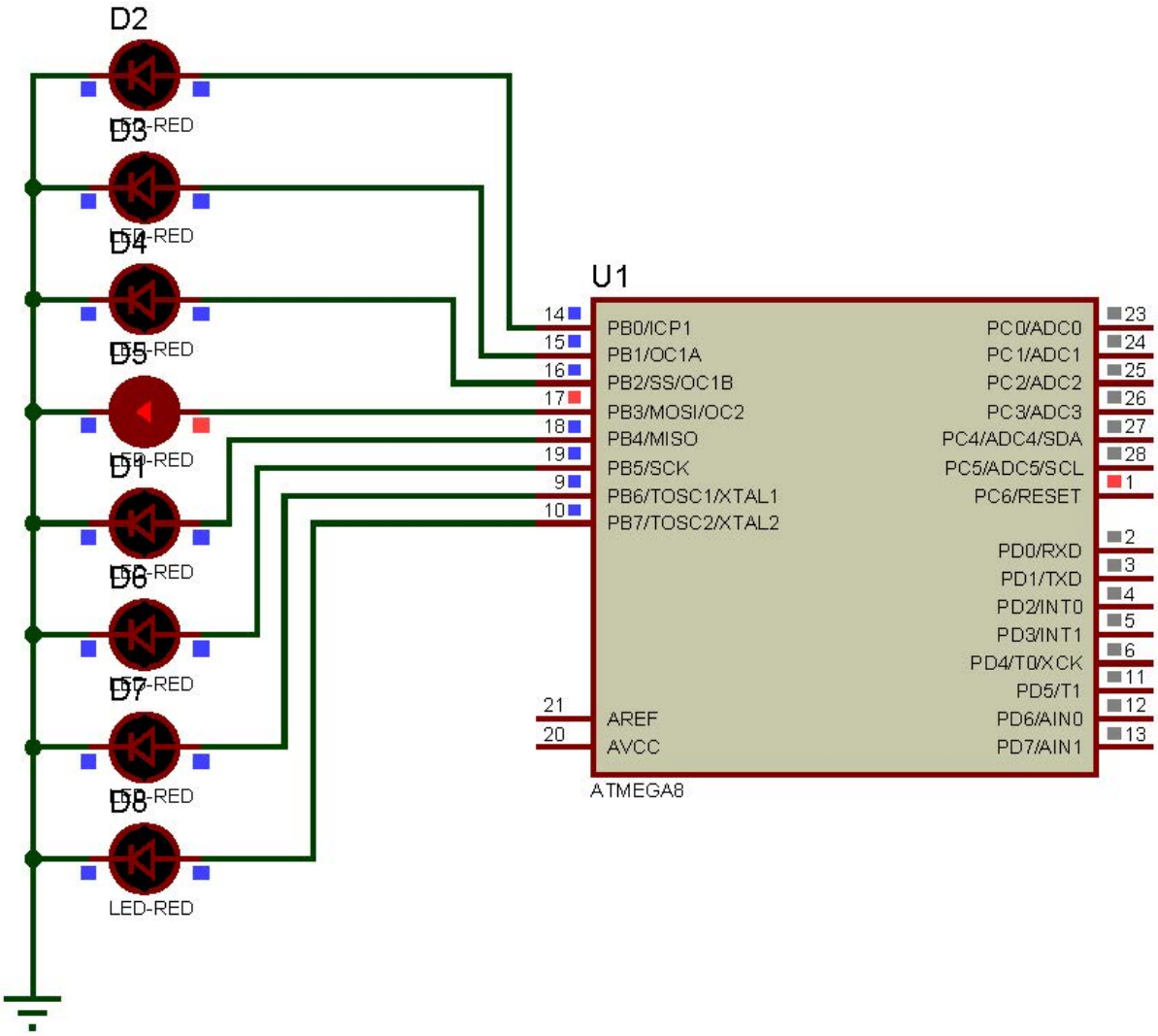
```c
//Program to Blink 8 LEDs connected on PORTB

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

void main()
{
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        PORTB = 0xFF; //LED ON
        _delay_ms(300);
        PORTB = 0; //LED OFF
        _delay_ms(300);
    }
}
```



```c
//Program to Nibble (4 at a time) LEDs Blinking connected on PORTB

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

void main()
{
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        PORTB = 0x0F; //LSB 4 LED ON
        _delay_ms(300);
        PORTB = 0xF0; //MSB 4 LED ON
        _delay_ms(300);
    }
}
```



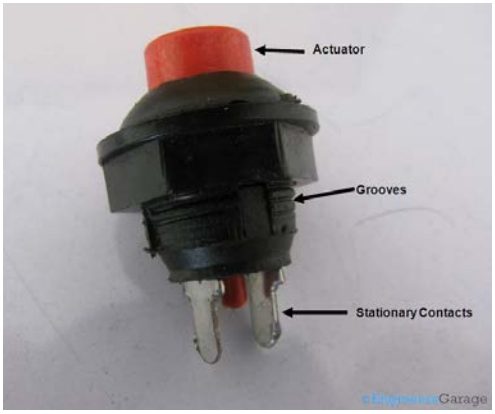
```c
//Program to Move LED from LSB to MSB connected on PORTB

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

void main()
{
    int i;
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        for(i=1;i<=128;i*=2)
        {
            PORTB = i; //LED start from LSB moving towards MSB
            _delay_ms(300);
        }
    }
}
```



# Insight - How Push Button Switch Works

Ever imagined how quickly the horn of your vehicle responds? Not even a second or two, just push the button and there comes the sound. You put your hand off the button and it's back to silent mode. An instant response of these horns is due to what we call "push -button switches".

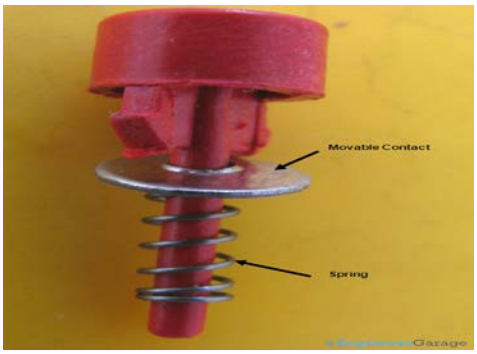
Push button switches are those which can be made to work with the force of a finger or two. Not only vehicles but camera, lifts and several other common and uncommon interactions with machines/gadgets involve push button switches applications. But what makes this switch so user friendly? What makes it respond only for the time it is pressed no longer or shorter? How the contacts inside this switch work? Let us explore the internal architecture of the pushbutton and find out the answers to the questions through this insight.




The image above shows the external view of a conventional SPST push button switch. Almost all the parts of the switch can be figured out by observing its external structure. The red colored bulge is the actuator of the switch. The actuator extends towards the bottom of the switch and emerges out as a thin cylinder. Among other prominent extensions are the two stationary metal contacts legs at the bottom. A groovy pattern is also provided for the purpose of easy mounting.

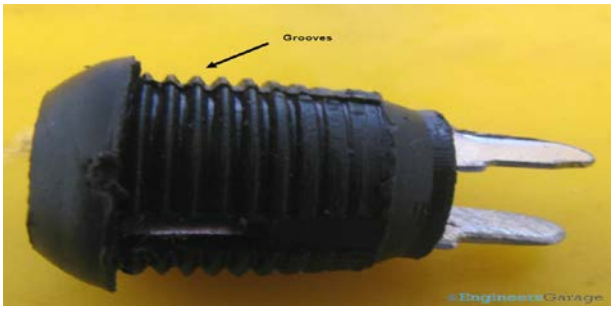
Usually, external body of push button switches is made from polymer plastics and can have multiple shapes, sizes and output terminals depending upon its use.

## Actuator & Lower part




The image above shows the movable contact and the spring which is the heart of the push mechanism.

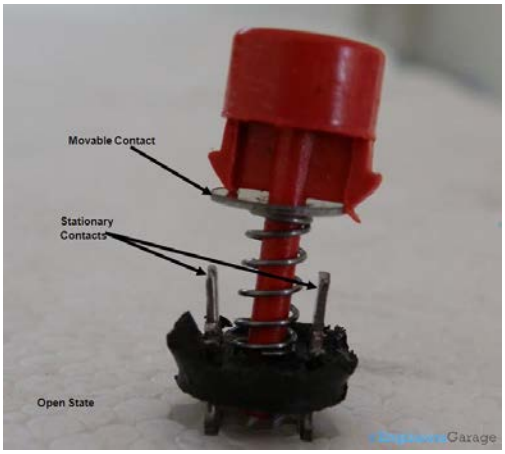
Whenever the actuator is pressed down, the spring compresses; the movable contact connects with the stationary ones. Some switches can have a latch in the internal structure which locks the state when the switch is engaged. When pressed again, the switch gets disengaged and returns to its normal position. Such switches are called push to ON switch. In the switch taken in this article, there is no latch to hold down the switch, hence, as soon as the force is removed from the actuator, spring instantaneously gets decompressed and switch comes back to its rest stage



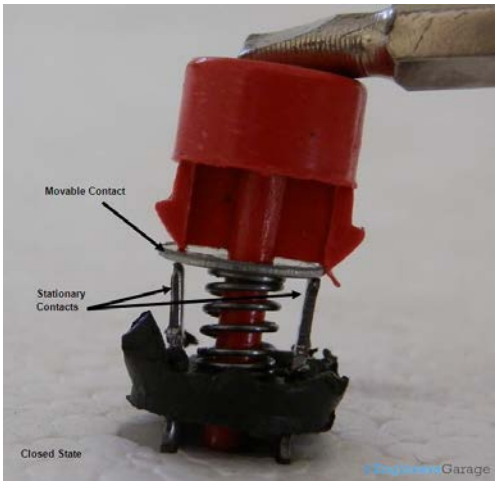

The lower section of the switch has grooves which ease out the process of mounting the switch over a surface.

## Open & Close State

The open and closed contact states of a push button switch can be understood with the help of the image below:

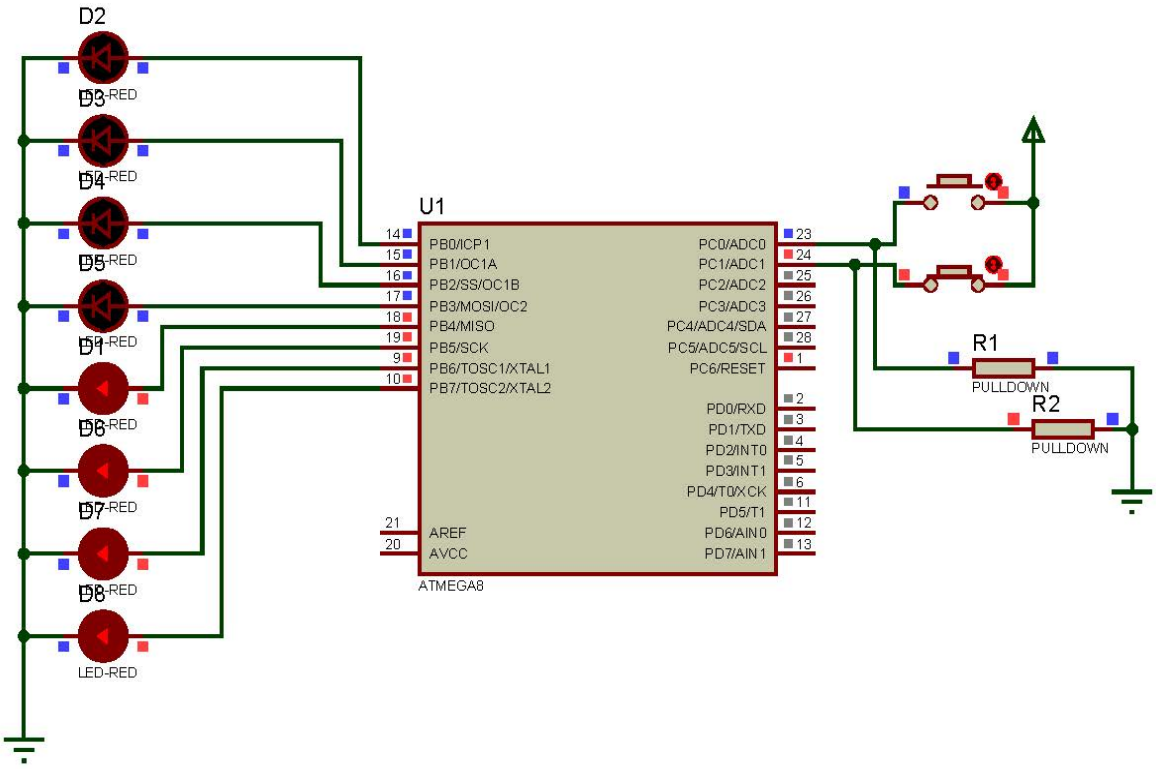



The image above shows the open state of the push button switch where the spring is at its rest state and the movable contact is not bridging the gap between stationary contacts.




This is the closed state of the switch where the spring is compressed enough so that the movable contact can connect to the stationary contacts and push button switch starts to conduct.

```c
//Program to Control LED using 2 Switch

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

void main()
{
    DDRC = 0; //Configuring PortC as Input
    DDRB = 0xFF; //Configuring PortB as Output
    while(1) //Forever Loop
    {
        if((PINC & 0x03) == 0) //SW1 & SW2 Not Pressed
        {
            PORTB = 0; //All 8 LEDs OFF
        }
        if((PINC & 0x03) == 0x01) //SW1 Pressed & SW2 Not Pressed
        {
            PORTB = 0x0F; //LSB 4 LEDs ON, MSB 4 LEDs OFF
        }
        if((PINC & 0x03) == 0x02) //SW1 Not Pressed & SW2 Pressed
        {
            PORTB = 0xF0; //LSB 4 LEDs OFF, MSB 4 LEDs ON
        }
        if((PINC & 0x03) == 0x03) //SW1 & SW2 Pressed
        {
            PORTB = 0xFF; //All 8 LEDs ON
        }
    }
}
```

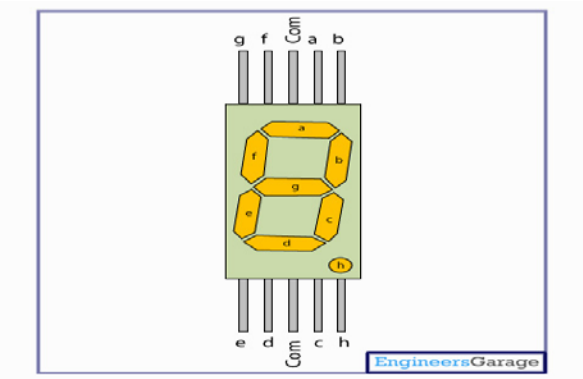


# SEVEN SEGMENT DISPLAY

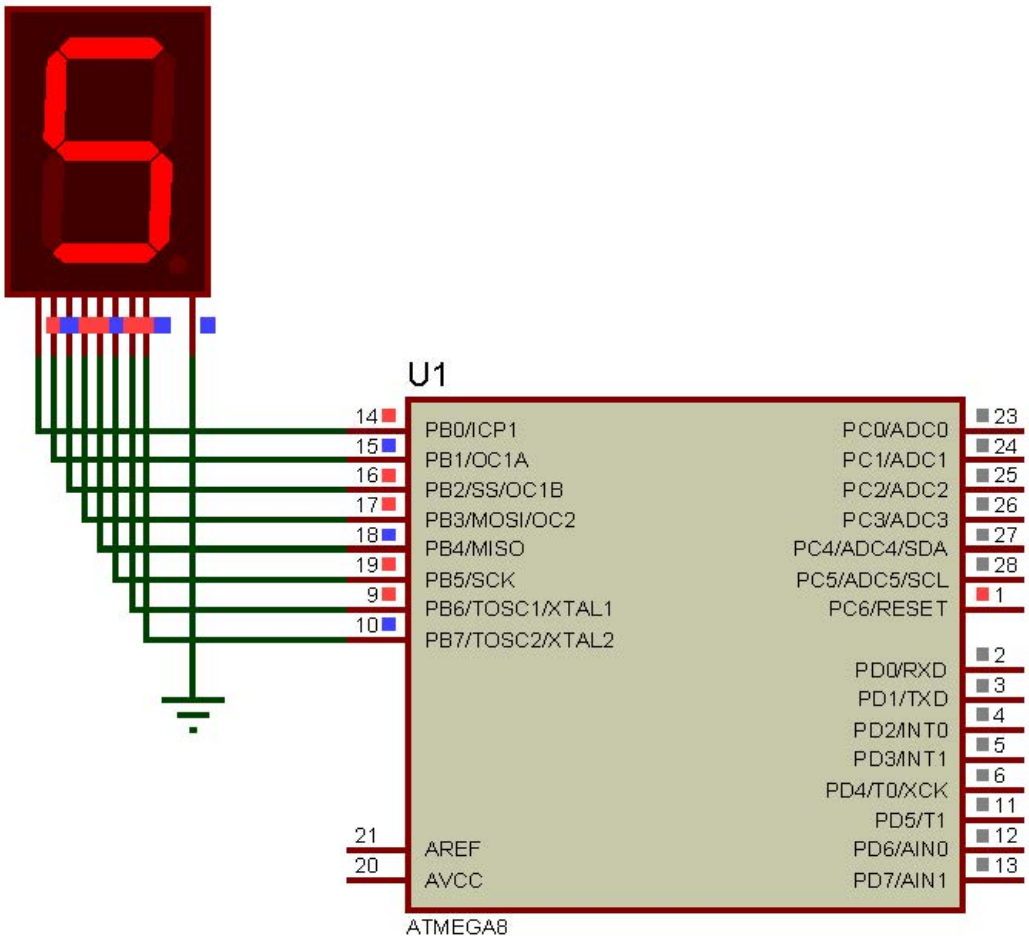
A **seven segment display** is the most basic electronic display device that can display digits from 0-9. They find wide application in devices that display numeric information like digital clocks, radio, microwave ovens, electronic meters etc. The most common configuration has an array of eight LEDs arranged in a special pattern to display these digits. They are laid out as a squared-off figure ‘8’. Every LED is assigned a name from 'a' to 'h' and is identified by its name. Seven LEDs 'a' to 'g' are used to display the numerals while eighth LED 'h' is used to display the dot/decimal.

A seven segment is generally available in ten pin package. While eight pins correspond to the eight LEDs, the remaining two pins (at middle) are common and internally shorted. These segments come in two configurations, namely, Common cathode (CC) and Common anode (CA). In CC configuration, the negative terminals of all LEDs are connected to the common pins. The common is connected to ground and a particular LED glows when its corresponding pin is given high. In CA arrangement, the common pin is given a high logic and the LED pins are given low to display a number. Find out more information about a seven segment display and its working.

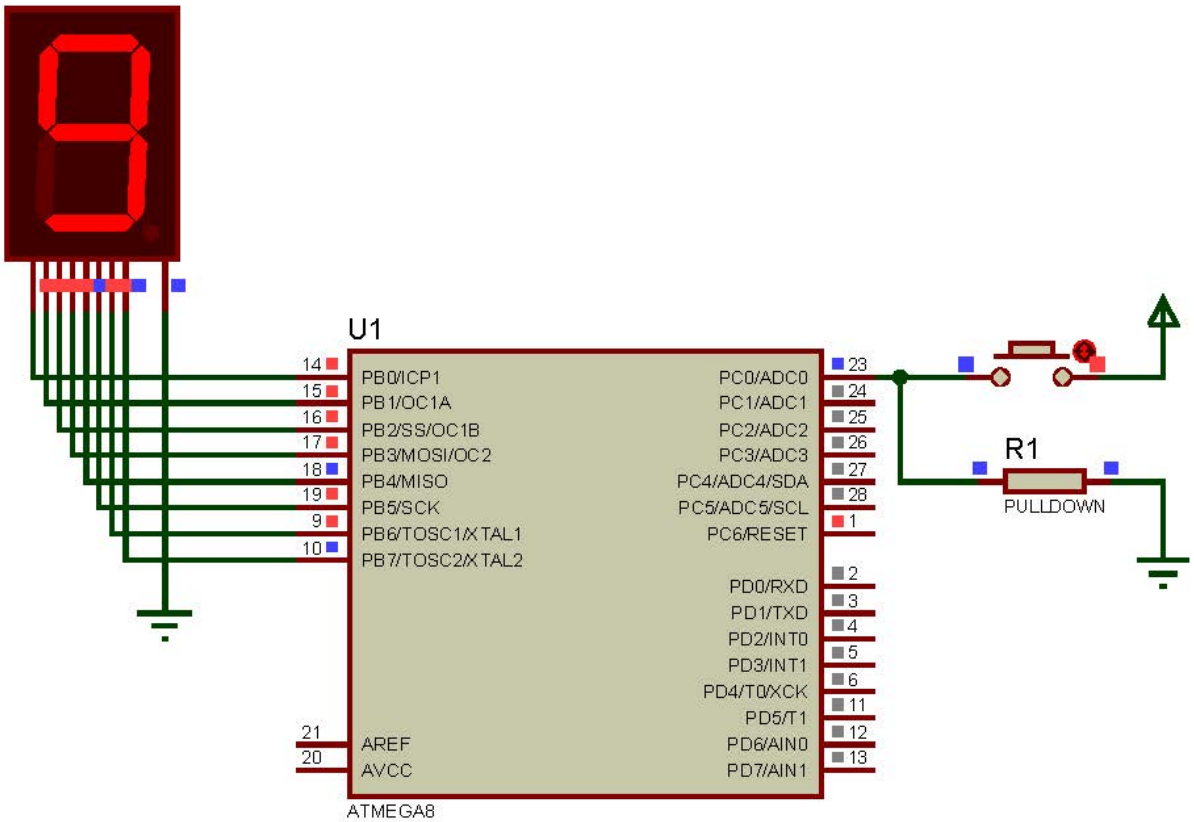
## Pin Diagram:



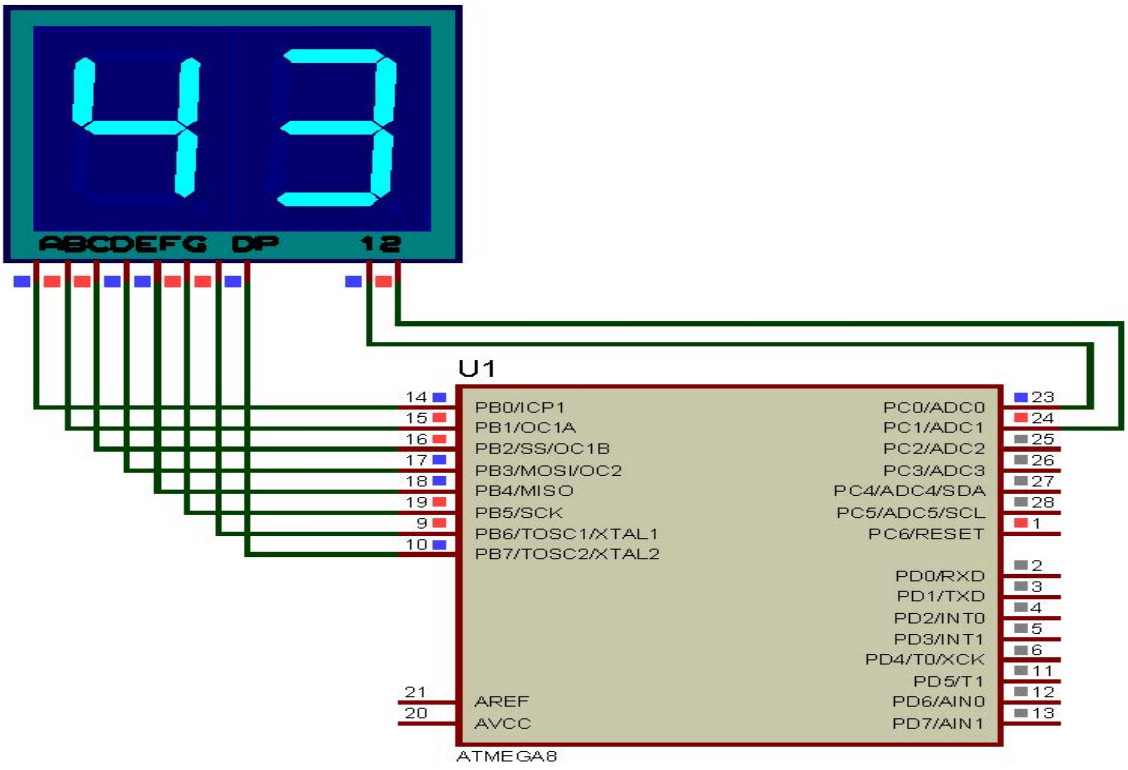
```c
//Program to Display 0 to 9 on Single Common Cathode Type 7 Segment

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

void main()
{
    int i, a[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
    DDRB = 0xFF; //Configuring PortB as Output
    while(1) //Forever Loop
    {
        for(i=0;i<=9;i++) //Loop to display numbers
        {
            PORTB = a[i]; //Loading Values of Numbers
            _delay_ms(300);
        }
    }
}
```



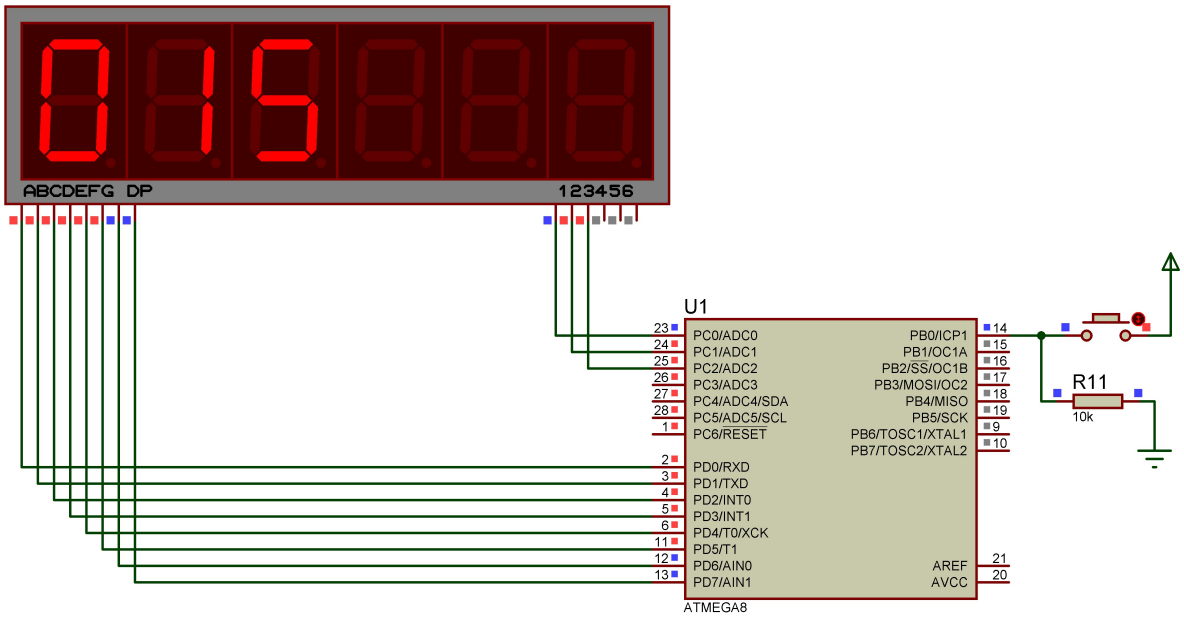
```c
//Program to develop Token Management System (Single 7 Segment using Single Switch)

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

void main()
{
    int count = 0, a[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
    DDRB = 0xFF; //Configuring PortB as Output
    DDRC = 0; //Configuring PC0 as Input
    while(1) //Forever Loop
    {
        if((PINC & 0x01)==0x01) //Switch Pressed
        {
            count++; //Increase the count if switch pressed
            if(count>9)
                count = 0; //reset count if count value increases to 10
            while((PINC & 0x01)==0x01); //Wait for the switch to be released
        }
        PORTB = a[count]; //Display the value on 7 Segment
        _delay_ms(100);
    }
}
```



```c
//Program to Display 0 to 99 on Double Common Cathode Type 7 Segment

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

void main()
{
    int i,j, a[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
    DDRB = 0xFF; //Configuring PortB as Output
    DDRC = 0x03; //Configuring PC0 PC1 as Output
    while(1) //Forever Loop
    {
        for(i=0;i<=9;i++) //Loop to display MSB digit
        {
            for(j=0;j<=9;j++)
            {
                PORTC = 2; //7Seg 1 ON, 7Seg 2 OFF
                PORTB = a[i]; //Loading Values of i
                _delay_ms(50);
                PORTC = 1; //7Seg 1 OFF, 7Seg 2 ON
                PORTB = a[j]; //Loading Values of j;
                _delay_ms(70);
            }
        }
    }
}
```



```
//Token Management System using Single switch and 3 digit 7 Segment Multiplexed

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

void display(int num) //Function to break the digits and display on 7 Segment
{
    int i=0, seg[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
    for(i=2;i>=0;i--) //For 3 Digit
    {
        PORTC = ~(1<<i); //Selecting 7 Seg
        PORTD = seg[num%10]; //Breaking digit
        _delay_ms(5);
        num/=10; //Updating number to next level
    }
}

void main()
{
    int count = 0;
    DDRC = DDRD = 0xFF; //Output Configured
    DDRB &= ~(1<<DDB0); //PB0 configured as Input
    while(1) //Forever Loop
    {
        if((PINB & 0x01)==1) //if switch pressed
        {
            count++; //Increment count by 1
            while((PINB & 0x01)); //Wait for switch to be released
        }
        else
        {
            display(count); // Calling display function
        }
    }
}
```



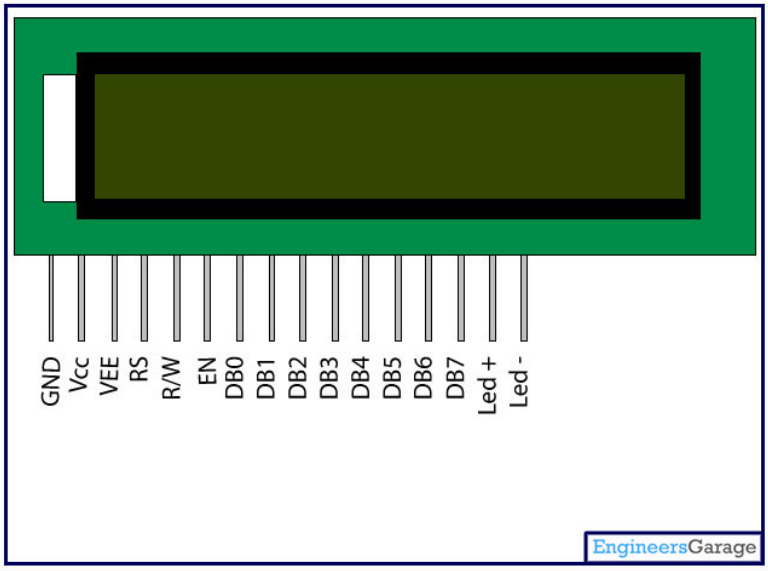
# LCD

LCD (Liquid Crystal Display) screen is an electronic display module and find a wide range of applications. A 16x2 LCD display is very basic module and is very commonly used in various devices and circuits. These modules are preferred over seven segments and other multi segment LEDs. The reasons being: LCDs are economical; easily programmable; have no limitation of displaying special & even custom characters (unlike in seven segments), animations and so on.

A **16x2 LCD** means it can display 16 characters per line and there are 2 such lines. In this LCD each character is displayed in 5x7 pixel matrix. This LCD has two registers, namely, Command and Data.

The command register stores the command instructions given to the LCD. A command is an instruction given to LCD to do a predefined task like initializing it, clearing its screen, setting the cursor position, controlling display etc. The data register stores the data to be displayed on the LCD. The data is the ASCII value of the character to be displayed on the LCD. Click to learn more about internal structure of a LCD.

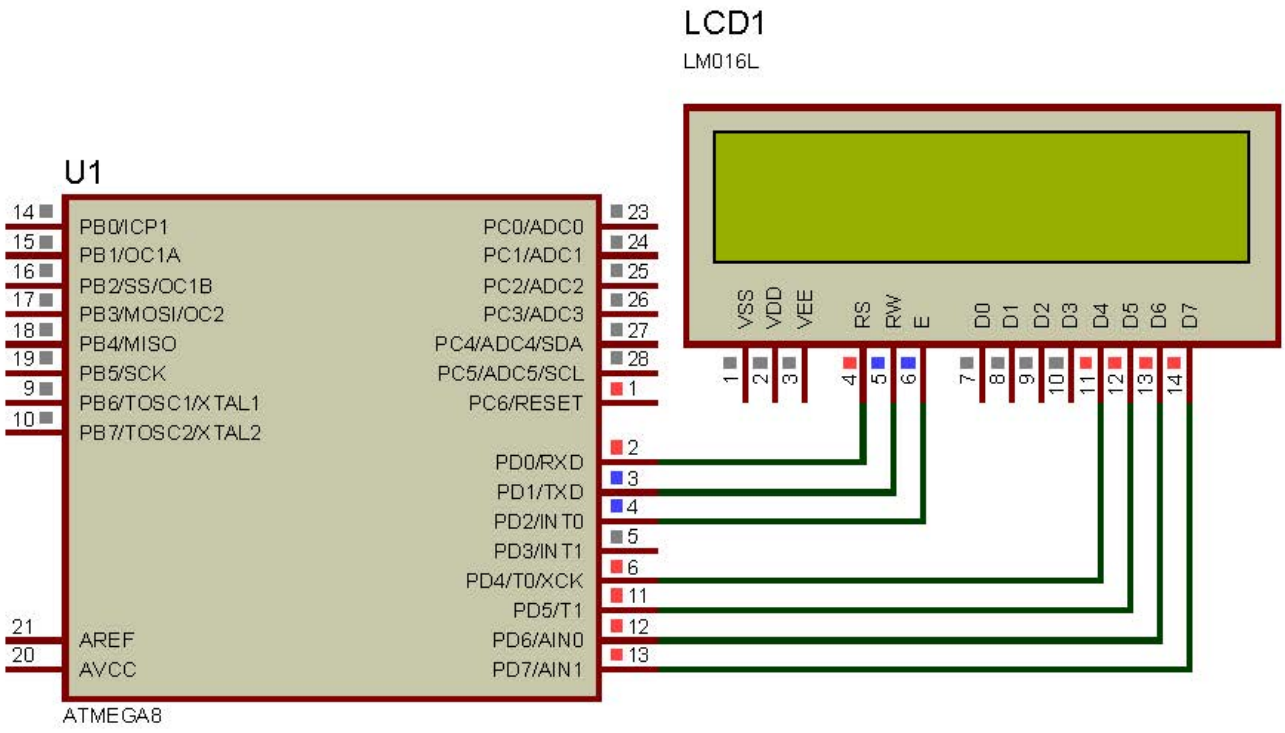
**Pin Diagram:**



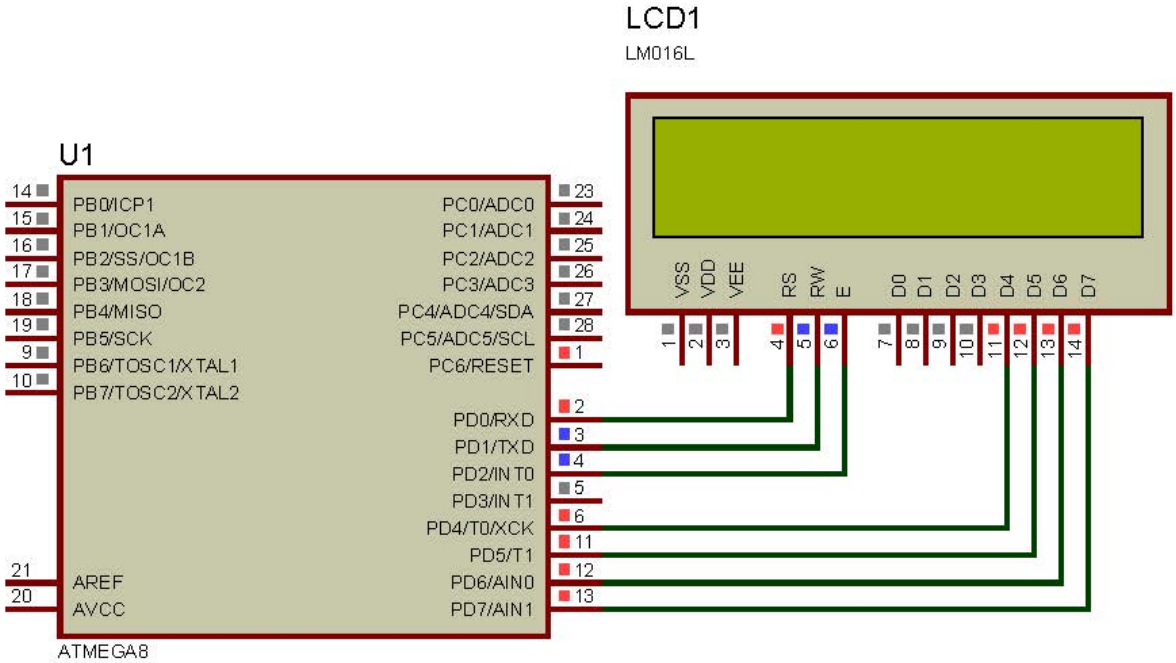
**Pin Description:**

| Pin No | Function | Name |
|---|---|---|
| 1 | Ground (0V) | Ground |
| 2 | Supply voltage; 5V (4.7V – 5.3V) | Vcc |
| 3 | Contrast adjustment; through a variable resistor | $V_{EE}$ |
| 4 | Selects command register when low; and data register when high | Register Select |
| 5 | Low to write to the register; High to read from the register | Read/write |
| 6 | Sends data to data pins when a high to low pulse is given | Enable |
| 7 | 8-bit data pins | DB0 |
| 8 | | DB1 |
| 9 | | DB2 |
| 10 | | DB3 |
| 11 | | DB4 |
| 12 | | DB5 |
| 13 | | DB6 |
| 14 | | DB7 |
| 15 | Backlight $V_{CC}$ (5V) | Led+ |
| 16 | Backlight Ground (0V) | Led- |

```c
//Program to display Name on Alphanumeric LCD (2 Rows, 16 Columns)

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

#include "lcd.h" //LCD Header File
#include "lcd.c"

void main()
{
    lcd_init(LCD_DISP_ON); //Initializing LCD
    lcd_puts("SOFCON INDIA"); //Function to display message on LCD
    while(1); //forever loop
}
```



```c
//Program to display Name on Alphanumeric LCD (2 Rows, 16 Columns)

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

#include "lcd.h" //LCD Header File
#include "lcd.c"

void main()
{
    int i;
    lcd_init(LCD_DISP_ON); //Initializing LCD
    while(1) //forever loop
    {
        for(i=0;i<10;i++)
        {
            lcd_clrscr(); //Clearing Message
            lcd_gotoxy(i,0); //Fixing Cursor Position
            lcd_puts("SOFCON"); //Message to be displayed on First Row
            lcd_gotoxy(10-i,1); //Fixing Cursor Position
            lcd_puts("INDIA"); //Message to be display on Second Row
            _delay_ms(200); //Delay
        }
    }
}
```

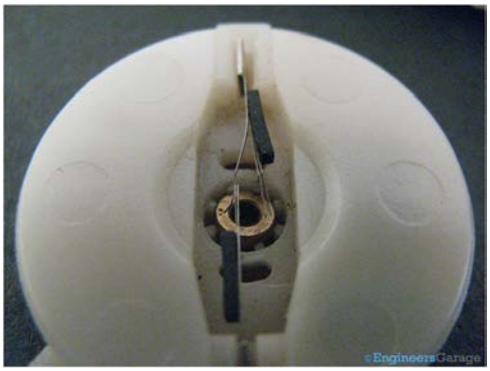


# Insight - How DC Motor Works

DC Motors convert electrical energy (voltage or power source) to mechanical energy (produce rotational motion). They run on direct current. The Dc motor works on the principle of Lorentz force which states that when a wire carrying current is placed in a region having magnetic field, than the wire experiences a force. This Lorentz force provides a torque to the coil to rotate.

A commonly used DC Motor is shown in the image above.

On opening the back cover of the motor you can have a look to the motor speed control circuit from behind.



The above image shows the brushes of the DC motor which helps the motor to take input current to the coil. The brushes always remain connected with any two commutators and supplying the input current to the coil while it is rotating.

You can have a closer look as to how the electrical coil is arranged in the permanent magnet.

When DC current is passed through the coil, it works as electromagnet. As you can see the iron plates attached around the coil helps the coil to keep in center and movable. Permanent magnet attracts these three iron plates equally; resultantly it stays in the center of the permanent magnet.

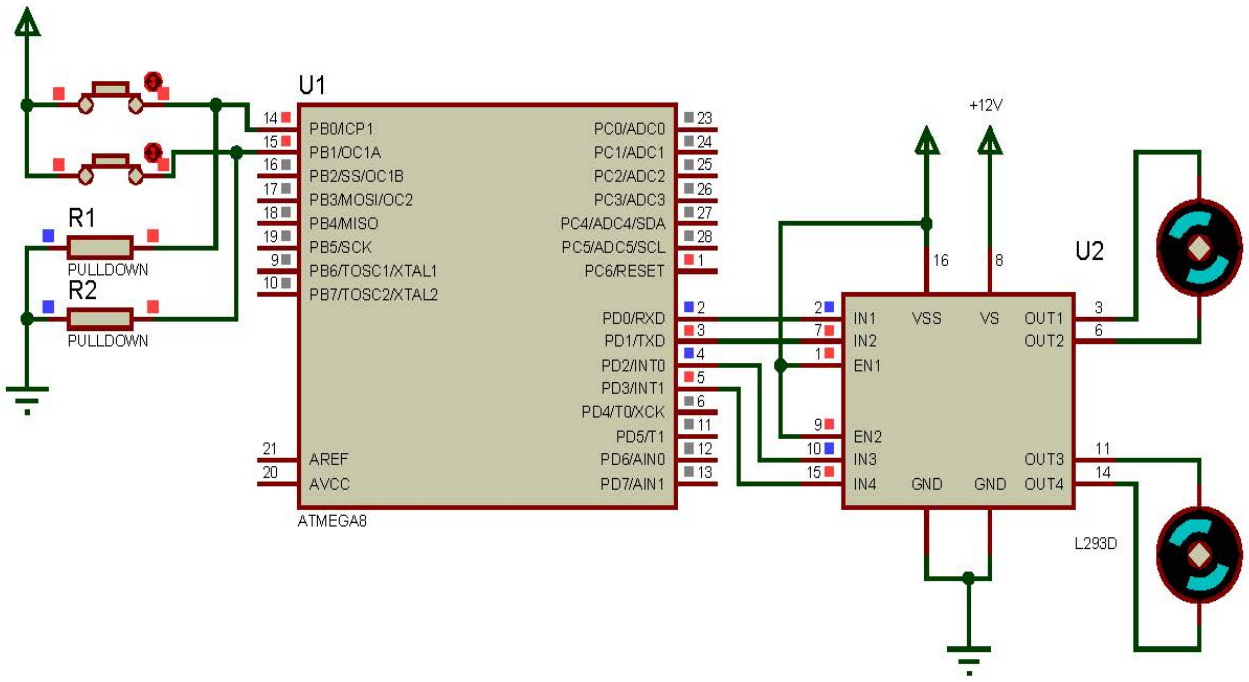
There are three commutators shown in the image. Each one is directly connected with the coil to supply the current in.

The permanent magnet is cascaded in the body of the motor. The coil working as electromagnet moves in the magnetic field of this magnet.

A motor speed control IC is used to control the rotating speed of compact DC motor. The IC integrated in the circuit has an inbuilt reverse voltage protection circuit.

**Working:** As we have discussed, DC motor work on Lorentz force concept. When we pass the input DC current to the coil through the brushes, it directly goes to the coil inside the motor body. This makes coil to work as an electromagnet. Magnetic fields of both magnets interact with each other that results in a force which in turn produces the necessary torque required to move the coil. This torque drives the coil to move round and a shaft attached with the coil moves too.

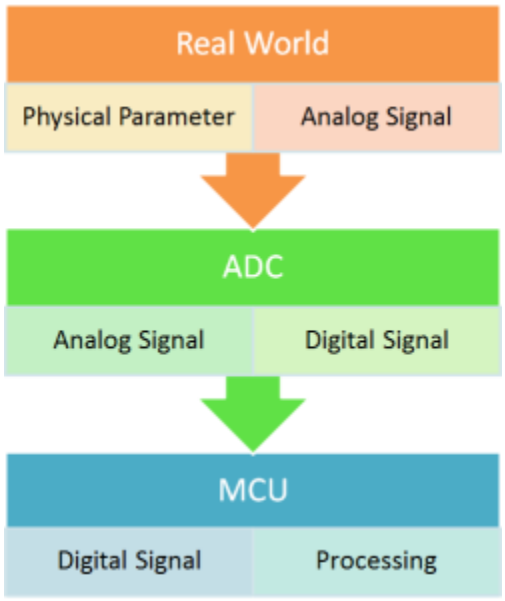
```c
//Program to develop LINE FOLLOWER ROBOT (2 Inputs (IR Sensor) and 2 Outputs (DC Motor)

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

void main()
{
    DDRD = 0x0F; //Configuring PD0 PD1 PD2 PD3 as Output
    DDRB = 0; //Configuring PortB as Input
    while(1) //Forever Loop
    {
        if((PINB & 0x03) == 0) //IR1 and IR2 on Black Line (Off Condition)
        {
            PORTD = 0; //Left and Right Motor OFF, STOP
        }
        if((PINB & 0x03) == 0x01) //IR1 on White Surface (On Condition) and IR2 on Black Line
        (Off Condition)
        {
            PORTD = 0x02; //Left Motor ON and Right Motor OFF, RIGHT
        }
        if((PINB & 0x03) == 0x02) //IR1 on Black Line(OFF Condition) and IR2 on White
        Surface(ON Condition)
        {
            PORTD = 0x08; //Left Motor OFF and Right Motor ON, LEFT
        }
        if((PINB & 0x03) == 0x03) //IR1 and IR2 on White Surface(ON Condition)
        {
            PORTD = 0x0A; //Left and Right Motor ON, FORWARD
        }
    }
}
```



# The ADC of the AVR

## Analog to Digital Conversion

Most real world data is analog. Whether it be temperature, pressure, voltage, etc, their variation is always analog in nature. For example, the temperature inside a boiler is around 800°C. During its light-up, the temperature never approaches directly to 800°C. If the ambient temperature is 400°C, it will start increasing gradually to 450°C, 500°C and thus reaches 800°C over a period of time. This is an analog data.



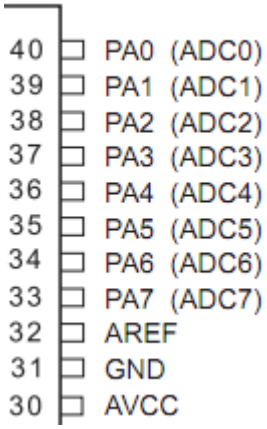

Signal Acquisition Process

Now, we must process the data that we have received. But analog signal processing is quite inefficient in terms of accuracy, speed and desired output. Hence, we convert them to digital form using an Analog to Digital Converter (ADC).

### Signal Acquisition Process

In general, the signal (or data) acquisition process has 3 steps.

- In the **Real World**, a sensor senses any physical parameter and converts into an equivalent analog electrical signal.
- For efficient and ease of signal processing, this analog signal is converted into a digital signal using an **Analog to Digital Converter (ADC)**.
- This digital signal is then fed to the **Microcontroller (MCU)** and is processed accordingly.




ADC Pins – ATMEGA16/32

### Interfacing Sensors

In general, sensors provide with analog output, but a MCU is a digital one. Hence we need to use ADC. For simple circuits, comparator op-amps can be used. But even this won't be required if we use a MCU. We can straightaway use the inbuilt ADC of the MCU. In ATMEGA16/32, PORTA contains the ADC pins.

# The ADC of the AVR

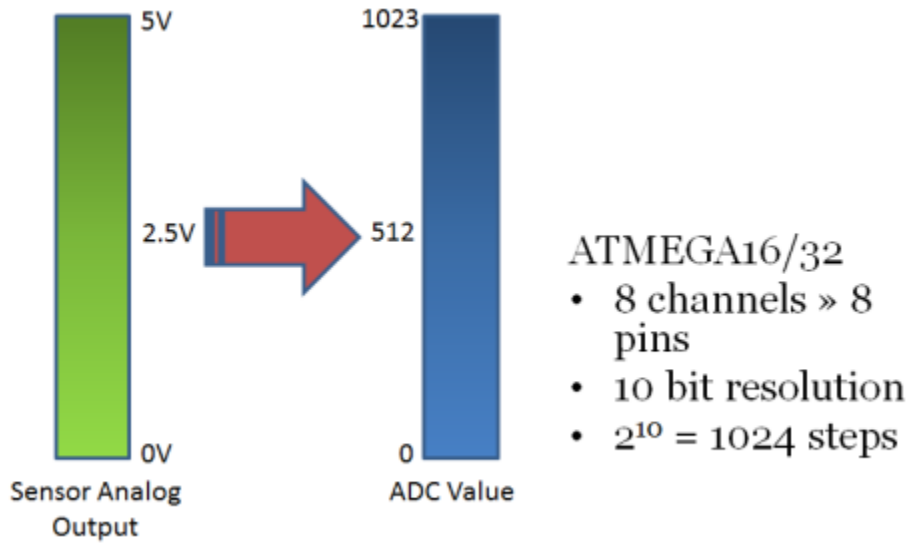
The AVR features inbuilt ADC in almost all its MCU. In ATMEGA16/32, PORTA contains the ADC pins. Some other features of the ADC are as follows:

- 10-bit Resolution
- 0.5 LSB Integral Non-linearity
- ±2 LSB Absolute Accuracy
- 13 - 260 µs Conversion Time
- Up to 15 kSPS at Maximum Resolution
- 8 Multiplexed Single Ended Input Channels
- 7 Differential Input Channels
- 2 Differential Input Channels with Optional Gain of 10x and 200x
- Optional Left adjustment for ADC Result Readout
- 0 - $V_{CC}$ ADC Input Voltage Range
- Selectable 2.56V ADC Reference Voltage
- Free Running or Single Conversion Mode
- ADC Start Conversion by Auto Triggering on Interrupt Sources
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

ADC Features – ATMEGA16/32

Right now, we are concerned about the **8 channel 10 bit resolution** feature.

- **8 channel** implies that there are 8 ADC pins are multiplexed together. You can easily see that these pins are located across PORTA (PA0…PA7).
- **10 bit resolution** implies that there are 2^10 = 1024 steps (as described below).




8 channel 10 bit ADC

Suppose we use a 5V reference. In this case, any analog value in between 0 and 5V is converted into its equivalent ADC value as shown above. The 0-5V range is divided into 2^10 = 1024 steps. Thus, a 0V input will give an ADC output of 0, 5V input will give an ADC output of 1023, whereas a 2.5V input will give an ADC output of around 512. This is the basic concept of ADC.

To those whom it might concern, the type of ADC implemented inside the AVR MCU is of Successive Approximation type.

Apart from this, the other things that we need to know about the AVR ADC are:

- ADC Prescaler
- ADC Registers – ADMUX, ADCSRA, ADCH, ADCL and SFIOR

# ADC Prescaler

The ADC of the AVR converts analog signal into digital signal at some regular interval. This interval is determined by the clock frequency. In general, the ADC operates within a frequency range of 50kHz to 200kHz. But the CPU clock frequency is much higher (in the order of MHz). So to achieve it, frequency division must take place. The prescaler acts as this division factor. It produces desired frequency from the external higher frequency. There are some predefined division factors – 2, 4, 8, 16, 32, 64, and 128. For example, a prescaler of 64 implies F_ADC = F_CPU/64. For F_CPU = 16MHz, F_ADC = 16M/64 = 250kHz.

Now, the major question is… which frequency to select? Out of the 50kHz-200kHz range of frequencies, which one do we need? Well, the answer lies in your need. **There is a trade-off between frequency and accuracy**. Greater the frequency, lesser the accuracy and vice-versa. So, if your application is not sophisticated and doesn't require much accuracy, you could go for higher frequencies.

# ADC Registers

We will discuss the registers one by one.

# ADMUX – ADC Multiplexer Selection Register

The ADMUX register is as follows.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

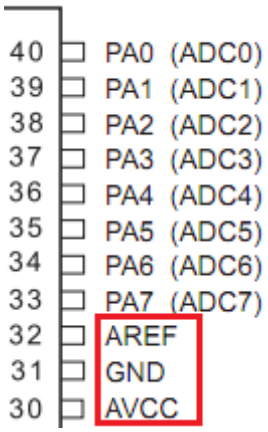
ADMUX Register

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

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

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

Reference Voltage Selection




ADC Voltage Reference Pins

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

Let's choose the second option for Vcc = 5V.

- **Bit 5 – ADLAR – ADC Left Adjust Result** – Make it '1′ to Left Adjust the ADC Result. We will discuss about this a bit later.

- **Bits 4:0 – MUX4:0 – Analog Channel and Gain Selection Bits** – There are 8 ADC channels (PA0…PA7). Which one do we choose? Choose any one! It doesn't matter. How to choose? You can choose it by setting these bits. Since there are 5 bits, it consists of 2^5 = 32 different conditions as follows. However, we are concerned only with the first 8 conditions. Initially, all the bits are set to zero.

| 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 | | | |

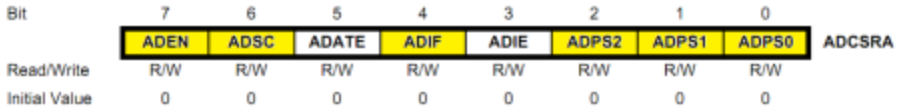
Input Channel and Gain Selections

Thus, to initialize ADMUX, we write

```
ADMUX = (1<<REFS0);
```

# ADCSRA – ADC Control and Status Register A

The ADCSRA register is as follows.




ADCSRA Register

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

- **Bit 7 – ADEN – ADC Enable** – As the name says, it enables the ADC feature. Unless this is enabled, ADC operations cannot take place across PORTA i.e. PORTA will behave as GPIO pins.

- **Bit 6 – ADSC – ADC Start Conversion** – Write this to ’1′ before starting any conversion. This 1 is written as long as the conversion is in progress, after which it returns to zero. Normally it takes 13 ADC clock pulses for this operation. But when you call it for the first time, it takes 25 as it performs the initialization together with it.

- **Bit 5 – ADATE – ADC Auto Trigger Enable** – Setting it to ’1′ enables auto-triggering of ADC. ADC is triggered automatically at every rising edge of clock pulse. View the SFIOR register for more details.

- **Bit 4 – ADIF – ADC Interrupt Flag** – Whenever a conversion is finished and the registers are updated, this bit is set to ’1′ automatically. Thus, this is used to check whether the conversion is complete or not.

- **Bit 3 – ADIE – ADC Interrupt Enable** – When this bit is set to ’1′, the ADC interrupt is enabled. This is used in the case of interrupt-driven ADC.

- **Bits 2:0 – ADPS2:0 – ADC Prescaler Select Bits** – The prescaler (division factor between XTAL frequency and the ADC clock frequency) is determined by selecting the proper combination from the following.

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

ADC Prescaler Selections

Assuming XTAL frequency of 16MHz and the frequency range of 50kHz-200kHz, we choose a prescaler of 128.

Thus, F_ADC = 16M/128 = 125kHz.

Thus, we initialize ADCSRA as follows.

```
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
// prescaler = 128
```

# ADCL and ADCH – ADC Data Registers

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

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

*ADLAR = 0*

ADC Data Registers (ADLAR = 0)

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

*ADLAR = 1*

ADC Data Registers (ADLAR = 1)

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

These options are will be discussed in the posts related to timers. Those who have prior knowledge of timers can use it. The rest can leave it for now, we won't be using this anyway.

# ADC Initialization

The following code segment initializes the ADC.

```
void adc_init()
{
    // AREF = AVcc
    ADMUX = (1<<REFS0);

    // ADC Enable and prescaler of 128
    // 16000000/128 = 125000
    ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}
```

# Reading ADC Value

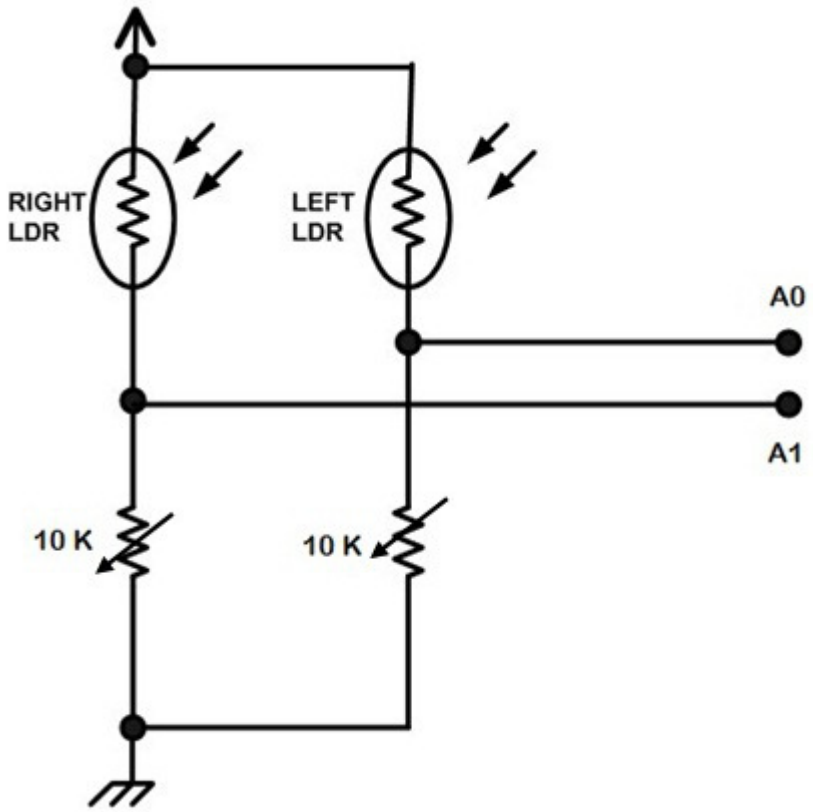
The following code segment reads the value of the ADC. Always refer to the register description above for every line of code.

```
uint16_t adc_read(uint8_t ch)
{
  // select the corresponding channel 0~7
  // ANDing with '7' will always keep the value
  // of 'ch' between 0 and 7
  ch &= 0b00000111;  // AND operation with 7
  ADMUX = (ADMUX & 0xF8)|ch; // clears the bottom 3 bits before ORing

  // start single convertion
  // write '1' to ADSC
  ADCSRA |= (1<<ADSC);

  // wait for conversion to complete
  // ADSC becomes '0' again
```

```
12  // till then, run loop continuously
13  while(ADCSRA & (1<<ADSC));
14
15  return (ADC);
16  }
17
18
19
```

# Physical Connections

Let's connect two LDRs (Light Dependent Resistors) to pins PA0 and PA1 respectively. The connection is as follows. The function of potentiometers is explained in a later section, **Sensor Calibration**. You can scroll down to it. 😉



LDR Connections

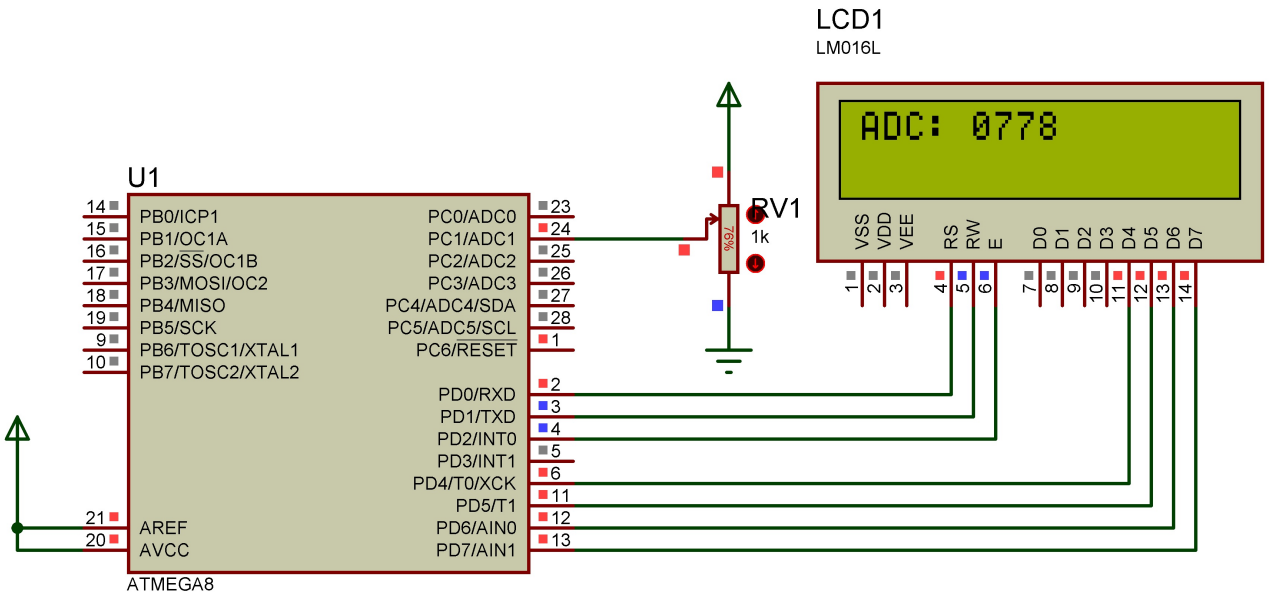
Now suppose we want to display the corresponding ADC values in an LCD. So, we also need to connect an LCD to our MCU. Read this post to know about LCD interfacing.

Since it is an LDR, it senses the intensity of light and accordingly change its resistance. The resistance decreases exponentially as the light intensity increases. Suppose we also want to light up an LED whenever the light level decreases. So, we can connect the LED to any one of the GPIO pins, say PC0.

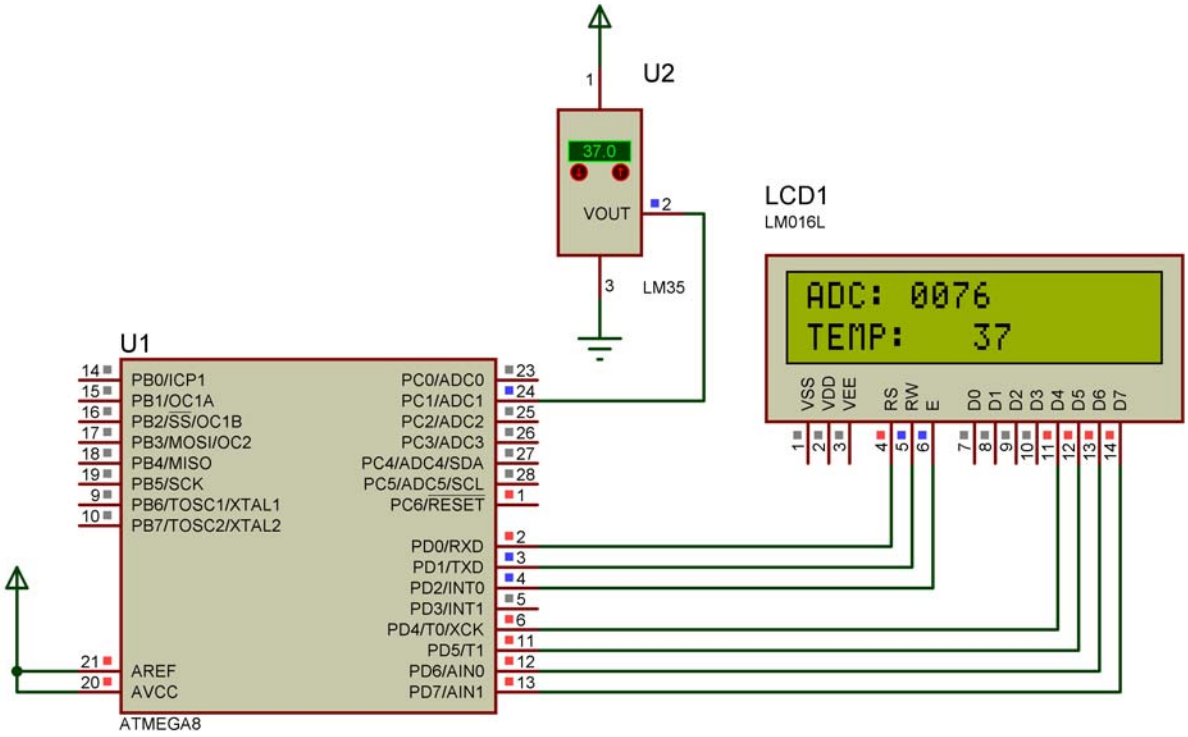
Note that since the ADC returns values in between 0 and 1023, for dark conditions, the value should be low (below 100 or 150) whereas for bright conditions, the value should be quite high (above 900).

Now let's write the complete code.

```c
//Program to convert Analog signal into Digital signal and displaying on LCD

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

#include "lcd.h" //Including LCD header file. Note: Please copy header file to current directory
#include "lcd.c"

int readADC(int channel) //Function to read Analog value and convert into digital
{
    channel = channel & 0x07; //Masking last 3 bits to select channel
    ADMUX = (1<<REFS0)|channel; //For Vref = AVcc and selecting channel
    ADCSRA = (1<<ADEN)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADSC); //Enable ADC, Prescalar = 64, Start
    Conversion
    while(!(ADCSRA & (1<<ADIF))); //Wait till conversion completes
    ADCSRA |= (1<<ADIF); //Clearing Flag bit
    return ADC; //Return digital value
}

void main() //Main Function
{
    char bf[15];
    int adc; //Declaring a variable
    DDRC = 0; //Configuring Port C as Input
    lcd_init(LCD_DISP_ON); //Initializing LCD
    while(1) //Forever Loop
    {
        adc = readADC(1); //Read values from ADC channel 1
        sprintf(bf,"ADC: %04d",adc); //Converting Integer values into String
        lcd_gotoxy(0,0); //Fixing position Row:0 Column: 0
        lcd_puts(bf); //Displaying on LCD
        _delay_ms(10); //delay function
    }
}
```



```
//Program to display Temperature (LM35) on LCD

/*Calculation Part:

Temp Sensor (LM35) Resolution: 10mV/C = 10/1000 = 1/100
ADC Resolution: 10 bit ie., 2^10 = 1024
Input Voltage = ADC Value * Step Size
Step Size = 5V / 1023
Temp Value  = Input Voltage / Temp Resolution
            = Input Voltage / (1/100);
            = Input Voltage * 100
            = ADC x (5V x 100)/1023 = (ADC Value * 500)/1023 */

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

#include "lcd.h" //Including LCD header file. Note: Please copy header file to current directory
#include "lcd.c"

unsigned int readADC(int channel) //Function to read Analog value and convert into digital
{
    channel = channel & 0x07; //Masking last 3 bits to select channel
    ADMUX = (1<<REFS0)|channel; //For Vref = AVcc and selecting channel
    ADCSRA = (1<<ADEN)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADSC); //Enable ADC, Prescalar = 64, Start
    Conversion
    while(!(ADCSRA & (1<<ADIF))); //Wait till conversion completes
    ADCSRA |= (1<<ADIF); //Clearing Flag bit
    return ADC; //Return digital value
}

void main() //Main Function
{
    char bf[15];
    unsigned int adc,temp; //Declaring a variable
    DDRC = 0; //Configuring Port C as Input
    lcd_init(LCD_DISP_ON); //Initializing LCD
    while(1) //Forever Loop
    {
        adc = readADC(1); //Read values from ADC channel 1
        temp = (adc * 500)/1023; //Temp Calculation
        sprintf(bf,"ADC: %04d\nTEMP: %4u",adc,temp); //Converting Integer values into String
        lcd_gotoxy(0,0); //Fixing position Row:0 Column: 0
        lcd_puts(bf); //Displaying on LCD
        _delay_ms(10); //delay function
    }
}
```


U2
1
37.0
VOUT
2
3
LM35
LCD1
LM016L
ADC: 0076
TEMP: 37
VSS
VDD
VEE
RS
RW
E
D0
D1
D2
D3
D4
D5
D6
D7
U1
PB0/ICP1
PB1/OC1A
PB2/SS/OC1B
PB3/MOSI/OC2
PB4/MISO
PB5/SCK
PB6/TOSC1/XTAL1
PB7/TOSC2/XTAL2
AREF
AVCC
PC0/ADC0
PC1/ADC1
PC2/ADC2
PC3/ADC3
PC4/ADC4/SDA
PC5/ADC5/SCL
PC6/RESET
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/T0/XCK
PD5/T1
PD6/AIN0
PD7/AIN1
ATMEGA8

# Introduction to AVR Timers.

## Timers

Timers are used everywhere. Without timers, you would end up nowhere! The range of timers vary from a few microseconds (like the ticks of a processor) to many hours (like the lecture classes 😔), and AVR is suitable for the whole range! AVR boasts of having a very accurate timer, accurate to the resolution of microseconds! This feature makes them suitable for timer applications. Let's see how.

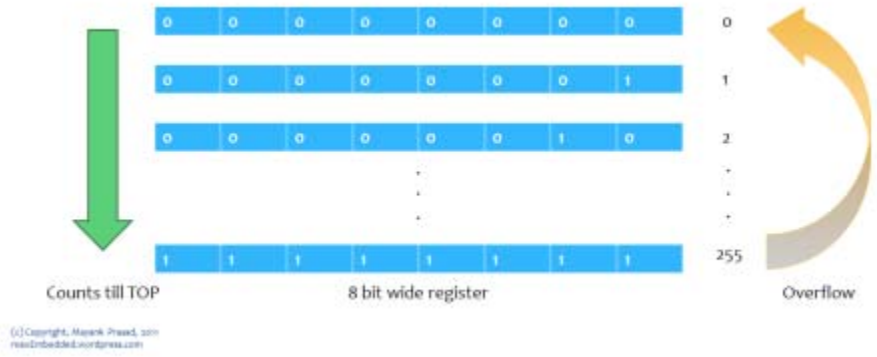
You come across timers everyday. Simplest example hangs on your wall or maybe tied around your wrist. You can say that they have a unique property to measure time. Everything in this world is synchronized with time. You wake up at, say, 6 o'clock; you work everyday for 8 hours; you need to drink water every 4 hours, etc. But the concept of timers isn't confined to your daily routines. Every electronic component works on a time base. This time base helps to keep all the work synchronized. Without a time base, you would have no idea as to *when* to do a particular thing.

Thus, timers is an important concept in the field of electronics. You can generate a time base using a timer circuit, using a microcontroller, etc. Since all the microcontrollers work at some predefined clock frequency, they all have a provision to set up timers.

AVR boasts of having a timer which is very accurate, precise and reliable. It offers loads of features in it, thus making it a vast topic. In this tutorial, we will discuss the basic concepts of AVR Timers. We will not be dealing with any code in this tutorial, just the concepts. The procedure of generating timers and their codes will be discussed in subsequent posts.

# Timers as registers

So basically, a timer is a register! But not a normal one. The value of this register increases/decreases automatically. In AVR, timers are of two types: 8-bit and 16-bit timers. In an 8-bit timer, the register used is 8-bit wide whereas in 16-bit timer, the register width is of 16 bits. This means that the 8-bit timer is capable of counting 2^8=256 steps from 0 to 255 as demonstrated below.




8 bit Counter

Similarly a 16 bit timer is capable of counting 2^16=65536 steps from 0 to 65535. Due to this feature, **timers are also known as counters**. Now what happens once they reach their MAX? Does the program stop executing? Well, the answer is quite simple. It returns to its initial value of zero. We say that the timer/counter **overflows**.

In ATMEGA32, we have three different kinds of timers:

- TIMER0 - 8-bit timer
- TIMER1 – 16-bit timer
- TIMER2 – 8-bit timer

The best part is that the timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention, which makes the timer quite accurate.

Apart from normal operation, these three timers can be either operated in normal mode, CTC mode or PWM mode. We will discuss them one by one.

# Timer Concepts

### Basic Concepts

Since childhood, we have been coming across the following formula:

$$Time\ Period = \frac{1}{Frequency}$$

Now suppose, we need to flash an LED every 10 ms. This implies that its frequency is 1/10ms = 100 Hz. Now let's assume that we have an external crystal XTAL of 4 MHz. Hence, the CPU clock frequency is 4 MHz. Now, as I said that the timer counts from 0 to TOP. For an 8-bit timer, it counts from 0 to 255 whereas for a 16-bit timer it counts from 0 to 65535. After that, they overflow. This value changes at every clock pulse.

Let's say the timer's value is zero now. To go from 0 to 1, it takes one clock pulse. To go from 1 to 2, it takes another clock pulse. To go from 2 to 3, it takes one more clock pulse. And so on. For F_CPU = 4 MHz, time period T = 1/4M = 0.00025 ms. Thus for every transition (0 to 1, 1 to 2, etc), it takes *only* 0.00025 ms!

Now, as stated above, we need a delay of 10 ms. This maybe a very short delay, but for the microcontroller which has a resolution of 0.00025 ms, its quite a long delay! To get an idea of *how long* it takes, let's calculate the timer count from the following formula:

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

Substitute *Required Delay = 10 ms* and *Clock Time Period = 0.00025 ms*, and you get ***Timer Count = 39999***. Can you imagine that? The clock has already ticked 39999 times to give a delay of *only* 10 ms!

Now, to achieve this, we definitely cannot use an 8-bit timer (as it has an upper limit of 255, after which it overflows). Hence, we use a 16-bit timer (which is capable of counting up to 65535) to achieve this delay.

## The Prescaler

Assuming F_CPU = 4 MHz and a 16-bit timer (MAX = 65535), and substituting in the above formula, we can get a maximum delay of 16.384 ms. Now what if we need a greater delay, say 20 ms? We are stuck?!

Well hopefully, there lies a solution to this. Suppose if we decrease the F_CPU from 4 MHz to 0.5 MHz (i.e. 500 kHz), then the clock time period increases to 1/500k = 0.002 ms. *Now* if we substitute *Required Delay = 20 ms* and *Clock Time Period = 0.002 ms*, we get ***Timer Count = 9999***. As we can see, this can easily be achieved using a 16-bit timer. At this frequency, a maximum delay of 131.072 ms can be achieved.

Now, the question is *how do we actually reduce the frequency?* This technique of frequency division is called ***prescaling***. We do not reduce the actual F_CPU. The actual F_CPU remains the same (at 4 MHz in this case). So basically, we *derive* a frequency from it to run the timer. Thus, while doing so, we divide the frequency and use it. There is a provision to do so in AVR by setting some bits which we will discuss later.

But don't think that you can use prescaler freely. It comes at a cost. **There is a trade-off between resolution and duration**. As you must have seen above, the overall duration of measurement has increased from a mere 16.384 ms to 131.072 ms. So has the resolution. The resolution has also increased from 0.00025 ms to 0.002 ms. This means each tick will take 0.002 ms. So, what's the problem with this? The problem is that the accuracy has decreased. Earlier, you were able to measure duration like 0.1125 ms accurately (0.1125/0.00025 = 450), but now you cannot (0.1125/0.002 = 56.25). The new timer can measure 0.112 ms and then 0.114 ms. No other value in between.

## Choosing Prescalers

Let's take an example. We need a delay of 184 ms (I have chosen any random number). We have F_CPU = 4 MHz. The AVR offers us the following prescaler values to choose from: 8, 64, 256 and 1024. A prescaler of 8 means the effective clock frequency will be F_CPU/8. Now substituting each of these values into the above formula, we get different values of timer value. The results are summarized as below:

Required Delay = 184 ms
F_CPU = 4 MHz

| Prescaler | Clock Frequency | Timer Count |
|---|---|---|
| 8 | 500 kHz | 91999 |
| 64 | 62.5 kHz | 11499 |
| 256 | 15.625 kHz | 2874 |
| 1024 | 3906.25 Hz | 717.75 |

Choosing Prescaler

Now out of these four prescalers, 8 cannot be used as the timer value exceeds the limit of 65535. Also, since the timer always takes up integer values, we cannot choose 1024 as the timer count is a decimal digit. Hence, we see that prescaler values of 64 and 256 are feasible. But out of these two, we choose 64 as it provides us with greater resolution. We can choose 256 if we need the timer for a greater duration elsewhere.

**Thus, we always choose prescaler which gives the counter value within the feasible limit** (255 or 65535) **and the counter value should always be an integer.**

We will discuss how to implement it in later posts.

# AVR Timers – TIMER0

Hello friends! Welcome back to the second part of the AVR Timers Series. In the previous post, we have discussed the basic concepts of AVR Timers. Let me summarize it:

- We have seen how timers are made up of registers, whose value automatically increases/decreases. Thus, the terms timer/counter are used interchangeably.
- In AVR, there are three types of timers – TIMER0, TIMER1 and TIMER2. Of these, TIMER1 is a 16-bit timer whereas others are 8-bit timers.
- We have seen how prescalers are used to trade duration with resolution.
- We have also discussed how to choose an appropriate value of a prescaler.
- And then, to finish off, we learnt about interrupts.

So, I will move towards its implementation directly. I have assumed that you have understood the concepts discussed above.

In this tutorial, we will learn to use TIMER0. Since timer is a peripheral, it can be activated by setting some bits in some registers. Instead of discussing all the registers at once, we will be discussing them as and when necessary. For those who are new to the term 'register', they can read about it from this page. To have an idea about AVR Peripherals, view this page (you need to scroll down a bit).

## Problem Statement

Let's define a problem statement for us. The simplest one being the LED flasher. Let's say, we need to flash an LED every 6 ms and we are have a CPU clock frequency of 32 kHz.

Well, I know that an LED flashing at every 6 ms will be always visible as *on* by our eye, but I could not find any simpler example which does not include prescalers. Take this as a demonstration.

# Methodology

Now, as per the following formula, with a clock frequency of 32 kHz and 8-bit counter, the maximum delay possible is of 8 ms. This is quite low (for us, but not for the MCU). Hence for a delay of 6 ms, we need a timer count of 191. This can easily be achieved with an 8-bit counter (MAX = 255).

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

Thus, what we need to do is quite simple. We need to keep a track of the counter value. As soon as it reaches 191, we toggle the LED value and reset the counter. For this, we need the help of the following registers.

### TCNT0 Register

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

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | TCNT0[7:0] | | | | | | | | TCNT0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

TCNT0 Register

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

Now we know where the counter value lies. But this register won't be activated unless we activate the timer! Thus we need to set the timer up. How? Read on…

## TCCR0 Register

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

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 | TCCR0 |
| Read/Write | W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

TCCR0 Register

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

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

Clock Select Bit Description

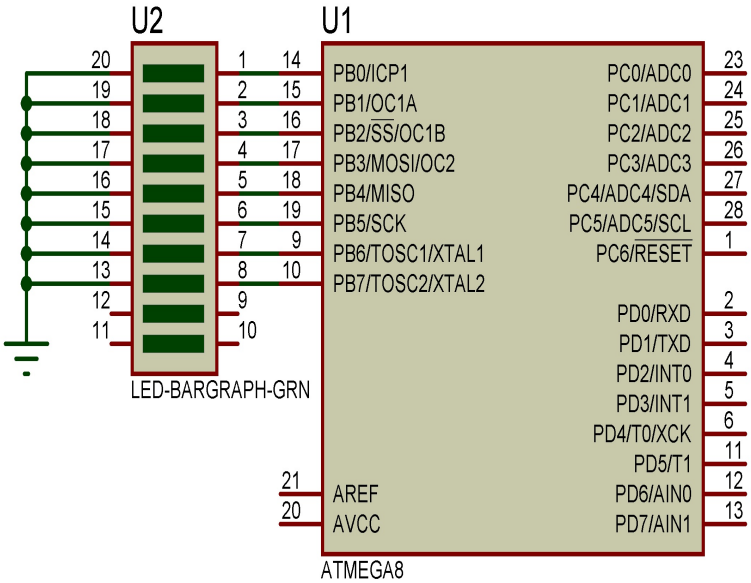
For this problem statement, we choose **No Prescaling**. Ignore the bits highlighted in grey. We will be using it later in this tutorial. Thus, we initialize the counter as:

```
TCCR0 |= (1 << CS00);
```

Please note that if you do not initialize this register, all the bits will remain as zero and the timer/counter will remain stopped.

Thus now we are ready to write a code for this.

```
// Program to Blink LED @ particular Rate using TIMER 0

/* Calculation Part:

XTAL = 1 MHz
Clock Cycle = 1
MFreq = XTAL / CC = 1 MHz
Prescalar = 64
CFreq = MFreq / 64 = 15625 Hz
Clock Time = 1/CFreq = 64 us

Timer0: 8 bit ie., Max Count = 2^8 = 256 (0-255)
Max Delay = Max Count * CTP = 16384 us

Recq Under Range Delay: 10000 us = 10 ms

Timer Count = (Recq Delay / CTP) - 1 = (10000/64)-1 = 155.25 = 155
Initia Value: ((Max Count)-1)-Timer Count = 255-155 = 100 */

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

void delay_using_timer(int time)
{
    int i;
    for(i=0;i<100*time;i++) //Single loop gives 10 ms delay, therefore, 100 times give 1000 ms
    or 1 sec delay)
    {
        TCNT0 = 100; //Timer Count Register. Refer Calculation part
        TCCR0 = (1<<CS01)|(1<<CS00); //Prescalar: 64
        while(!(TIFR & ( 1<<TOV0))); //Wait for Flag bit to go high ie., 10
        ms TIFR |= (1<<TOV0); //Clearing Flag Bit
    }
}

void main() //Main Function
{
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        PORTB = 0xFF; //LED ON
        delay_using_timer(2); //2 Sec Delay
        PORTB = 0; //LED OFF
        delay_using_timer(1); //1 Sec Delay
    }
}
```



# AVR Timers – TIMER1

Hello folks! Welcome back! In this tutorial, we will come across TIMER1 of the AVR. I hope that you have read and understood the previous posts:

- Introduction to AVR Timers
- AVR Timers – TIMER0

So basically, in this tutorial, we will do whatever we did in the previous one. In the TIMER0 tutorial, we generated a timer running at the CPU frequency. We then modified the code to include prescalers, and once again modified the code to include interrupts.

Now that you are aware of the concepts, we will deal with TIMER1 in a short and snappy way. Whatever we did in the previous TIMER0 tutorial, we will do the same here. Thus, we will discuss only one problem statement which will include both, prescalers and interrupts.

Once we are done with this, we can proceed to the CTC and PWM modes of operations in subsequent posts.

# Problem Statement

Okay, let's make it loud and clear. We need to flash an LED every 2 seconds, i.e. at a frequency of 0.5 Hz. We have an XTAL of 16 MHz.

### Methodology – Using prescaler and interrupt

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

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

Given that we have a CPU Clock Frequency of 16 MHz. At this frequency, and using a 16-bit timer (MAX = 65535), the maximum delay is 4.096 ms. It's quite low. Upon using a prescaler of 8, the timer frequency

reduces to 2 MHz, thus giving a maximum delay of 32.768 ms. Now we need a delay of 2 s. Thus, 2 s ÷ 32.768 ms = 61.035 ≈ 61. This means that the timer should overflow 61 times to give a delay of approximately 2 s.

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

# TCCR1B Register

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

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

TCCR1B Register

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

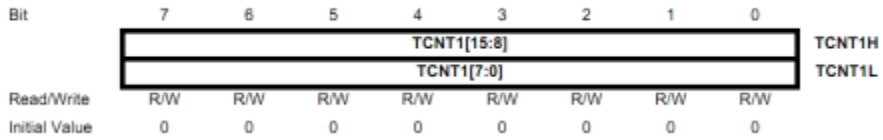
| CS12 | CS11 | CS10 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{I/O}$/1 (No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |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

Clock Select Bits Description

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

# TCNT1 Register

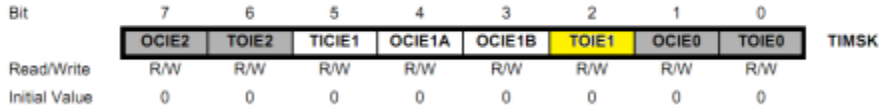
The **Timer/Counter1** - TCNT1 Register is as follows. It is 16 bits wide since the TIMER1 is a 16-bit register. **TCNT1H** represents the HIGH byte whereas **TCNT1L** represents the LOW byte. The timer/counter value is stored in these bytes.




TCNT1 Register

# TIMSK Register

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

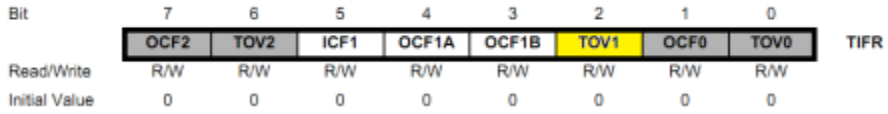



TIMSK Register

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

# TIFR Register

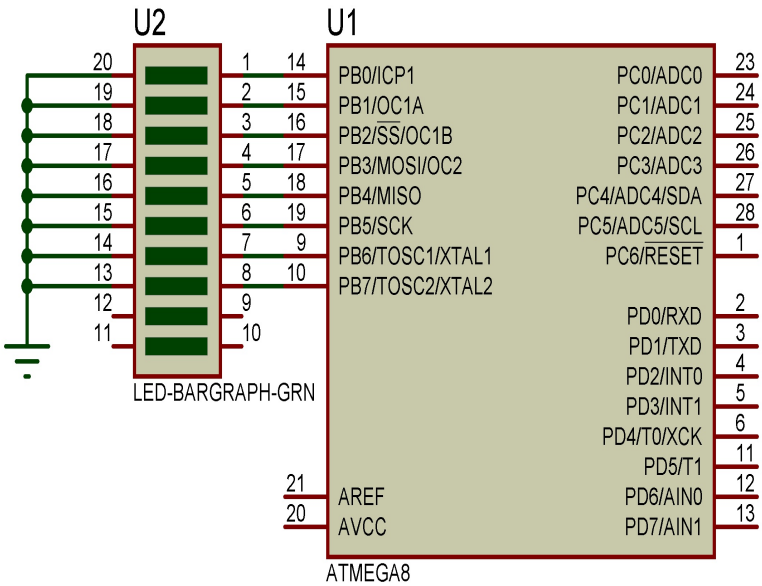
The **Timer/Counter Interrupt Flag Register** – TIFR is as follows.




TIFR Register

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

```
// Program to Blink LED @ particular Rate using TIMER 1

/* Calculation Part:

XTAL = 1 MHz
Clock Cycle = 1
MFreq = XTAL / CC = 1 MHz
Prescalar = 64
CFreq = MFreq / 64 = 15625 Hz
Clock Time = 1/CFreq = 64 us

Timer1: 16 bit ie., Max Count = 2^16 = 65536 (0-65535)
Max Delay = Max Count * CTP = 4194304 us = 4.19 sec

Recq Under Range Delay: 1000000 us = 1 sec

Timer Count = (Recq Delay / CTP) - 1 = (1000000/64)-1 = 15624
Initia Value: ((Max Count)-1)-Timer Count = 65535-15624 = 49911 */

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

void delay_using_timer(int time)
{
    int i;
    for(i=0;i<time;i++) //Single loop gives 1 sec delay
    {
        TCNT1 = 49911; //Timer Count Register. Refer Calculation part
        TCCR1B = ( 1<<CS11)|(1<<CS10); //Prescalar: 64
        while(!(TIFR & ( 1<<TOV1))); //Wait for Flag bit to go high ie., 1s
        TIFR |= (1<<TOV1); //Clearing Flag Bit
    }
}

void main() //Main Function
{
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        PORTB = 0xFF; //LED ON
        delay_using_timer(2); //2 Sec Delay
        PORTB = 0; //LED OFF
        delay_using_timer(1); //1 Sec Delay
    }
}
```



# AVR Timers – TIMER2

Hello friends! Welcome to the tutorial on the TIMER2 of AVR ATMEGA16/32. I hope that you have already come across and read the following posts, in which the basic concepts and applications of AVR Timers are discussed.

- Introduction to AVR Timers
- AVR Timers – TIMER0
- AVR Timers – TIMER1

In this post, we will discuss about TIMER2. Since TIMER2 is an 8-bit timer (like TIMER0), most of the registers are similar to that of TIMER0 registers. Apart from that, TIMER2 offers a special feature which other timers don't – **Asynchronous Operation**. We will discuss about it later.

Since you are already aware of the concepts (I assume so, or else refer to my previous posts), we will proceed the way we did in TIMER1 tutorial. We will implement both prescalers and interrupts in the same problem statement.

# Problem Statement

We need to flash an LED every 50 ms. We have an XTAL of 16 MHz. This is the same problem statement that we discussed in TIMER0 (the last one). We will implement the same using TIMER2.

### Methodology – Using Prescaler and Interrupt

As discussed in the TIMER0 tutorial, we use a prescaler of 256. For this, the overflow time is 4.096 ms. Thus the timer should overflow 12 times (MAX = 255) and count up to 53 in the 13th iteration, and then reset the timer. The formula used is as follows:

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

Now let's have a look at the TIMER2 registers.

# 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 | |

TCCR2 Register

Since we will be dealing with the CTC mode later, 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!

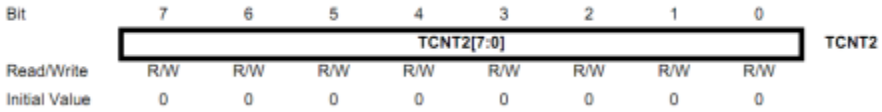
| CS22 | CS21 | CS20 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{T2S}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{T2S}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{T2S}$/32 (From prescaler) |
| 1 | 0 | 0 | $clk_{T2S}$/64 (From prescaler) |
|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

Clock Select Bit Description

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

# TCNT2 Register

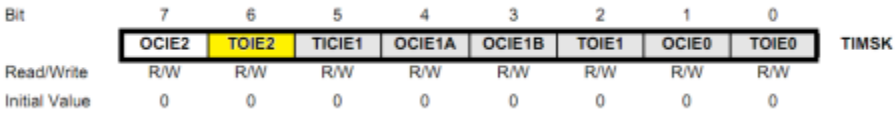
In the **Timer/Counter Register** – TCNT2, the value of he timer is stored. Since TIMER2 is an 8-bit timer, this register is 8 bits wide.




TCNT2 Register

# TIMSK Register

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

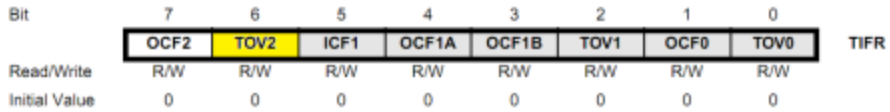



TIMSK Register

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.

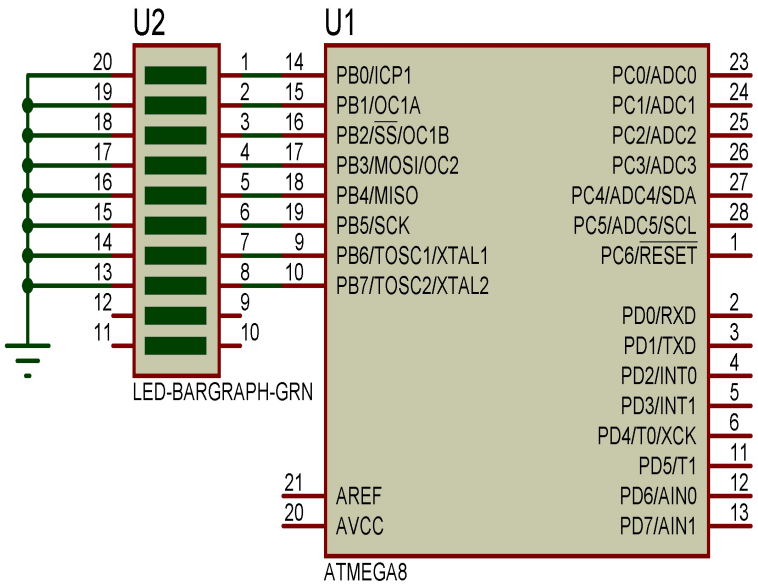


| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | OCF2 | TOV2 | ICF1 | OCF1A | OCF1B | TOV1 | OCF0 | TOV0 | TIFR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

TIFR Register

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.

```
// Program to Blink LED @ particular Rate using TIMER 2

/* Calculation Part:

XTAL = 1 MHz
Clock Cycle = 1
MFreq = XTAL / CC = 1 MHz
Prescalar = 64
CFreq = MFreq / 64 = 15625 Hz
Clock Time = 1/CFreq = 64 us

Timer2: 8 bit ie., Max Count = 2^8 = 256 (0-255)
Max Delay = Max Count * CTP = 16384 us = 16.38ms

Recq Under Range Delay: 10000 us = 10 ms

Timer Count = (Recq Delay / CTP) - 1 = (10000/64)-1 = 155.25 = 155
Initia Value: ((Max Count)-1)-Timer Count = 255-155 = 100 */

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

void delay_using_timer(int time)
{
    int i;
    for(i=0;i<100*time;i++) //Single loop gives 1 sec delay
    {
        TCNT2 = 100; //Timer Count Register. Refer Calculation part
        TCCR2 = (1<<CS22); //Prescalar: 64
        while(!(TIFR & (1<<TOV2))); //Wait for Flag bit to go high ie., 10 ms
        TIFR |= (1<<TOV2); //Clearing Flag Bit
    }
}

void main() //Main Function
{
    DDRB = 0xFF; //Configuring PORTB as Output
    while(1)
    {
        PORTB = 0xFF; //LED ON
        delay_using_timer(2); //2 Sec Delay
        PORTB = 0; //LED OFF
        delay_using_timer(1); //1 Sec Delay
    }
}
```



# AVR Timers – CTC Mode

Hello friends! Welcome to the another tutorial on AVR Timers. Till now, we have covered the following topics in AVR Timers:

- Introduction to AVR Timers
- 8-bit TIMER0
- 16-bit TIMER1
- 8-bit TIMER2

The basic concepts of timers and its applications have been discussed in earlier posts. In this post, we will discuss about a special mode of operation – **Clear Timer on Compare (CTC) Mode**.

# CTC Mode

So till now, we have dealt with the basic concepts. We had two timer values with us – **Set Point** (SP) and **Process Value** (PV). In every iteration, we used to compare the process value with the set point. Once the process value becomes equal (or exceeds) the set point, the process value is reset. The following code snippet explains it:

```
max = 39999;   // max timer value set  <--- set point

// some code here
// ...
// ...
// ...

// TCNT1 <--- process value
if (TCNT1 >= max)   // process value compared with the set point
{
    TCNT1 = 0;      // process value is reset
}
```

```
// ...
```

Here, we have used the example of TIMER1. Since TIMER1 is a 16-bit timer, it can count upto a maximum of 65535. Here, what we desire is that the timer (process value) should reset as soon as its value becomes equal to (or greater than) the set point of 39999.

So basically, the CTC Mode implements the same thing, but unlike the above example, it implements it in hardware. Which means that we no longer need to worry about comparing the process value with the set point every time! This will not only avoid unnecessary wastage of cycles, but also ensure greater accuracy (i.e. no missed compares, no double increment, etc).

Hence, this comparison takes place in the hardware itself, inside the AVR CPU! Once the process value becomes equal to the set point, a flag in the status register is set and the timer is reset *automatically!* Thus goes the name – CTC – *Clear* Timer on *Compare!* Thus, all we need to do is to take care of the flag, which is much more faster to execute.

Let us analyze this CTC Mode in detail with the help of a problem statement.

# Problem Statement

Let's take up a problem statement to understand this concept. We need to flash an LED every 100 ms. We have a crystal of XTAL 16 MHz.

## Methodology – Using CTC Mode

Before proceeding any further, let's jot down the formula first. I also recommend you to read my TIMER0 tutorial in order to understand this better. I won't be revising the basic concepts here, just their application. $$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$ Now, given XTAL = 16 MHz, with a prescaler of 64, the frequency of the clock pulse reduces to 250 kHz. With a Required Delay = 100 ms, we get the Timer Count to be equal to 24999. Up until

now, we would have let the value of the timer increment, and check its value every iteration, whether it's equal to 24999 or not, and then reset the timer. *Now,* the same will be done in hardware! We won't check its value every time in software! We will simply check whether the flag bit is set or not, that's all. *Confused, eh?* Well, don't worry, just read on! 😉

Okay, so now let me introduce you to the register bits which help you to implement this CTC Mode.

## TCCR1A and TCCR1B Registers

The **Timer/Counter1 Control Register A** – TCCR1A Register is as follows:

| 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 | |

TCCR1A 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

You are already aware of the **Clock Select Bits – CS12:0** in TCCR1B (if not, view the TIMER1 post, scroll down a bit and you will find it there). Hence, right now, we are concerned with the **Wave Generation Mode Bits – WGM13:0**. As you can see, these bits are spread across both the TCCR1 registers (A and B). Thus we need to be a bit careful while using them. Their selection is as follows:

| 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 |

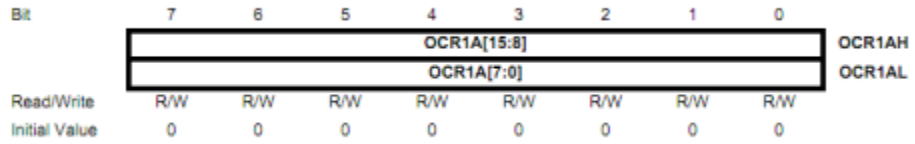
Wave Generation Mode Bit Description

We can see that there are two possible selections for CTC Mode. Practically, both are the same, except the fact that we store the timer compare value in different registers. Right now, let's move on with the first option (0100). Thus, the initialization of TCCR1A and TCCR1B is as follows.

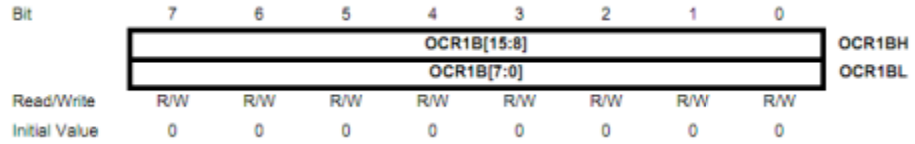
```
TCCR1A |= 0;
// not required since WGM11:0, both are zero (0)

TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);
// Mode = CTC, Prescaler = 64
```

## OCR1A and OCR1B Registers

Now that we have set the CTC Mode, we must tell the AVR to reset the timer as soon as its value reaches *such and such value.* So, the question is, how do we set *such and such values*? The **Output Compare Register 1A** – OCR1A and the **Output Compare Register 1B** – OCR1B are utilized for this purpose.



OCR1A Register



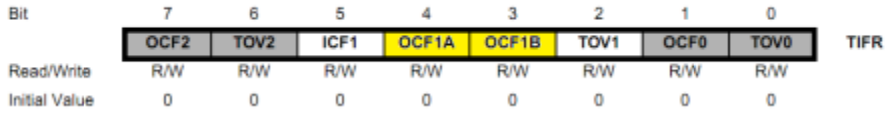

OCR1B Register

Since the compare value will be a 16-bit value (in between 0 and 65535), OCR1A and OCR1B are 16-bit registers. In ATMEGA16/32, there are two CTC channels – A and B. We can use any one of them or both. Let's use OCR1A.

```
OCR1A = 24999;   // timer compare value
```

## TIFR Register

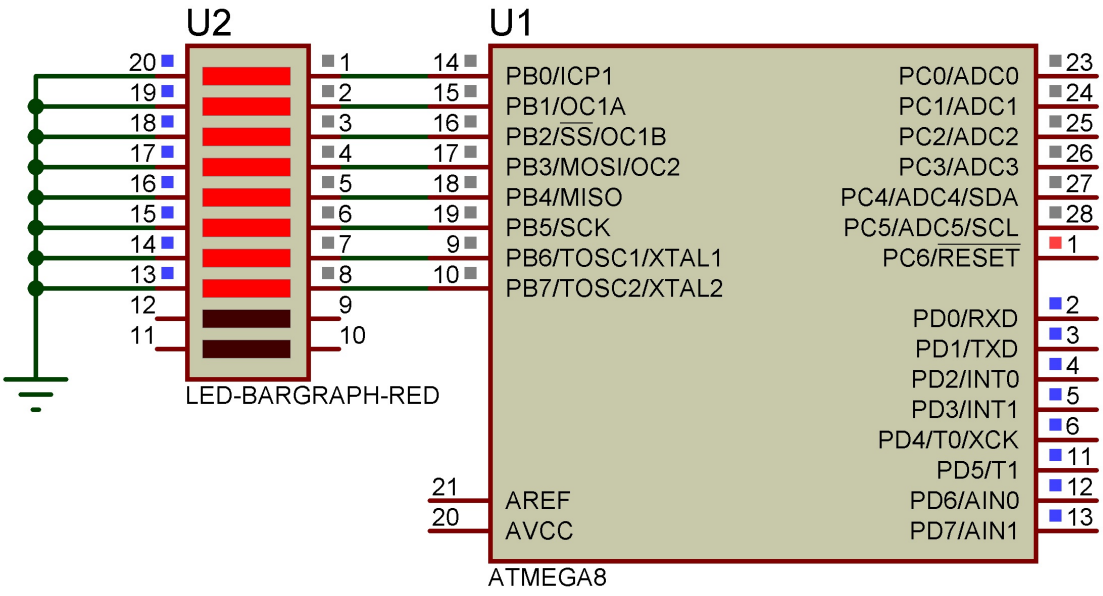
The **Timer/Counter Interrupt Flag Register** – TIFR is as follows. It is a common register to all the timers.




TIFR Register

We are interested in **Bit 4:3 – OCF1A:B – Timer/Counter1, Output Compare A/B Match Flag Bit**. This bit is set (one) by the AVR whenever a match occurs i.e. TCNT1 becomes equal to OCR1A (or OCR1B). It is cleared automatically whenever the corresponding Interrupt Service Routine (ISR) is executed. Alternatively, it can be cleared by writing '1′ to it!

```
//Program to blink LED @ 1sec using Timer 1 CTC Mode

//1s delay with 1MHz using prescalar 64
#include<avr/io.h>
void timer_init()
{
    //TCCR1A=0; //CTC Mode: WGM13:10 - 0100
    TCCR1B|=(1<<WGM12)|(1<<CS11)|(1<<CS10); //ctc mode, 64 prescalar for 1 Mhz
    OCR1A=15624; //OCR1A:B - 16 bit register (Output Compare Register), can use either
                 //cfreq=1000000/64=15625KHz, Clock time = 64us, Timer Count(1s)=1s/64us-1=15624
    TCNT1=0;
}
void main()
{
    int count=0;
    DDRB=DDRD=255;
    timer_init();
    while(1)
    {
        //if(TCNT1>=OCR1A)
        if(TIFR & (1<<OCF1A)) //flag will be raised when TCNT becomes equal to OCR1A
        {
            PORTB=~PORTB;
            //TIFR=(0<<OCF1A);
            TIFR|=(1<<OCF1A); //clear flag writing 1 to it as per datasheet as we are not using
            interrupt
        //  TCNT1=0;
        }

    }
}
```



# AVR Timers – PWM Mode – Part I

Pulse Width Modulation (PWM) is a very common technique in telecommunication and power control. Learn how easily you can do so using AVRs! This post discusses all the necessary theoretical concepts related to PWM. Here it goes…

Welcome back! Till now, in the AVR Timers, we have discussed regarding the timer concepts, prescalers, interrupts, ctc mode, etc. Refer to the following tutorials for them.

- Introduction to AVR Timers
- AVR TIMER0
- AVR TIMER1
- AVR TIMER2
- AVR Timers – CTC Mode

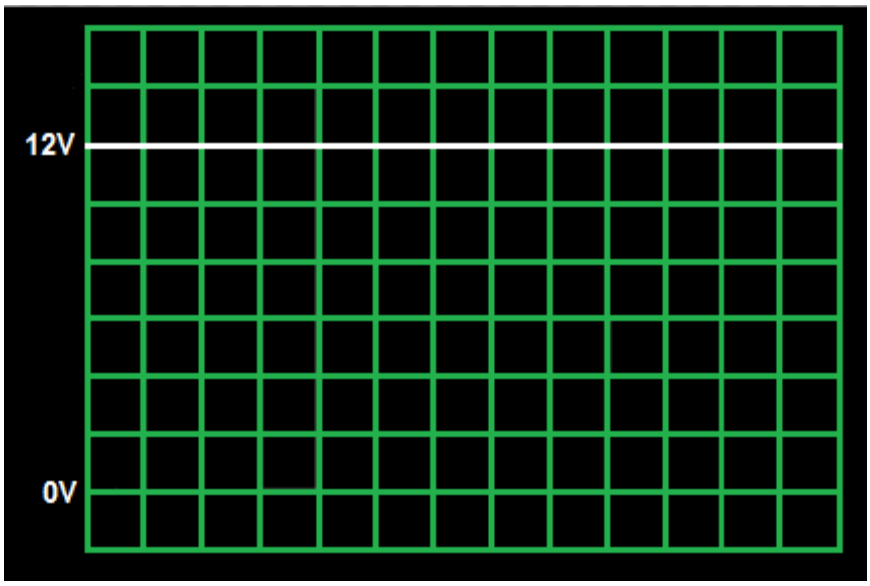
In this tutorial, I will introduce you to another mode of AVR Timers – PWM Mode.

# Introduction

Let us assume that a DC motor is connected to the power supply as follows.

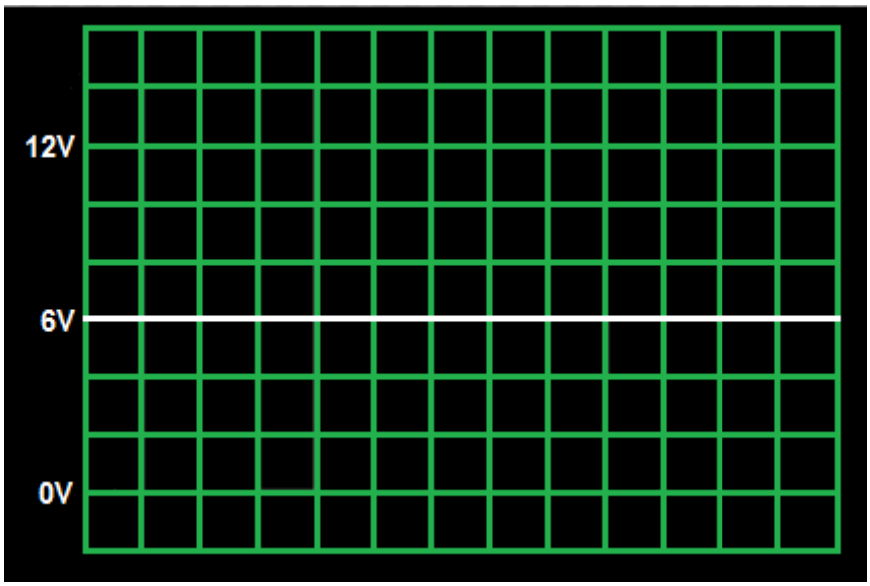
Motor 12V 300rpm

The motor is rated 12V/300rpm. This means that (assuming ideal conditions) the motor will run at 300 rpm only when 12V DC is supplied to it. If we apply 6V, the motor will run at only 150 rpm. For more details regarding controlling DC motor using AVR, view this. Now let us provide the following signal (12V DC) to the motor.



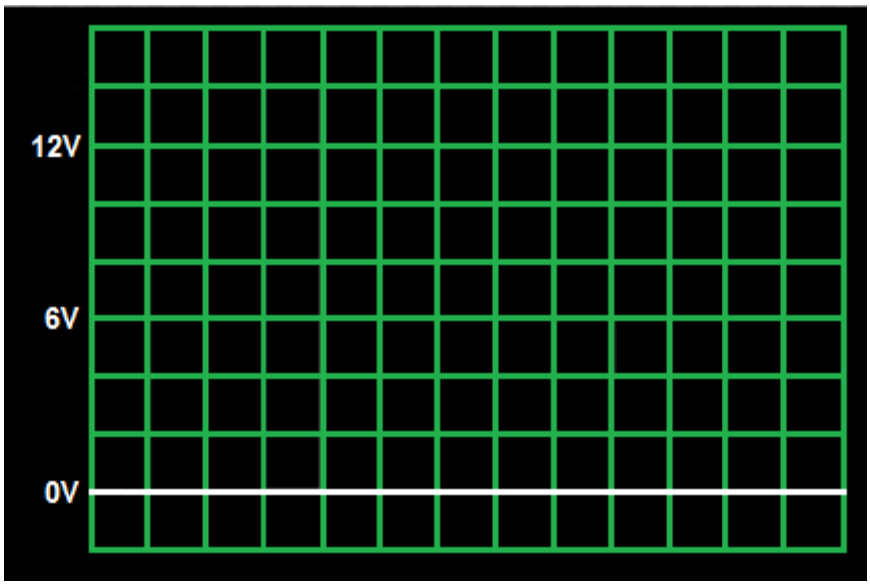

12V DC Supply

The motor will rotate at 300 rpm. Now let us change the voltage level as follows (6V DC).



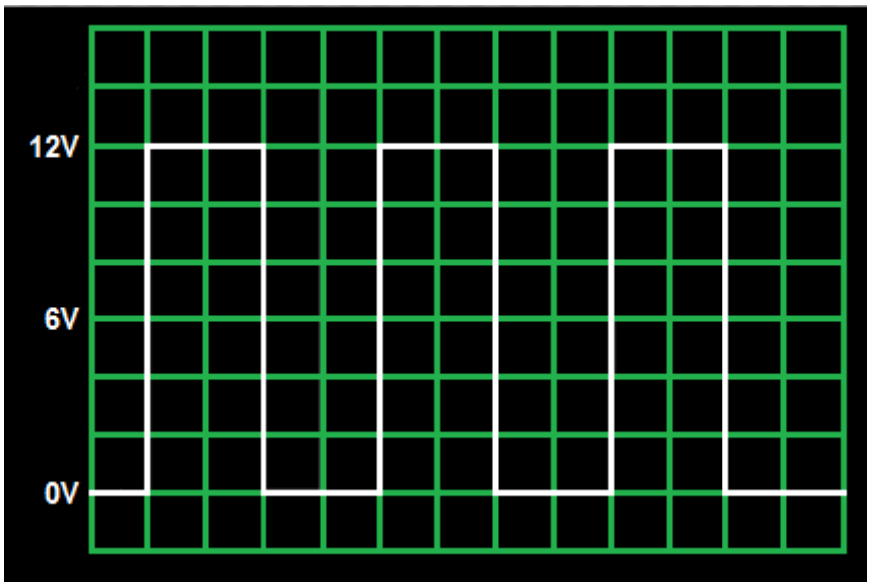

6V DC Supply

We find that the motor rotates at 150 rpm. Now let us change the voltage level once again as follows (0V DC).




0V DC Supply

This time, unsurprisingly, the motor doesn't run at all. Okay, so let's make it more interesting. *What if* we provide the following supply to the motor.



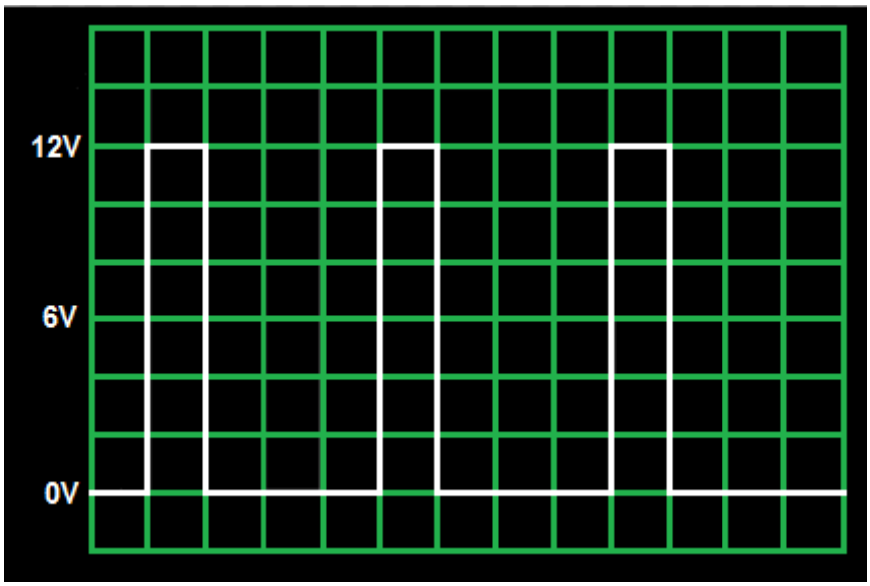

50% Duty Cycle PWM

Now how will the motor respond? Will it start for some time, then after some time it stops, then starts, then stops, and then starts again, and so on. Or will it get confused and simply blast? 😁

Well, each and every body in this world has some inertia. Say the motor above rotates whenever it is powered on. As soon as it is powered off, it will *tend* to stop. But it doesn't stop

immediately, it takes some time. But before it stops completely, it is powered on again! Thus it starts to move. But even now, it takes some time to reach its full speed. But before it happens, it is powered off, and so on. Thus, the overall effect of this action is that the motor rotates continuously, but at a lower speed. In the above case, the motor will behave exactly as if a 6V DC is supplied to it, i.e. rotate at 150 rpm!

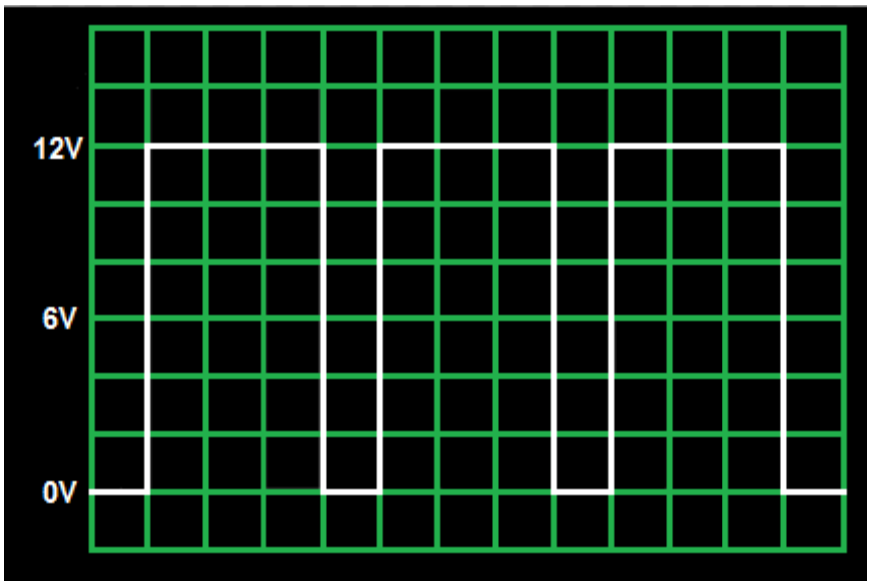
Okay, so now, let's modify the signal as follows.




25% Duty Cycle PWM

*Now* what happens? Yes! You guessed it right! (I hope so ) Since the on-time is less than the off-time, the effective speed of the motor reduce. In this case, the speed becomes 75 rpm (since off-time = 3 times on-time, i.e. speed = 300/4 = 75 rpm).

Now it's your turn to say what happens in this case:




75% Duty Cycle PWM

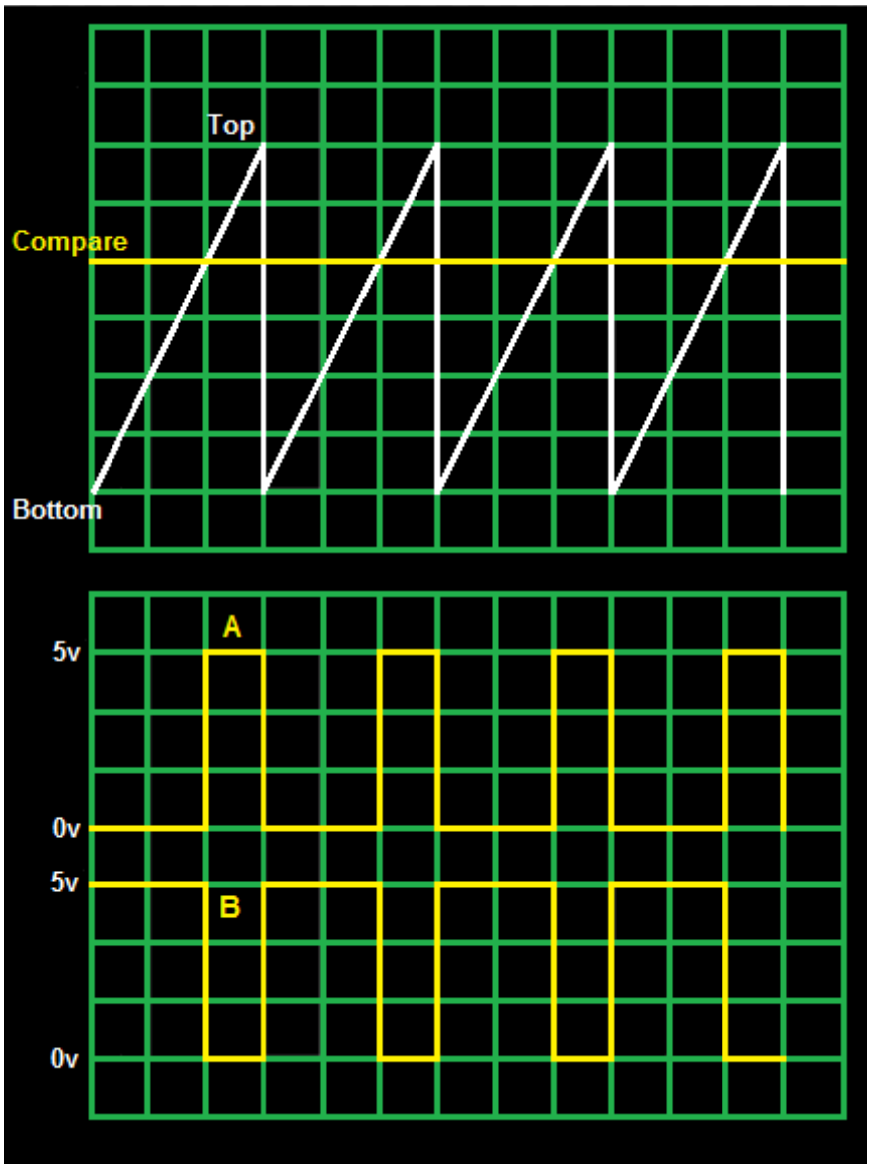
This is what we call **Pulse Width Modulation**, commonly known as **PWM**.

# PWM – Pulse Width Modulation

PWM stands for Pulse Width Modulation. It is basically a modulation technique, in which the width of the carrier pulse is varied in accordance with the analog message signal. As described above, it is commonly used to control the power fed to an electrical device, whether it is a motor, an LED, speakers, etc.

## PWM Generation

The simplest way to generate a PWM signal is by comparing the a predetermined waveform with a fixed voltage level as shown below.

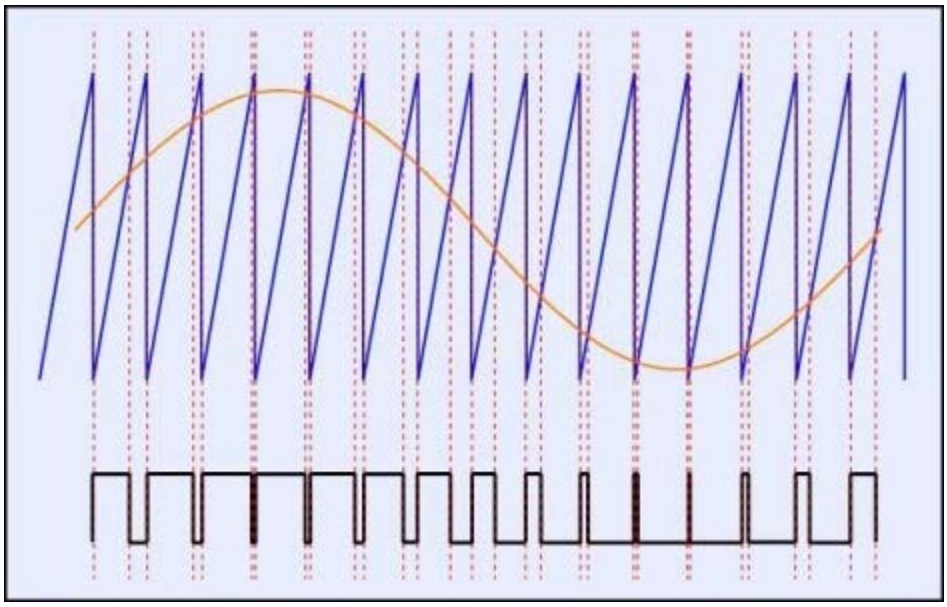



Compare PWM

In the diagram shown above, we have a predetermined waveform, sawtooth waveform. We *compare* this waveform with a fixed DC level. It has three **compare output modes** of operation:

- **Inverted Mode -** In this mode, if the waveform value is greater than the compare level, then the output is set high, or else the output is low. This is represented in figure A above.
- **Non-Inverted Mode -** In this mode, the output is high whenever the compare level is greater than the waveform level and low otherwise. This is represented in figure B above.
- **Toggle Mode -** In this mode, the output toggles whenever there is a compare match. If the output is high, it becomes low, and vice-versa.

But it’s always not necessary that we have a fixed compare level. Those who have had exposure in the field of analog/digital communication must have come across cases where a sawtooth carrier wave is compared with a sinusoidal message signal as shown below.



PWM Modulation

Here you can *clearly* see and understand the meaning of ‘width’ in Pulse *Width* Modulation!

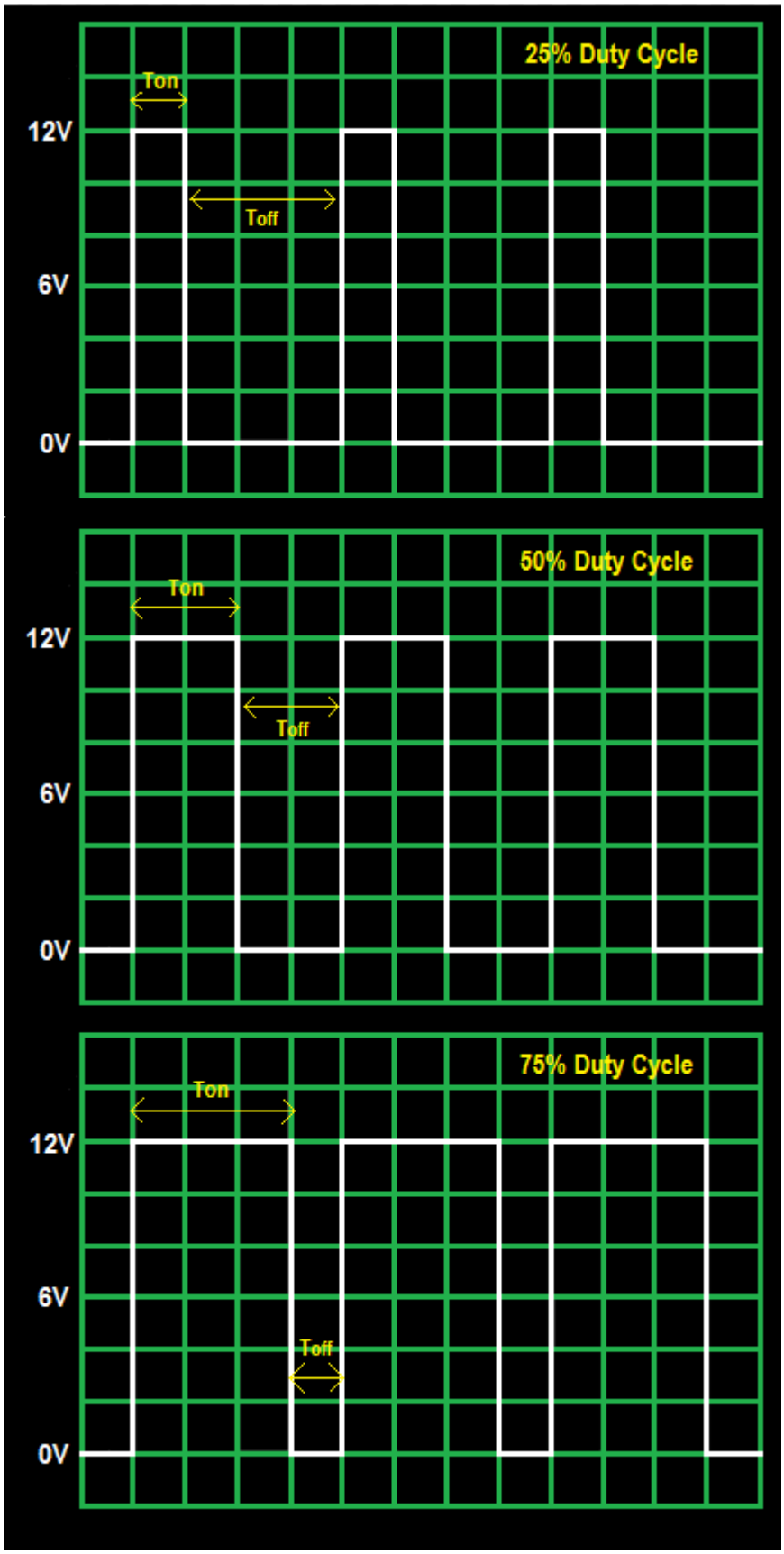
PWM can also be generated by making analog circuits like the one described here.

# Duty Cycle

The Duty Cycle of a PWM Waveform is given by

$$Duty\ Cycle = \frac{T_{on}}{T_{on} + T_{off}} \times 100\ \%$$

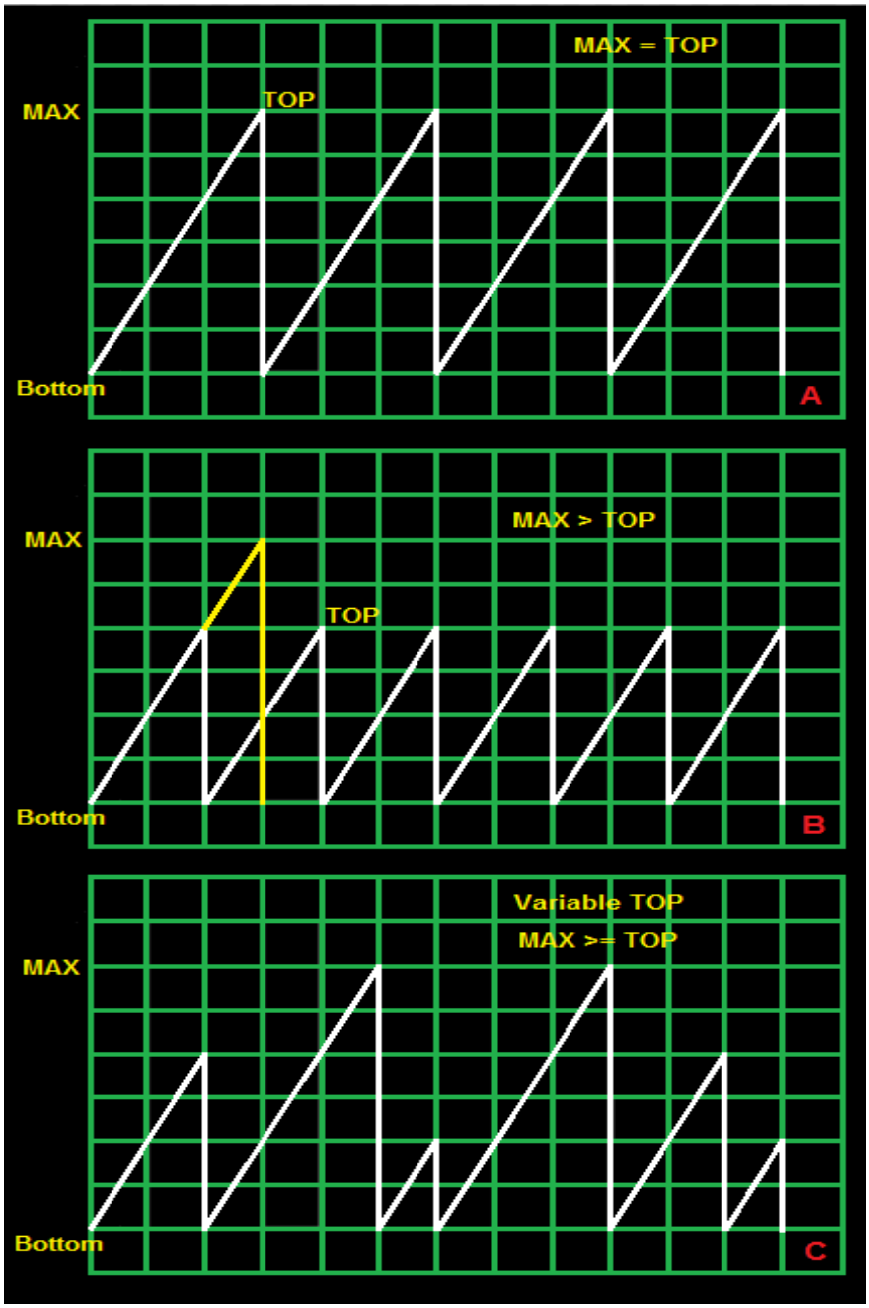
This is clarified in the following diagram.


25% Duty Cycle
Ton
Toff
12V
6V
0V
50% Duty Cycle
Ton
Toff
12V
6V
0V
75% Duty Cycle
Ton
Toff
12V
6V
0V

Duty Cycle Explained

# Timer Concepts – Revisited

In this section, we will revise some important and necessary concepts related to timers. Consider the following timer diagram.




Fixed and Variable TOP in Timers

We are very well aware that the AVR provides us with an option of 8 and 16 bit timers. 8bit timers count from 0 to 255, then back to zero and so on. 16bit timers count from 0 to 65535, then back to zero. Thus for a 8bit timer, MAX = 255 and for a 16bit timer, MAX = 65535.

The timer *always* counts from 0 to TOP, then overflows back to zero. In figure A shown above, TOP = MAX. Now, I guess you all are familiar with timers in CTC Mode, in which you can clear the timer whenever a compare match occurs. Due to this, the value of TOP can be reduced as shown in figure B. The yellow line shows how the timer would have gone in normal mode. Now, the CTC Mode can be extended to introduce variable TOP as shown in figure C (however there isn't any practical utility of this).

**TOP never exceeds MAX. TOP <= MAX.**

Now that you are aware of the terminologies of TOP, BOTTOM and MAX, we can proceed to the different modes of operation.

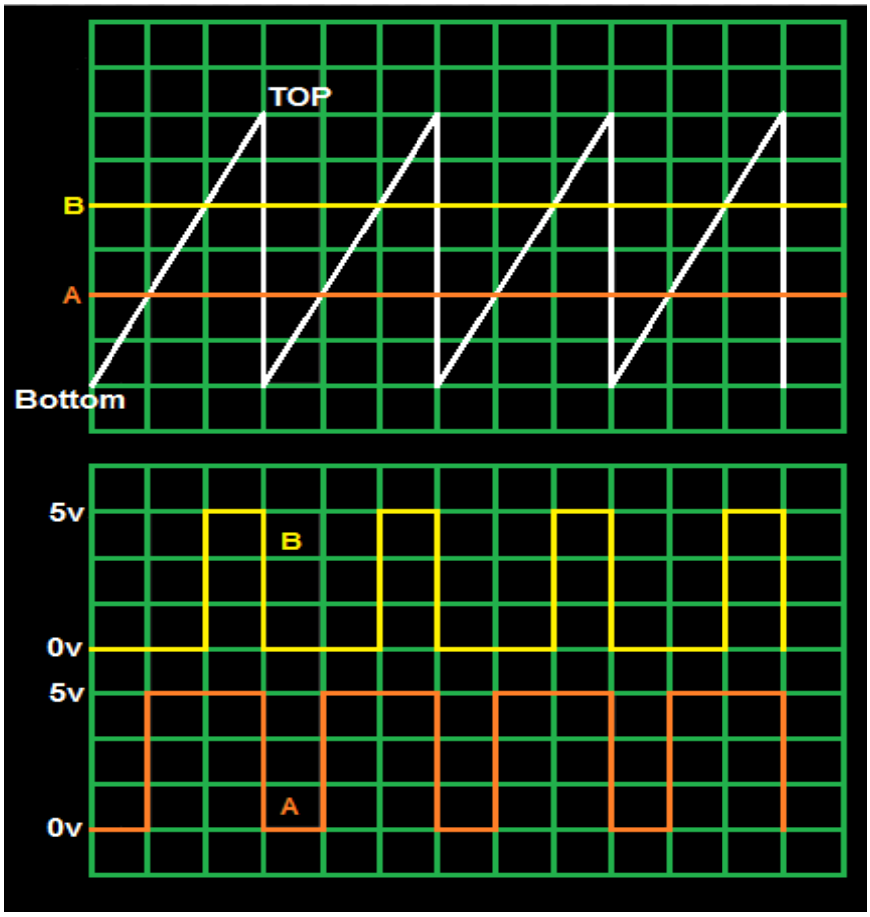
# PWM Modes of Operation

In general, there are three modes of operation of PWM Timers:

- Fast PWM
- Phase Correct PWM
- Frequency and Phase Correct PWM

## Fast PWM

Consider the following diagram.




Fast PWM

In simple terms, *this* is Fast PWM! We have a sawtooth waveform, and we compare it with a fixed voltage level (say A), and thus we get a PWM output as shown (in A). Now suppose we increase the compare voltage level (to, say B). In this case, as we can see, the pulse width has reduced, and hence the duty cycle. *But,* as you can see, both the pulses (A and B) end at the same time irrespective of their starting time.

In this mode, since sawtooth waveform is used, the timer counter TCNTn (n = 0,1,2) counts from BOTTOM to TOP and then it is simply allowed to overflow (or cleared at a compare match) to BOTTOM.

and Phase Correct PWM.

# Making Choices

Now that you are familiar with all the PWM concepts, it's upto you to decide

- Which timer to choose?
- Which mode of operation to choose?
- Which compare output mode to choose?

## Choosing Timer

In AVR, PWM Mode is available in all timers. TIMER0 and TIMER2 provide 8bit accuracy whereas TIMER1 provides 16bit accuracy. In 8bit accuracy, you have 256 individual steps, whereas in 16bit accuracy, you have 65536 steps.

Now suppose you want to control the speed of a DC motor. In this case, having 65536 steps is totally useless! Thus we can use an 8bit timer for this. Even 8bit is too much, but there is no other choice. Obviously there isn't much difference in speed between 123/256th and 124/256th of full speed in case of a motor! But if you use servo motors, you have to use 16bit timer. More on that later.

If you need quite high resolution in your application, go for 16bit timer.

## Choosing Mode of Operation

If you want to control the speed of DC motors or brightness of LEDs, go for any one of them. But if you are using it for telecommunication purposes, or for signal sampling, fast PWM would be better. For general applications, phase correct PWM would do.

## Choosing Compare Output Modes

Out of the three modes, inverted, non-inverted and toggle mode, non-inverted mode is the most reasonable. This is because upon increasing the compare voltage, the duty cycle increases. However, you can choose any of them. Regarding toggle mode, I wonder if there is any practical application of it.

So this is it! In the next post, we will learn how to implement it in AVRs!

# AVR Timers – PWM Mode – Part II

This article is in continuation with the previous PWM post. Learn how to program the timers to operate in PWM mode! So let's begin!

Hello folks! Long time no see!

In my previous post, we have discussed the basic concepts of PWM. Let's summarize it first:

- PWM stands for Pulse Width Modulation.
- It can be generated by comparing predetermined waveform with a reference voltage level or by making simple analog circuits.
- Duty Cycle of a PWM waveform is given by the following relation.

$$Duty\ Cycle = \frac{T_{on}}{T_{on} + T_{off}} \times 100\ \%$$

- There are three modes of PWM operation - Fast PWM, Phase Correct PWM and Frequency and Phase Correct PWM
- How to choose timer, operation mode and compare output mode for generating the desired PWM.

So now, without much hassle, let's see how to implement it using the AVR microcontrollers. Before we proceed, I suggest you to go through my previous posts on Timers and PWM.

# Problem Statement

Let us take a problem statement. We need to generate a 50 Hz PWM signal having 45% duty cycle.

### Analysis

Given that

```
Frequency = 50 Hz
```

In other words, the time period, T

```
T = T(on) + T(off) = 1/50 = 0.02 s = 20 ms
```

Also, given that

```
Duty Cycle = 45%
```

Thus, solving according to equation given above, we get

```
T(on)  = 9 ms
T(off) = 11 ms
```

Now, this can be achieved in two ways:

1. Use Timer in CTC Mode
2. Use Timer in PWM Mode

# Methodology – CTC Mode

Okay, so I won't be writing any code here (just the pseudo code). I assume that after reading my previous posts, you are smart enough to write one yourself! We will discuss only the concepts.

Firstly, choose a suitable timer. For this application, we can choose any of the three timers available in ATMEGA32. Choose a suitable prescaler. Then set up the timer and proceed as usual. The catch lies here is that you need to update the compare value of OCRx register everytime. One such way is discussed in the pseudo code given below.

This is **analogous** to the traditional LED flasher, except the fact that the on and off times are different.

### Pseudo Code

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

uint8_t count = 0;              // global counter

// initialize timer, interrupt and variable
void timerX_init()
{
    // set up timerX with suitable prescaler and CTC mode
    // initialize counter
    // initialize compare value
    // enable compare interrupt
    // enable global interrupts
}
```

```
13
14// process the ISR that is fired
15ISR (TIMERx_COMPA_vect)
16{
17    // do whatever you want to do here
18    // say, increment the global counter
19    count++;
20
21    // check for the global counter
22    // if count == odd, delay required = 11 ms
23    // if count == even, delay required = 9 ms
24    // thus, the value of the OCRx should be constantly updated
25    if (count % 2 == 0)
26        OCRx = 9999;      // calculate and substitute appropriate value
27    else
28        OCRx = 10999;     // calculate and substitute appropriate value
29}
30
31int main(void)
32{
33    // initialize the output pin, say PC0
34    DDRC |= (1 << 0);
35
36    // initialize timerX
37    timerX_init();
38
39    // loop forever
40    while(1)
41    {
42        // do nothing
43
44    }
45}
46
47
```

Now this is one method. And it's very inefficient. You can increase its efficiency by writing a better C code (syntax-wise), however the concept remains the same. If you have any other method/concept, you are most welcome to share it here!

Please note that this code not tested yet! So, if any of you is trying it out, do post your results here, I would be happy to see them!

# Methodology – PWM Mode

Okay, so now lets learn about the PWM mode. The PWM Mode in AVR is hardware controlled. This means that everything, by *everything* I mean *"everything"*, is done by the AVR CPU. All you need to do is to initialize and start the timer, and set the duty cycle! Cool, eh?! Let's learn how!

Here, I have used Timer0 of ATMEGA32 for demonstration. You can choose any other other timer or AVR microcontroller as well. Now let's have a look at the registers.

## TCCR0 – Timer/Counter0 Control Register

We have come across this register in my Timer0 tutorial. Here, we will learn how to set appropriate bits to run the timer in PWM mode.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 | TCCR0 |
| Read/Write | W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

TCCR0 Register

We will discuss only those bits which are of interest to us now.

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

| Mode | WGM01 (CTC0) | WGM00 (PWM0) | Timer/Counter Mode of Operation | TOP | Update of OCR0 | TOV0 Flag Set-on |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX |
| 1 | 0 | 1 | PWM, Phase Correct | 0xFF | TOP | BOTTOM |
| 2 | 1 | 0 | CTC | OCR0 | Immediate | MAX |
| 3 | 1 | 1 | Fast PWM | 0xFF | TOP | MAX |

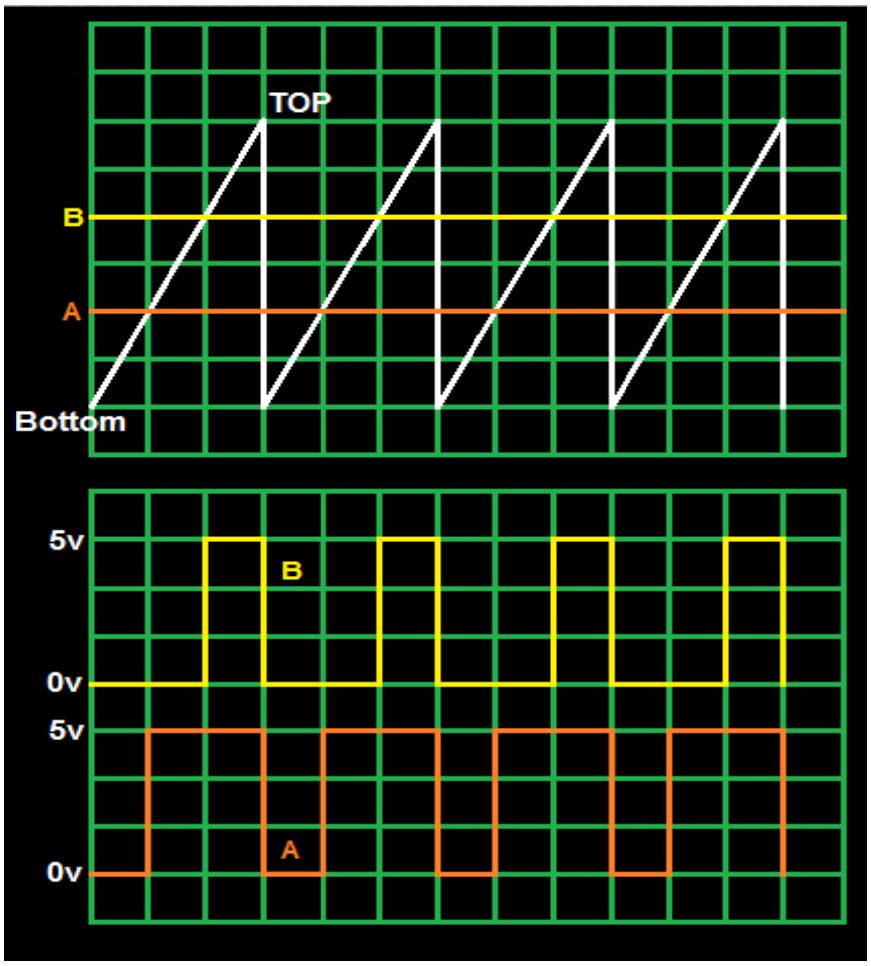
Waveform Generation Mode Bit Description

- **Bit 5,4 – COM01:0 – Compare Match Output Mode** - These bits are set in order to control the behavior of Output Compare pin (OC0, pin 4 in ATMEGA32) in accordance with the WGM01:0 bits. The following look up table determine the operations of OC0 pin for Fast PWM mode.

| COM01 | COM00 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0 disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0 on compare match, set OC0 at TOP |
| 1 | 1 | Set OC0 on compare match, clear OC0 at TOP |

Compare Output Mode, Fast PWM Mode

Now lets have a look at the Fast PWM waveforms. Detailed explanation can be found in my previous tutorial.



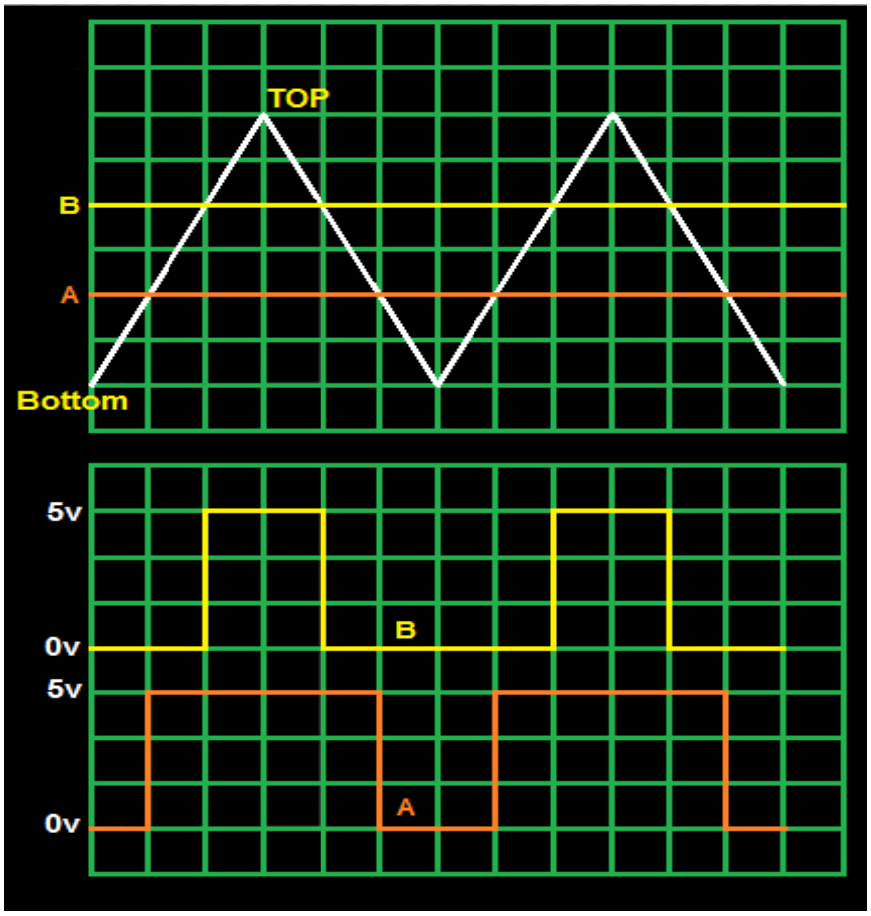

Fast PWM

Now let me remind you that the AVR PWM is fully hardware controlled, which means that even the timer compare operation is done by the AVR CPU. All we need to do is to *tell* the CPU *what* to do once a match occurs. The COM01:0 pins come into play here. We see that by setting it to "10″ or "11″, the output pin OC0 is either set or cleared (in other words, it determines whether the PWM is in inverted mode, or in non-inverted mode).

Similarly for Phase Correct PWM, the look up table and the waveforms go like this.

| COM01 | COM00 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0 disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0 on compare match when up-counting. Set OC0 on compare match when downcounting. |
| 1 | 1 | Set OC0 on compare match when up-counting. Clear OC0 on compare match when downcounting. |

Compare Output Mode, Phase Correct PWM Mode



Phase Correct PWM

Even here, setting COM01:0 to “10″ or “11″ determines the behavior of OC0 pin. As shown in the waveforms, there are two instances – one during up-counting, and other during down-counting. The behavior is clearly described in the look up table.

Please note that OC0 is an output pin. Thus, the effects of WGM and COM won’t come into play unless the DDRx register is set properly. Refer this tutorial for more info.

- **Bit 2:0 – CS02:0 – Clock Select Bits** - These bits are already discussed in Timer0 tutorial.

# OCR0 – Output Compare Register

We have come across even this register in my Timer0 tutorial. We use this register to store the compare value. But when we use Timer0 in PWM mode, the value stored in it acts as the duty cycle (obviously!). In the problem statement, its given that the duty cycle is 45%, which means

```
OCR0 = 45% of 255 = 114.75 = 115
```

And that’s it! Now we are ready to write a code for it!

# Edit: Note

The following code discusses how to create a PWM signal of a desired duty cycle. If you wish to change its frequency, you need to alter the TOP value, which can be done using the ICRx register (which is not supported by 8-bit timers). For 16-bit Timer1, it can be varied using ICR1A. I will discuss about this soon when we discuss about servo control.

# Code

So here goes the code. To learn about I/O port operations in AVR, view this. To know about bit manipulations, view this. To learn how to use AVR Studio 5, view this. To learn how this code is structured, view the previous TIMER0 post.

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3 void pwm_init()
4 {
5     // initialize TCCR0 as per requirement, say as follows
      TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00);
```

```
6
7       // make sure to make OC0 pin (pin PB3 for atmega32) as output pin
8       DDRB |= (1<<PB3);
9   }
10
11  void main()
12  {
13      uint8_t duty;
14      duty = 115;         // duty cycle = 45% of 255 = 114.75 = 115
15
16      // initialize timer in PWM mode
17      pwm_init();
18
19      // run forever
20      while(1)
21      {
22          OCR0 = duty;
23      }
24  }
25
```

# Problem Statement

So now, let's take another problem statement. This one is going to be a more of a practical stuff unlike the previous one!

Let's take the traditional LED flasher where we need to blink an LED at a particular frequency. But hey, wait, didn't we discuss it long back in this post (scroll down towards the end)? Hmm, so let's modify it so as to incorporate PWM. Unlike the traditional LED flasher (where LEDs are either ON or OFF), lets make it glow at the maximum brightness, and then slowly decrease its brightness till it reaches zero, and then again increase its brightness slowly till it becomes maximum.

### Analysis and Code

So how do we do it? Yes, you guessed it right! Decrease the duty cycle slowly from 255 to zero, and then increase it from zero to 255. Depending upon the duty cycle, the voltage applied to the LED varies, and thus the brightness. The following formula gives the relation between voltage and duty cycle.

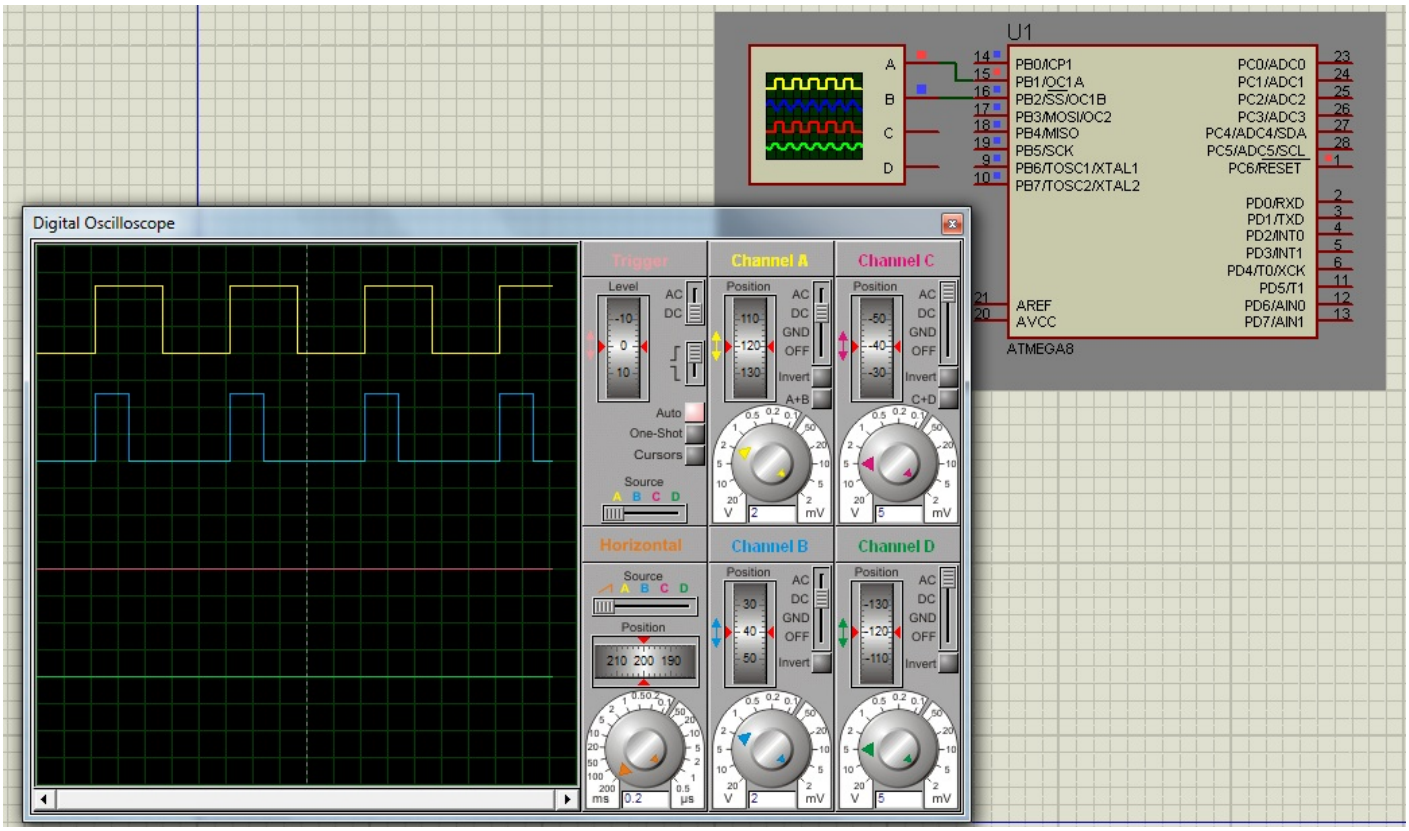
$$V_{out} = \frac{Duty\ Cycle}{255} \times 5\ volts$$

So here goes the code. I won't explain it, you can decode it yourself. To learn about I/O port operations in AVR, view this. To know about bit manipulations, view this. To learn how to use AVR Studio 5, view this. To learn how this code is structured, view the previous TIMER0 post.

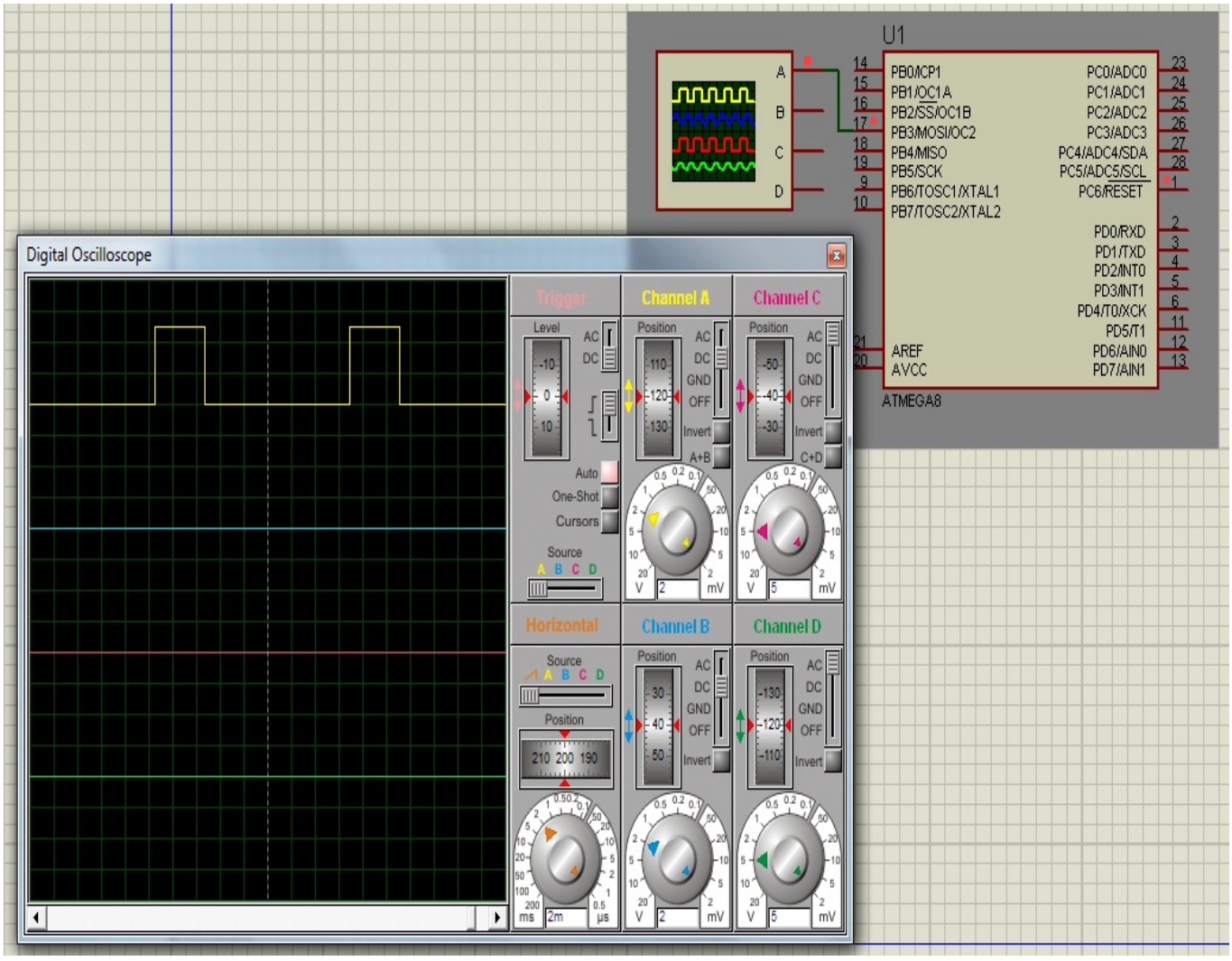
```
// program to change brightness of an LED
// demonstration of PWM

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

// initialize PWM
void pwm_init()
{
    // initialize timer0 in PWM mode
    TCCR0 |= (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00);

    // make sure to make OC0 pin (pin PB3 for atmega32) as output pin
    DDRB |= (1<<PB3);
}

void main()
{
    uint8_t brightness;

    // initialize timer0 in PWM mode
    pwm_init();

    // run forever
    while(1)
    {
        // increasing brightness
        for (brightness = 0; brightness < 255; ++brightness)
        {
            // set the brightness as duty cycle
            OCR0 = brightness;

            // delay so as to make the user "see" the change in brightness
            _delay_ms(10);
        }

        // decreasing brightness
        for (brightness = 255; brightness > 0; --brightness)
```

```
        {
            // set the brightness as duty cycle
            OCR0 = brightness;

            // delay so as to make the user "see" the change in brightness
            _delay_ms(10);
        }

    // repeat this forever
    }
}
```

```
//PWM for 20ms for duty cycle 5-10 % at 1MHz

#include<avr/io.h>
#include<util/delay.h>
//#define F_CPU 1000000UL
void main()
{
    DDRB=255;
    TCCR1B|=(1<<WGM13)|(1<<WGM12)|(1<<CS10)|(1<<CS11); //64 prescalar
    TCCR1A|=(1<<WGM11)|(1<<COM1A1)|(1<<COM1B1); //clear OCR1A on compare, fast pwm mode, A,B -
    Inverting Mode
    TCNT1=0;
    /* Calculation part:
     F_CPU = 1000000
     Prescalar = 64
     Clock_CPU = 1000000/64 = 15625
     Clock Time = 1/15625 = 64us
     Time Delay Recq = 1s
     Timer Count (ICR1) = 1s/64us - 1 = 15624
     Duty Cycle:
               50% of 15624 = (50/100)*15624 = 7812
               25% of 15624 = (25/100)*15624 = 3906
     */
    ICR1=15624;
    OCR1A=7812;
    OCR1B=3906;
    while(1);
}
```



```c
// Program to generate 25% Duty Cycle (PWM) using Timer 2

#include<avr/io.h>

void timer2_init()
{
    TCNT2=0;
    TCCR2=(1<<WGM21)|(1<<WGM20)|(1<<COM21)|(1<<CS22); //Fast PWM, Non Inverted Mode
}
void main()
{
    DDRB=(1<<DDB3);
    timer2_init();
    OCR2=64; //Duty Cycle: 25%, Max(Top) = 255, OCR2=25% of 255 = 25/100 * 255 = 63.75 =~64
    while(1);
}
```



# Serial Communication – Introduction

Hey folks! Guess what? It's now time for one of the most desired tutorials on maxEmbedded – the Serial Communication series! In these series, we will discuss the basic concepts of serial communication; the loopback test, the USART/UART of AVR and then we will proceed towards implementing the SPI and I2C in AVR.

This post will cover the basics of serial communication and will be mostly a theoretical topic. We will do some practical stuff from next tutorial onwards. Lets have a glance at the contents.

# Contents



# What is Communication?

Before we move on to serial communication, lets discuss a bit about communication in general. In simple terms, communication is an exchange of ideas between two individuals. Ideas can be anything and in any form – they could be written/spoken words, in form of media like audio/video, or if you like sci-fi, then it can also in form of telepathy! 

But what does communication between two microcontrollers mean? Its simple! An exchange of data (bits)! There are many protocols for communication (which would be discussed later) but all of them are based on either serial communication or parallel communication.

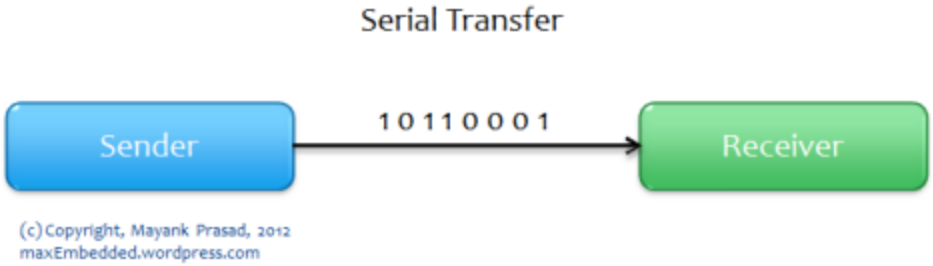
# Why do we need Communication?

Lets take an example. As kids, we all must have played with those remote controlled toy cars and airplanes. It was pretty fun and fascinating at that time. I am sure that most of us at that time didn't try to figure out how it was possible! How could the remote control device in your hand control the car or the aeroplane? Well, of course, the device in your hand sends some data, which is received by the car/aeroplane. There is a microcontroller onboard the toy, which interprets the signals and acts accordingly. Correct! So far so good, but *now* it doesn't end here. As grown ups, there are a few more questions which should arise! Like how does the device send the signal? From where is the signal being sent? What is actually being sent? Who receives it? How is it processed?

Lets take another example. This one's a more common example. You have a file in your mobile and you would like to share it with your friend who is sitting next to you? How would you do it – Bluetooth, IR, NFC, LAN or email? Mostly people would use Bluetooth. IR is obsolete, NFC is still in developmental phase and isn't available in most devices, LAN needs a WiFi/LAN network whereas email requires an active Internet connection. The same questions can be put forth here as well – how is it send, from where is it sent and to where, what is being sent and how is it processed?!

Well, this is why communication is required! And to answer all those questions, several communication protocols have been developed! Now lets discuss a little about serial and parallel communication.

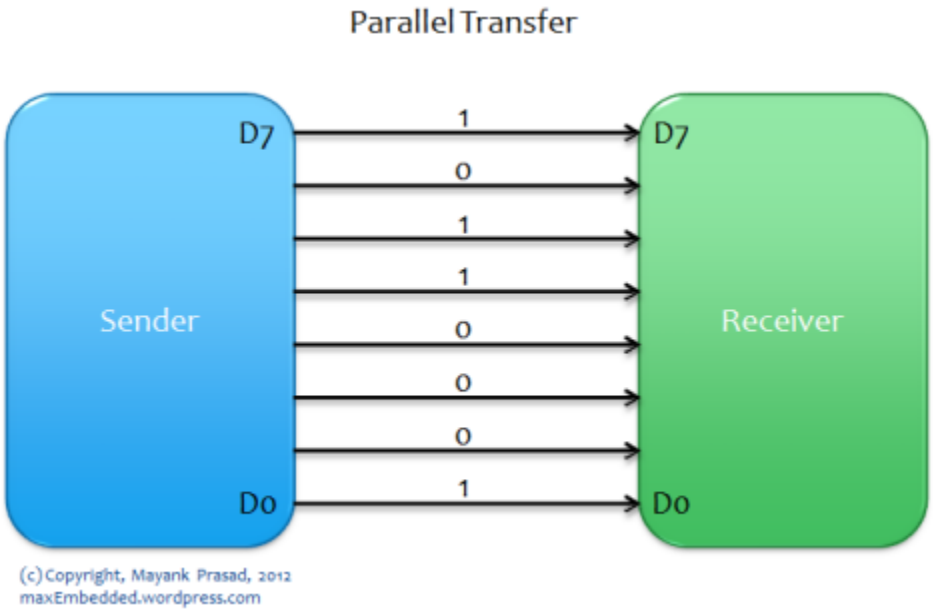
# Serial Communication




Serial Transfer

In Telecommunication and Computer Science, serial communication is the process of sending/receiving data in one bit at a time. It is like you are firing bullets from a *machine gun* to a target… that's one bullet at a time! 😉

# Parallel Communication



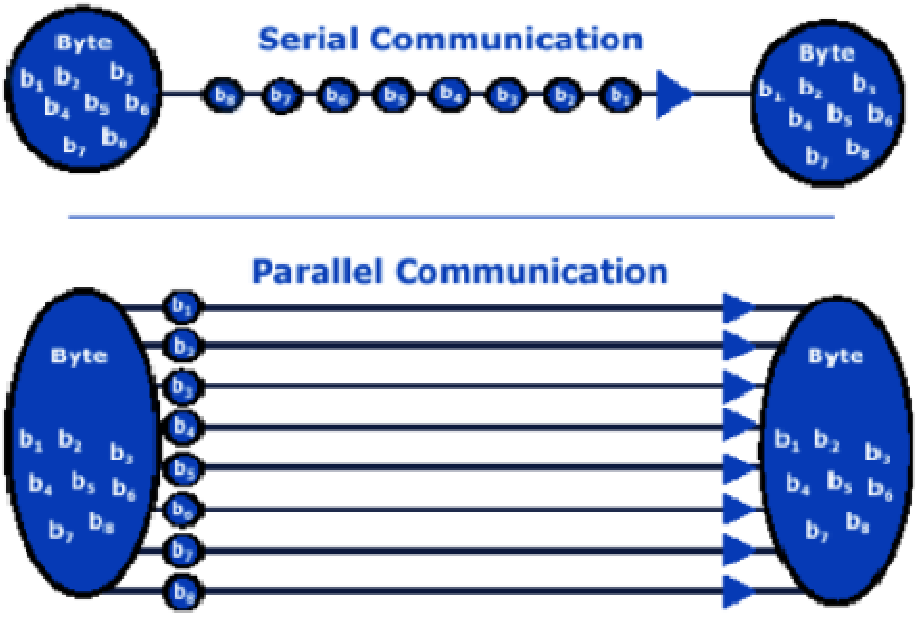

Parallel Transfer

Parallel communication is the process of sending/receiving multiple data bits at a time through parallel channels. It is like you are firing using a *shotgun* to a target – where multiple bullets are fired from the same gun at a time! 😉

# Serial vs Parallel Communication

Now lets have a quick look at the differences between the two types of communications.

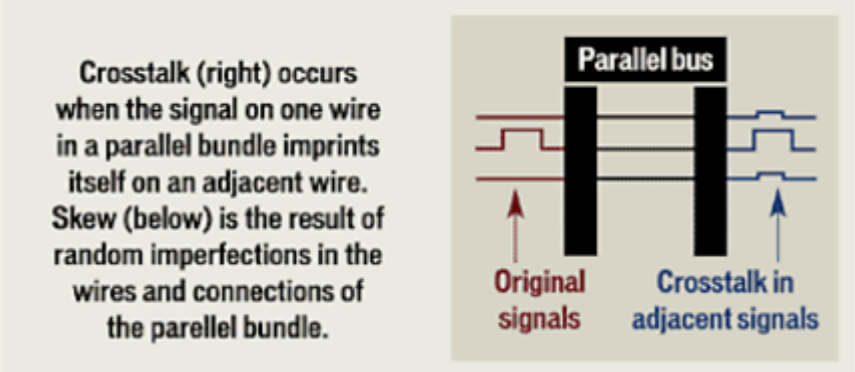
| **Serial Communication** | **Parallel Communication** |
|---|---|
| 1. One data bit is transceived at a time | 1. Multiple data bits are transceived at a time |
| 2. Slower | 2. Faster |
| 3. Less number of cables required to transmit data | 3. Higher number of cables required |

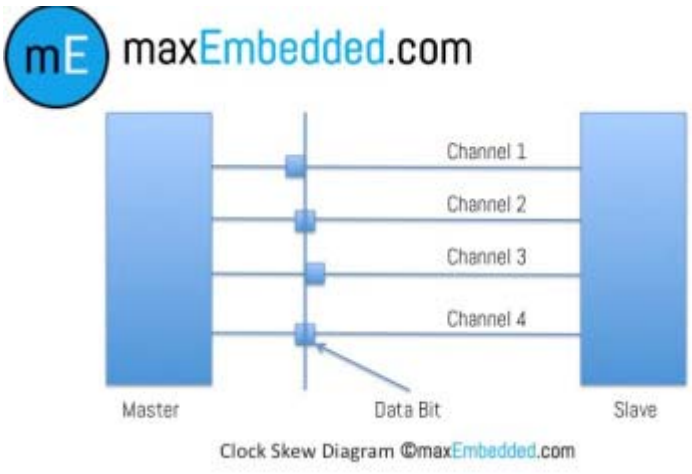



Serial vs Parallel

So these were the basic differences between serial and parallel communication. From the above differences, one would obviously think that parallel communication is far better than serial communication. But wait, these are just the *basic* differences. Before we proceed further, we need to be acquainted with a few terminologies:

1. **Bit Rate:** It is the number of bits that are transmitted (sent/received) per unit time.
2. **Clock Skew:** In a parallel circuit, clock skew is the time difference in the arrival of two sequentially adjacent registers. To explain it further, let us take the machine gun example again. When, say around 5 people are firing at the same time, there is bound to be a time difference in the arrival of the bullet from the first shooter and that from the second shooter and so on. This time difference is what we call clock skew. This is better illustrated in the picture below: There is a time lag in the data bits through different channels of the same bus. Clock skew is inevitable due to differences in physical conditions of the channels, like temperature, resistance, path length, etc
3. **Crosstalk:** Phenomenon by which a signal transmitted on one channel of a transmission bus creates an undesired effect in another channel. Undesired capacitive, inductive, or conductive coupling is usually what is called crosstalk, from one circuit, part of a circuit, or channel, to another. It can be seen from the following diagram that clock skew and crosstalk are inevitable.




# Major Factors Limiting Parallel Communication

Before the development of high-speed serial technologies, the choice of parallel links over serial links was driven by these factors:

1. **Speed**: Superficially, the speed of a parallel link is equal to *bit rate*number of channels*. In practice, clock skew reduces the speed of every link to the slowest of all of the links.
2. **Cable length**: Crosstalk creates interference between the parallel lines, and the effect only magnifies with the length of the communication link. This limits the length of the communication cable that can be used.

These two are the major factors, which limit the use of parallel communication.

# Advantages of Serial over Parallel

Although a serial link may seem inferior to a parallel one, since it can transmit less data per clock cycle, it is often the case that serial links can be clocked considerably faster than parallel

links in order to achieve a higher data rate. A number of factors allow serial to be clocked at a higher rate:

- Clock skew between different channels is not an issue (for un-clocked asynchronous serial communication links).
- A serial connection requires fewer interconnecting cables (e.g. wires/fibers) and hence occupies less space. The extra space allows for better isolation of the channel from its surroundings.
- Crosstalk is not a much significant issue, because there are fewer conductors in proximity.

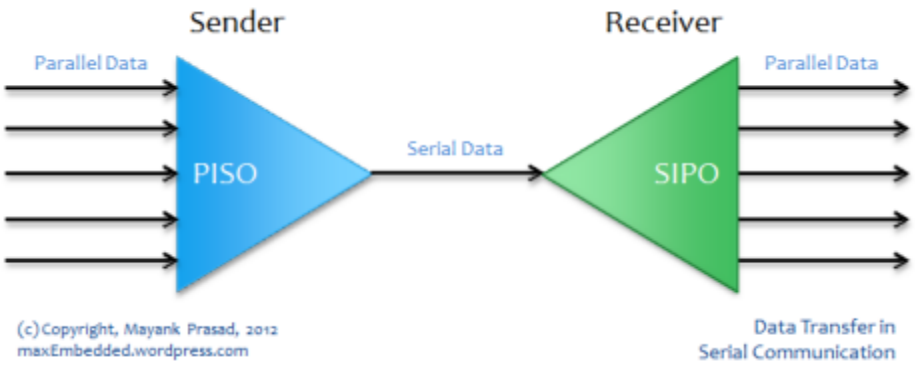
In many cases, serial is a better option because it is cheaper to implement. Many ICs have serial interfaces, as opposed to parallel ones, so that they have fewer pins and are therefore less expensive. It is because of these factors, serial communication is preferred over parallel communication.

# How is Data sent Serially?

Since we already know what are registers and data bits, we would now be talking in these terms only. If not, I would recommend you to first take a detour and go through the introduction of this post by Mayank.

When a particular data set is in the microcontroller, it is in parallel form, and any bit can be accessed irrespective of its bit number. When this data set is transferred into the output buffer to be transmitted, it is still in parallel form. This output buffer converts this data into Serial data (**PISO**) (**Parallel In Serial Out**), MSB (Most Significant Bit) first or LSB (Least Significant Bit) first as according to the protocol. Now this data is *transmitted in Serial mode.*

When this data is received by another microcontroller in its receiver buffer, the receiver buffer converts it back into parallel data (**SIPO**) (**Serial In Parallel Out**) for further processing. The following diagram should make it clear.




Data Transfer in Serial Communication

This is how serial communication works! But it is not as simple as it looks. There is a catch in it, which we will discuss little later in the same post. For now, lets discuss about two modes of serial data transfer – synchronous and asynchronous.

# Serial Transmission Modes

Serial data can be transferred in two modes – asynchronous and synchronous.

## Asynchronous Data Transfer

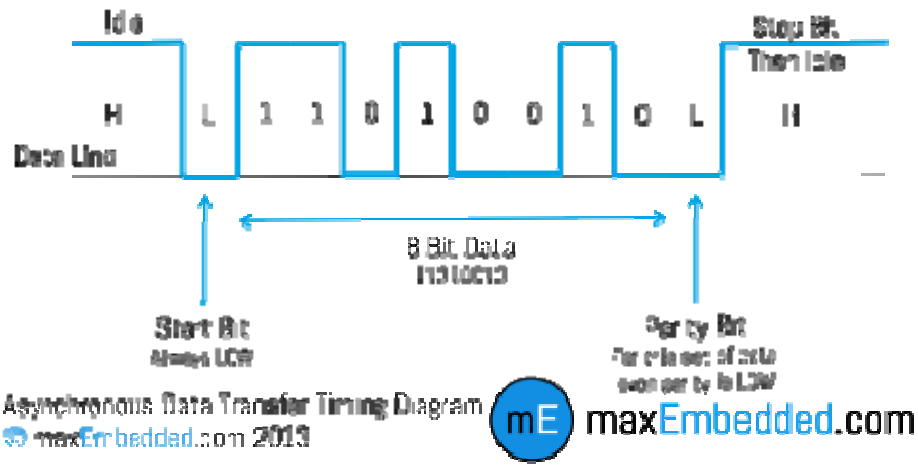
Data Transfer is called Asynchronous when data bits are not "synchronized" with a clock line, i.e. there is no clock line at all!

Lets take an analogy. Imagine you are playing a game with your friend where you have to throw colored balls (let's say we have only two colors – red (R) and yellow (Y)). Lets assume you have unlimited number of balls. You have to throw a combination of these colored balls to your friend. So you start throwing the balls. You throw R, then R, then Y, then R again and so on. So you start your sequence RRYR… and then you end your round and start another round. How will your buddy on the other side know that you have finished sending him first round of balls and that you are already sending him the second round of balls?? He/she will be completely lost! How nice it would be if you both sit together and fix a protocol that each round consists of 8 balls! After every 8 balls, you will throw two R balls to ensure that your friend has caught up with you, and then you again start your second round of 8 balls. This is what we call *asynchronous data transfer*.

Asynchronous data transfer has a protocol, which is *usually* as follows:

- The first bit is always the START bit (which signifies the start of communication on the serial line), followed by DATA bits (usually 8-bits), followed by a STOP bit (which signals the end of data packet). There may be a Parity bit just before the STOP bit. The Parity bit was earlier used for error checking, but is seldom used these days.
- The START bit is always low (0) while the STOP bit is always high (1).

The following diagram explains it.




Asynchronous Data Transfer Timing Diagram

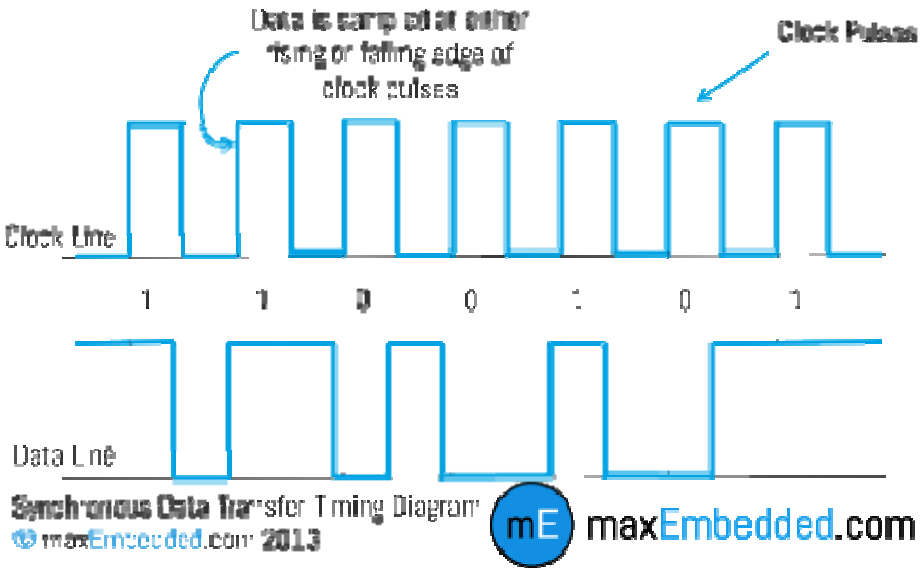
# Synchronous Data Transfer

Synchronous data transfer is when the data bits are “synchronized” with a clock pulse.

We will take the same analogy as before. You are still playing the throw-ball game, but this time, you have set a timer in your watch such that it beeps every minute. You will *not* throw a ball unless you hear a beep from your watch. As soon as you hear a beep from your watch, you and your friend, both know that you are going to throw a ball to her. Both of you can keep a track of time using this; say you start a new round after every 8 beeps. Isn’t it a much better approach? This approach is what we call *synchronous data transfer*.

The concept for synchronous data transfer is simple, and as follows:

- The basic principle is that data bit sampling (or in other words, say, ‘recording’) is done with respect to clock pluses, as you can see in the timing diagrams.
- Since data is sampled depending upon clock pulses, and since the clock sources are very reliable, so there is much less error in synchronous as compared to asynchronous.

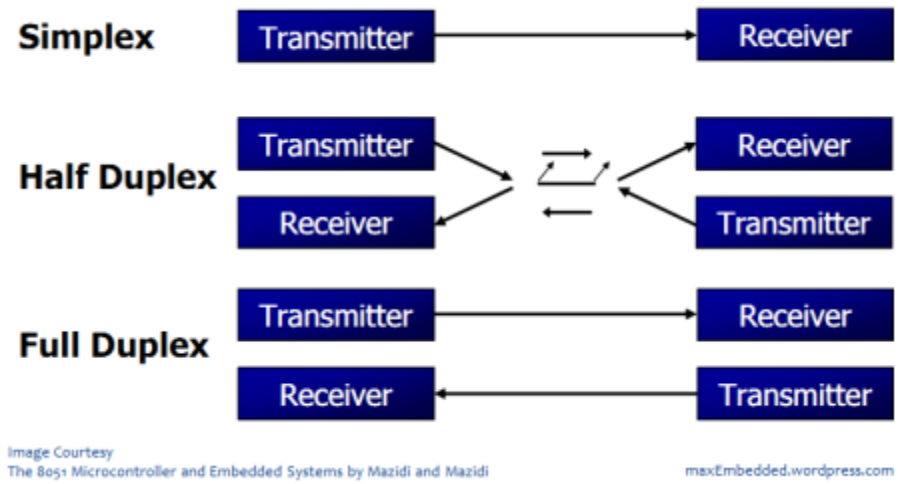



Synchronous Data Transfer Timing Diagram

# Serial Communication Terminologies

Now its time to learn about some new words, which we will use frequently in the next few posts. There are many terminologies, or 'keywords' associated with serial communication. We will discuss all of them one by one:

1. **MSB/LSB:** this stands for Most Significant Bit (or Least Significant Bit). You can refer to Mayank's this post for more information on MSB and LSB. Since data is transferred bit-by-bit in serial communication, one needs to know which bit is sent out first: MSB or LSB.
2. **Simplex Communication:** In this mode of serial communication, data can only be transferred from transmitter to receiver and not vice versa.
3. **Half Duplex Communication:** this means that data transmission can occur in only one direction at a time, i.e. either from master to slave, or slave to master, but not both.
4. **Full Duplex Communication:** full duplex communication means that data can be transmitted from the master to the slave, and from slave to the master as the *same time!*




Types of Transmission

5. **Baud Rate:** according to Wikipedia, baud is synonymous to symbols per second or pulses per second. It is the unit of symbol rate, also known as *baud* or *modulation rate*. However, though technically incorrect, in the case of modem manufacturers baud commonly refers to bits per second.

### Importance of Baud Rate

For two microcontrollers to communicate serially they should have the *same* baud rate, else serial communication won't work. This is because when you set a baud rate, you direct the microcontroller to transmit/receive the data at that particular rate. So if you set different baud rates, then the receiver might miss out the bits the transmitter is sending (because it is configured to receive data and process it with a different speed!)

Different baud rates are available for use. The most common ones are 2400, 4800, 9600, 19200, 38400 etc. You cannot choose any arbitrary baud rate, there are some fixed values which you must use like 2400, 4800, etc. Please note that the unit of baud rate is bps (bits per second).

# The Catch in Serial Communication

Now it's all clear to you. You have data. You decide how to send your data (synchronous/asynchronous). You send your data by following proper protocols. The transmitter converts your parallel data to serial, sends it across the channel, then the receiver converts your serial data to parallel. Bingo! But that's not sufficient for a proper serial communication. There are two things which still needs to be taken care of:

1. **Baud Rate:** Unless the baud rate of both the transmitter and receiver are the same, serial communication cannot work. The reason is specified in the previous section.
2. **Address:** If you are trying to send multiple data together over the same channel and/or you are sharing the same channel space with other users sending their own data, then you need to take care to properly address your data. We won't discuss about it in this post, but we will surely discuss about it in one of our upcoming posts.

If you take care of these two factors, your serial communication will be established perfectly and your data will go through properly. These are the two main reasons for unsuccessful serial link.

# UART and USART

UART stands for Universal Asynchronous Receiver Transmitter, whereas USART stands for Universal Synchronous Asynchronous Receiver Transmitter. They are basically just a piece of computer hardware that converts parallel data into serial data. The only difference between them is that UART supports only asynchronous mode, whereas USART supports both asynchronous and synchronous modes. Unlike Ethernet, Firewire etc., there is no specific port for UART/USART. They are commonly used in conjugation with protocols like RS-232, RS-434 etc. (we have specific ports for these two!).

In *synchronous* transmission, the clock data is recovered separately from the data stream and no start/stop bits are used. This improves the efficiency of transmission on suitable channels since more of the bits sent are usable data and not character framing.

The USART has the following components:

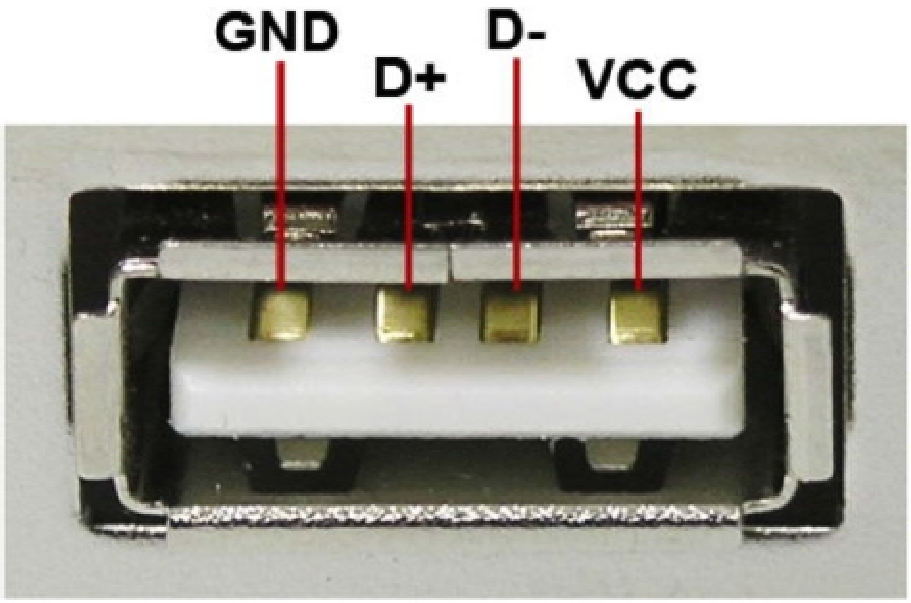
- A clock generator, usually a multiple of the bit rate to allow sampling in the middle of a bit period
- Input and output shift registers
- Transmit/receive control
- Read/write control logic
- Transmit/receive buffers (optional)
- Parallel data bus buffer (optional)
- First-in, first-out (FIFO) buffer memory (optional)

# Serial Communication Protocols

A variety of communication protocols have been developed based on serial communication in the past few decades. Some of them are:

1. **SPI – Serial Peripheral Interface:** It is a three-wire based communication system. One wire each for Master to slave and Vice-versa, and one for clock pulses. There is an additional SS (Slave Select) line, which is mostly used when we want to send/receive data between multiple ICs.
2. **I2C – Inter-Integrated Circuit:** Pronounced eye-two-see or eye-square-see, this is an advanced form of USART. The transmission speeds can be as high as a whopping 400KHz. The I2C bus has two wires – one for clock, and the other is the data line, which is bi-directional – this being the reason it is also sometimes (not always – there are a few conditions) called **Two Wire Interface (TWI)**. It is a pretty new and revolutionary technology invented by Philips.
3. **FireWire** – Developed by Apple, they are high-speed buses capable of audio/video transmission. The bus contains a number of wires depending upon the port, which can be either a 4-pin one, or a 6-pin one, or an 8-pin one.


4. **Ethernet:** Used mostly in LAN connections, the bus consists of 8 lines, or 4 Tx/Rx pairs.
5. **Universal serial bus (USB):** This is the most popular of all. Is used for virtually all type of connections. The bus has 4 lines: $V_{CC}$, Ground, Data+, and Data-.




USB Pins

6. **RS-232 – Recommended Standard 232:** The RS-232 is typically connected using a DB9 connector, which has 9 pins, out of which 5 are input, 3 are output, and one is Ground. You can still find this so-called “Serial” port in some old PCs. In our upcoming posts, we will discuss mainly about RS232 and USART of AVR microcontrollers.

# Serial Communication – RS232 Basics

In the previous post, we discussed about the basics of serial communication. In this post, we will learn about the RS-232 protocol of serial communication. This is the protocol you will be using the most when involving microcontrollers like AVR. As we proceed ahead in this post, we will deal with the concept of level conversion and towards the end, we have something interesting and practical for you – the loopback test!

# Contents



# RS-232 Basics

RS-232 (Recommended Standard – 232) is a standard interface approved by the Electronic Industries Association (EIA) for connecting serial devices. In other words, RS-232 is a long established standard that describes the physical interface and protocol for relatively low-speed serial data communication between computers and related devices. RS-232 is the interface that your computer uses to "talk" to and exchange data with your modem and other serial devices. The serial ports on most computers use a subset of the RS-232C standard. RS-232 protocol is mostly used over the DB9 port (commonly known as *serial port*), however earlier it was used over the DB25 port (also known as *parallel port*). We will have a look at both of them here.

## RS-232 over DB-9

The pin configuration of DB-9 port is as follows. Yes, it looks exactly like (in fact it is) the serial port you would find in older computers.

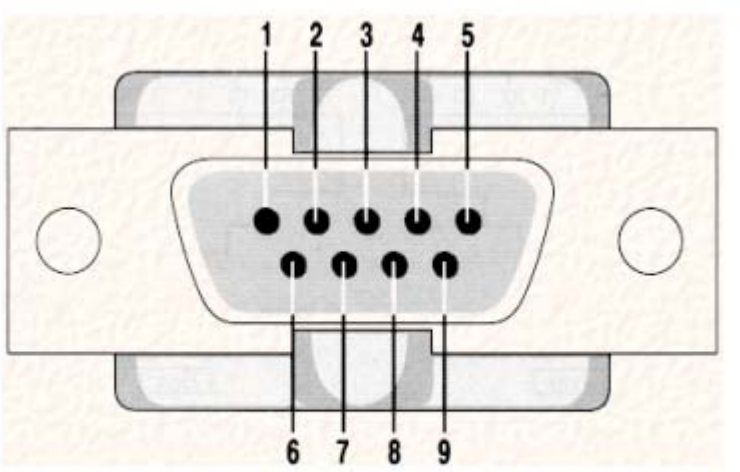



Image Courtesy
The 8051 Microcontroller and Embedded Systems by Mazidi and Mazidi

DB9 Connector

RS232 DB9 Connector

| Pin | Description |
|---|---|
| 1 | Data carrier detect (-DCD) |
| 2 | Received data (RxD) |
| 3 | Transmitted data (TxD) |
| 4 | Data terminal ready (DTR) |
| 5 | Signal ground (GND) |
| 6 | Data set ready (-DSR) |
| 7 | Request to send (-RTS) |
| 8 | Clear to send (-CTS) |
| 9 | Ring indicator (RI) |

Courtesy
The 8051 Microcontroller and Embedded Systems
by Mazidi and Mazidi

DB9 Connector Pins

DB9 Connector Pins

The pin description for the RS-232 pins is as follows:

- **DTR** (data terminal ready): When terminal is turned on, it sends out signal DTR to indicate that it is ready for communication.
- **DSR** (data set ready): When DCE is turned on and has gone through the self-test, it assert DSR to indicate that it is ready to communicate.
- **RTS** (request to send): When the DTE device has byte to transmit, it assert RTS to signal the modem that it has a byte of data to transmit.
- **CTS** (clear to send): When the modem has room for storing the data it is to receive, it sends out signal CTS to DTE to indicate that it can receive the data now.
- **DCD** (data carrier detect): The modem asserts signal DCD to inform the DTE that a valid carrier has been detected and that contact between it and the other modem is established.
- **RI** (ring indicator): An output from the modem and an input to a PC indicates that the telephone is ringing. It goes on and off in synchronous with the ringing sound.
- **RxD** (Received data): The RxD pin is the Data Receive pin. This is the pin where the receiver receives data.
- **TxD** (Transmitted data): The TxD pin is the Data Transmit pin. This is the pin through which data is transmitted to the receiver.
- **GND:** Ground pin.

# RS-232 over DB-25

The pin configuration of DB-25 port is as follows. Yes, it looks exactly like (in fact it is) the parallel (printer) port you would find in even older computers!

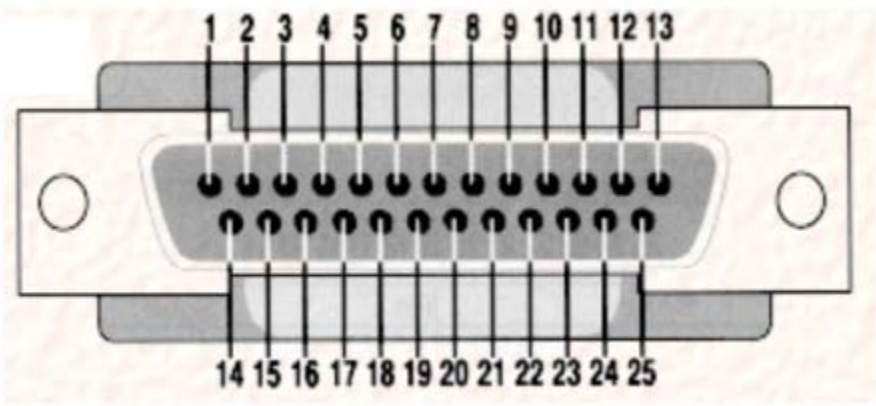



Image Courtesy
The 8051 Microcontroller and Embedded Systems by Mazidi and Mazidi

DB25 Connector

RS232 DB25 Connector

| Pin | Description | Pin | Description |
|---|---|---|---|
| 1 | Protective ground | 14 | Secondary transmitted data |
| 2 | Transmitted data (TxD) | 15 | Transmitted signal element timing |
| 3 | Received data (RxD) | 16 | Secondary receive data |
| 4 | Request to send (-RTS) | 17 | Receive signal element timing |
| 5 | Clear to send (-CTS) | 18 | Unassigned |
| 6 | Data set ready (-DSR) | 19 | Secondary receive data |
| 7 | Signal ground (GND) | 20 | Data terminal ready (-DTR) |
| 8 | Data carrier detect (-DCD) | 21 | Signal quality detector |
| 9/10 | Reserved for data testing | 22 | Ring indicator (RI) |
| 11 | Unassigned | 23 | Data signal rate select |
| 12 | Secondary data carrier detect | 24 | Transmit signal element timing |
| 13 | Secondary clear to send | 25 | Unassigned |

Courtesy
The 8051 Microcontroller and Embedded Systems by Mazidi and Mazidi

DB25 Connector Pins

DB25 Connector Pins

As we can see, most of the pins are similar to that of a DB9 port. If you notice, we see that in DB25 connector there are **two** TxD and RxD pairs of pins. Now what does this mean? In simple words, it means that communication through the DB-25 Connector takes place through two channels, **hence DB-25 is a parallel port for RS-232, whereas the DB-9 is a serial port!**

**NOTE: Another important thing to note is that the simplest way to in which a microcontroller can communicate to a PC is through RxD, TxD, and Ground Pins.** And this is what we will be doing here and hence forth in upcoming posts. The other pins are not of much

use to us, for now. Now this was something about RS-232. Our next topic is level conversion. Btw, have you heard of TTL? Sounds familiar, but what is TTL? Lets read on!

# Logic Level Families

By 'Logic Level' one means the range of voltage over which a high bit (1) and a low bit (0) is accepted in a particular IC, gate, etc. Various logic levels have been standardized, out of which the most popular ones are:

### 1. TTL

TTL stands for Transistor-Transistor Logic. These days TTL is the most widely used logic. TTL is mostly used in ICs and gates, like 74xx logic gates. A major drawback of the TTL logic is that most of the devices working on the TTL Logic consume a lot of current, even individual gates may draw up to 3-4 mA. In TTL Logic, a HIGH (or 1) is +5 volts, whereas a LOW (or 0) is 0 volts. But since attaining exact +5 volts and 0 volt is practically not possible every time, various IC manufactures define TTL logic level range differently, but the usual accepted range for a HIGH is within +3.5 ~ +5.0 volts, and the range for a LOW is 0 ~ +0.8 volts.

### 2. LVTTL

LVTTL stands for Low Voltage Transistor-Transistor Logic. LVTTL is increasingly becoming popular these days, because of the nominal HIGH voltages, and hence lesser power consumption. By lowering the power supply from 5v to 3.3v, switching power reduces by almost 60%! There are several transistors and gates, which work on LVTTL logic. Atmel's Atmega microcontrollers are designed to work on both, LVTTL and TTL, depending upon the $V_{CC}$ supplied to the IC. In LVTTL Logic, a LOW is defined for voltages 0V ~ 1.2V, and High for voltages 2.2V ~ 3.3V, making 1.2V~2.2V undefined.

### 3. RS-232

RS232 is also one of the most popular logic. Though now quite old, it is still in use. In RS-232 logic, a HIGH (1) is represented within -3V ~ -25 V, whereas a LOW (0) is in between +3V ~ +25 V, making -3V to +3V undefined. Weird isn't it? But that's how it is defined! Apart from these, there are many other logic families like ECL, RTL, CMOS, LVCMOS, etc. At present, we are not much concerned about them. You can refer to this article to know more about them.

# Level Conversion – TTL/RS232

So what is (logic) level conversion?? To interconnect any two logic level families, their respective HIGHs and LOWs must be same else they wouldn't work. For example, when we want to interconnect two devices, one of which works over TTL and the other over RS232, we need to convert the HIGH of TTL (which is 3.3v~5v) into the HIGH of RS232 (which is -3v ~ -25v) and similarly, the LOW of TTL (0v~0.8v) into the low of RS232 (which is +3v ~ +25v). So

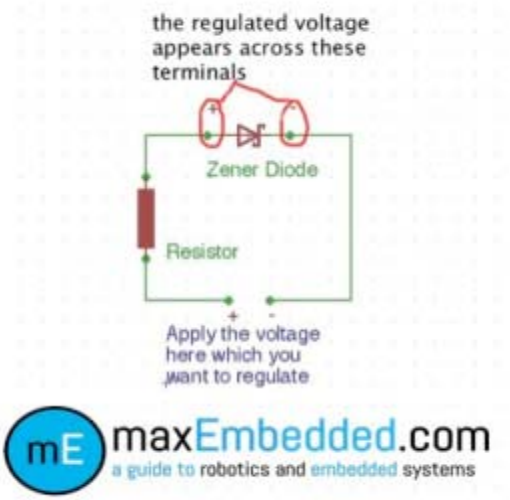
you see, here lies the problem! If we do not convert the logic levels (in this case) then the LOW signal of TTL would be interpreted as a HIGH in RS232, making all the data transfer go wrong!!

# The Solution

One solution is to use additional pull-up resistors, or to use Zener diodes. A better solution is the use of ICs that directly converts logic levels. Luckily logic level conversion is quite simple these days with the use of ICs like MAX232 and CP2012! We would talk about all these solutions one by one.

# Zener Diodes

Zener diodes are widely used to regulate voltage between two points. When Zener diode is used in reverse bias in series with a suitable resistor, and a voltage > breakdown voltage is applied across the terminals of the Zener-resistor pair, then a voltage V = Zener Voltage appears across the terminals of the Zener diode, while the rest of the voltage appears across the terminals of the resistor. The simple circuit below shows how to use Zener diodes to convert logic levels from TTL to LVTTL (Note that this circuit is only applicable for high to low logic level conversions):

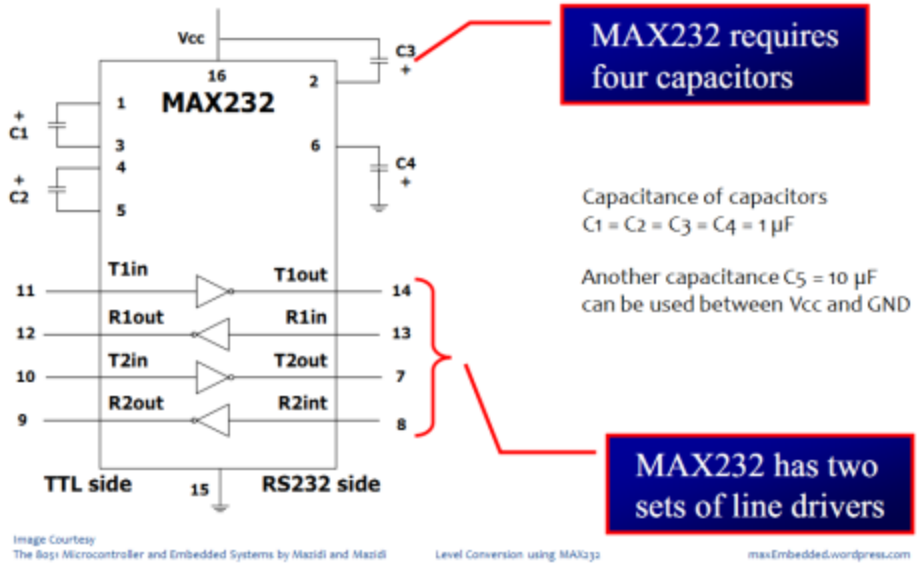



Zener Diode Circuit

Bidirectional Logic level converters are easily available in the market. Some of the websites selling them are: Adafruit, Freetronics, Sparkfun, Embedded Market etc.

# IC MAX232

MAX232 ICs were invented by *Maxim.* These IC packages are used to convert TTL/CMOS logics to RS232 logic directly! All we need are some passive components, and we are done! Below is the circuit diagram of the MAX232 IC.



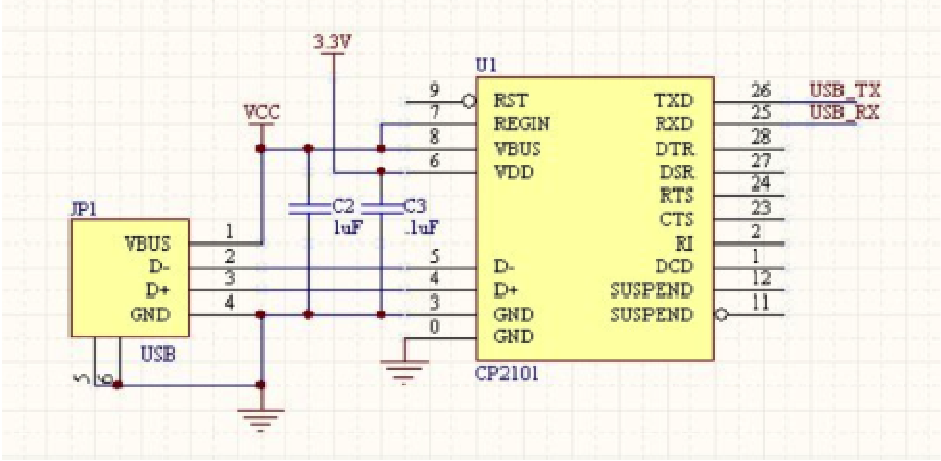

Level Conversion using MAX232

MAX232 is used to convert TTL to RS232, and vice-versa as shown in the above circuit diagram. But these days, USB is the most used protocol! Everything runs on USB – be it printer, scanner, displays or anything! But how to convert USB to UART? One way is USB → TTL → UART. The other way is to use USB-UART bridges which directly convert USB → UART. They are widely and easily available these days. Here are some of the websites: Robokits, eXtreme Electronics, Sparkfun, Adafruit etc.

All these devices work on CP210x based USB-UART conversions.

# IC CP210x

CP210x is the series of ICs made by Silicon Labs. These are used to directly convert USB to UART. Below is the circuit schematic of CP2101:




CP2101 Schematics

Though these ICs are not available in DIP Packages, so it is always advisable to buy any one of the modules listed above.

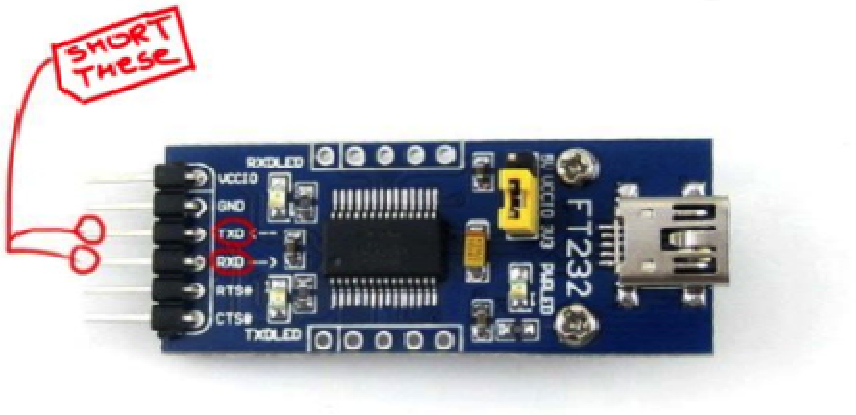
This is how a CP210x based USB-UART Bridge looks like

The drivers of CP210x can be found here**.** They work with the Windows platform. Drivers for Mac and Linux are also available. We will discuss a little later in the same post as to how to install these drivers and work with them.

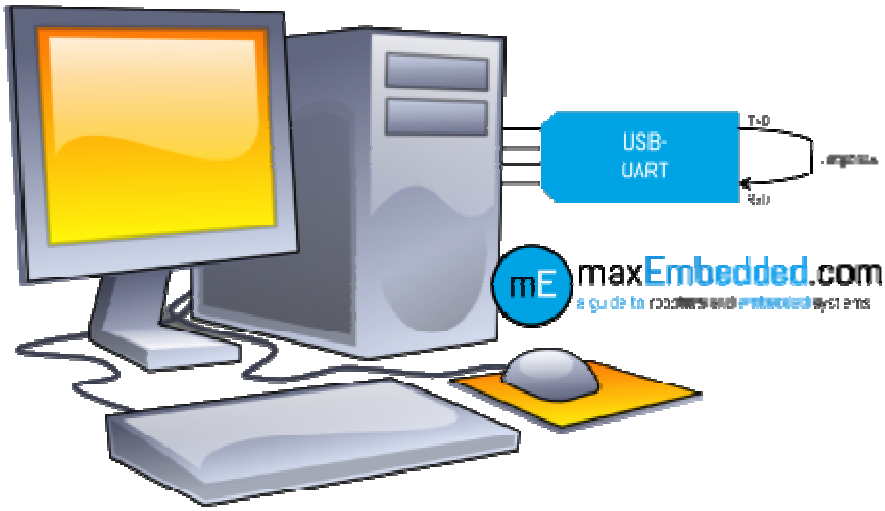
This was all about what is logic level conversion, why we need it, and how to do it. Now its enough of theoretical topics, lets have something practical stuff to do now! Next, we deal how to use the serial communication devices with your PC. So get your PC and USB-UART bridge ready! That's all you need.

# Loopback Test

A Loopback Test in serial communication is a test in which we check whether the Rx and Tx are working properly or not. How we do it? Its simple! Just short the Tx and Rx pins of the USB-UART Bridge and connect it to your PC! So it's like transmitting data from your PC and receiving the same through the same port! Have a look at the following block diagram.




Loopback Test Setup for USB-UART Bridges




Loopback Test Block Diagram

# The USART of the AVR

# Contents



# UART and USART

The UART and USART have already been discussed here. Anyways, lets have a quick recap.

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 Asynchronous Receiver/Transmitter. This is of the synchronous type, i.e. the data bits are synchronized with the clock pulses.

If you refer to the USART section in the datasheet of any AVR microcontroller, you will find several features listed there. 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?

Before we continue, please note that the 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. We will discuss in detail later.

1. 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.
2. Set the number of data bits, which needs to be sent.
3. 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.
4. 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 (ATMega8 Pin 2; ATMega16/32 Pin 14)
2. TxD: USART Transmit Pin (ATMega8 Pin 3; ATMega16/32 Pin 15)
3. XCK: USART Clock Pin (ATMega8 Pin 6; ATMega16/32 Pin 1)

# 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 | |

UBRR Register (Click to Enlarge)

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 ($F_{OSC}$), 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 (= $F_{OSC}/(UBRR+1)$). 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 | $BAUD = \frac{f_{OSC}}{2(UBRR+1)}$ | $UBRR = \frac{f_{OSC}}{2BAUD} - 1$ |

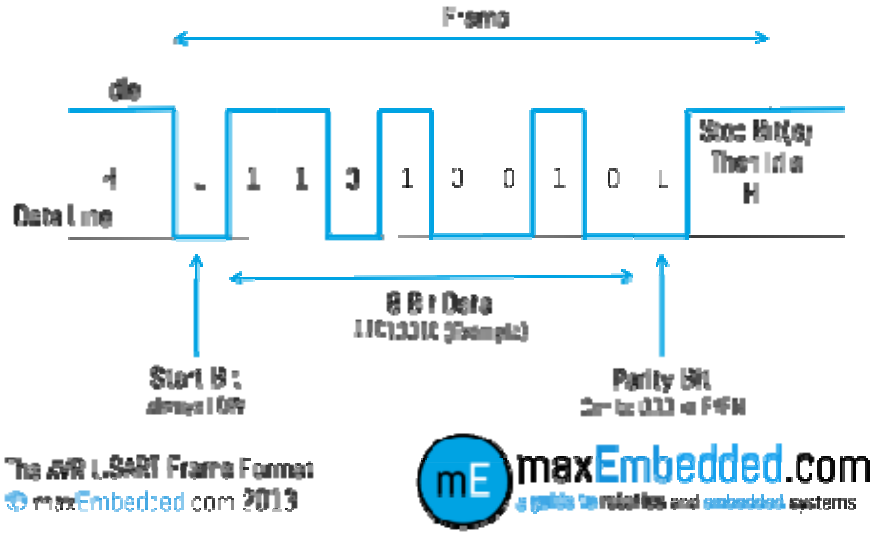
Baud Rate Calculation (Click to Enlarge)

1. BAUD = Baud Rate in Bits/Second (bps) (Always remember, Bps = Bytes/Second, whereas bps = Bits/Second)
2. $F_{OSC}$ = 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. If you have gone through the loopback test discussed in the previous tutorial, you will notice that we have chosen options such as 8 bits data, 1 stop bit, no parity, etc. 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!

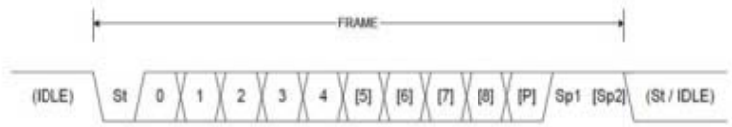



Frame Format (Click to Enlarge)

## Order of Bits

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

A frame starts with the start bit followed by the least significant data bit. Then the next data bits, up to a total of nine, are succeeding, ending with the most significant bit. If enabled, the parity bit is inserted after the data bits, before the stop bits. When a complete frame is transmitted, a new frame can directly follow it, or the communication line can be set to an idle (high) state. Here is the frame format as mentioned in the AVR datasheet-




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

Frame Format (Click to Enlarge)

**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, feel free to comment below! Let me see how many of you can spot it! And I'm not kidding, there *is* a mistake! 🙂

## 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

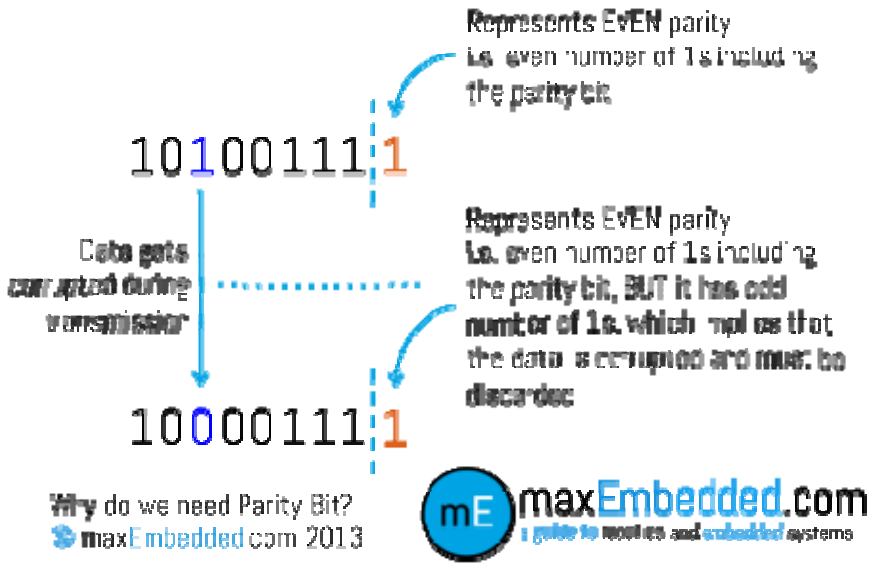
In the above example, we saw that the number has an odd parity. In case of even parity, the parity bit is set to 1, if the number of ones in a given set of bits (not including the parity bit) is odd, making the number of ones in the entire set of bits (including the parity bit) even. If the number of 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 | 0000000**0** | 0000000**1** |
| 1010001 | 3 | 1010001**1** | 1010001**0** |
| 1101001 | 4 | 1101001**0** | 1101001**1** |
| 1111111 | 7 | 1111111**1** | 1111111**0** |

### 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 0b100011111. 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. This is explained in the following diagram as well-




Why do we need Parity Bit? (Click to Enlarge)

Limitations of using single parity bit is that it can detect only single bit errors. If two bits are changed simultaneously, it fails. Using Hamming Code is a better solution, but it doesn't fit in for USART and is out of the scope of this tutorial as well!

The Parity Generator calculates the parity bit for the serial frame data. When parity bit is enabled (UPM1 = 1), the Transmitter control logic inserts the parity bit between the last data bit and the first stop bit of the frame that is sent. The parity setting bits are available in the UPM1:0 bits in the UCSRC Register.

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

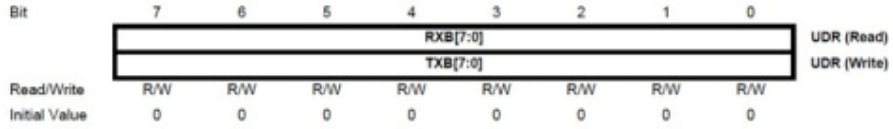
Parity Settings (Click to Enlarge)

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

# Register Description

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

## UDR: USART Data Register (16-bit)

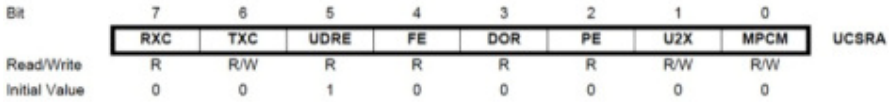



UDR – UART Data Register (Click to Enlarge)

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)




UCSRA – USART Control and Status Register A (Click to Enlarge)

- **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 | |

UCSRB – USART Control and Status Register B (Click to Enlarge)

- **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 | |

UCSRC – USART Control Register C (Click to Enlarge)

- **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 |

Synchronous/Asynchronous Selection (Click to Enlarge)

- **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 |

Parity Settings (Click to Enlarge)

- **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 |

Stop Bit Settings (Click to Enlarge)

- **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 |

Data Bit Settings (Click to Enlarge)

- **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 |

UCPOL Bit Settings (Click to Enlarge)

## 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 | |

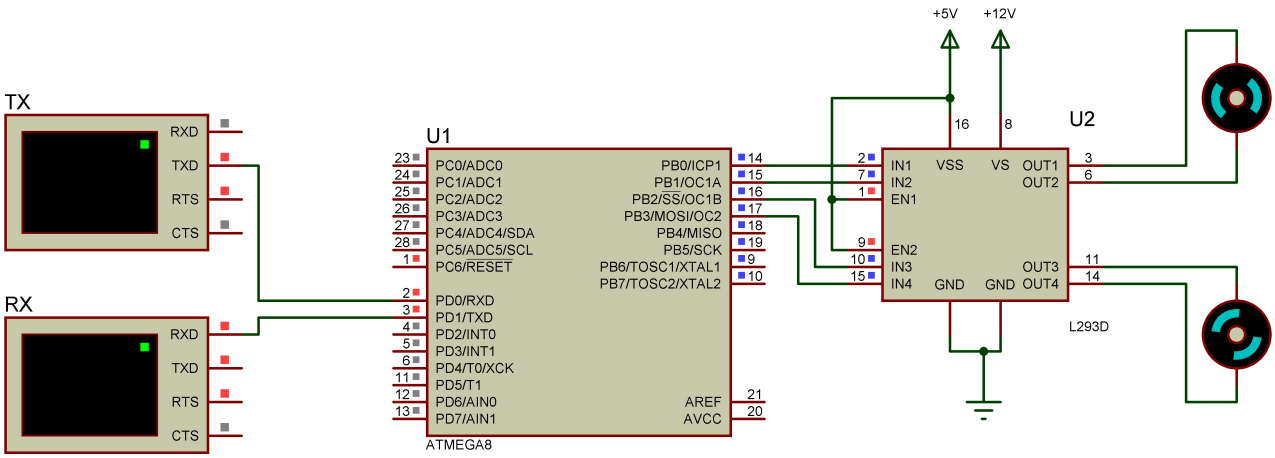
UBRR Register (Click to Enlarge)

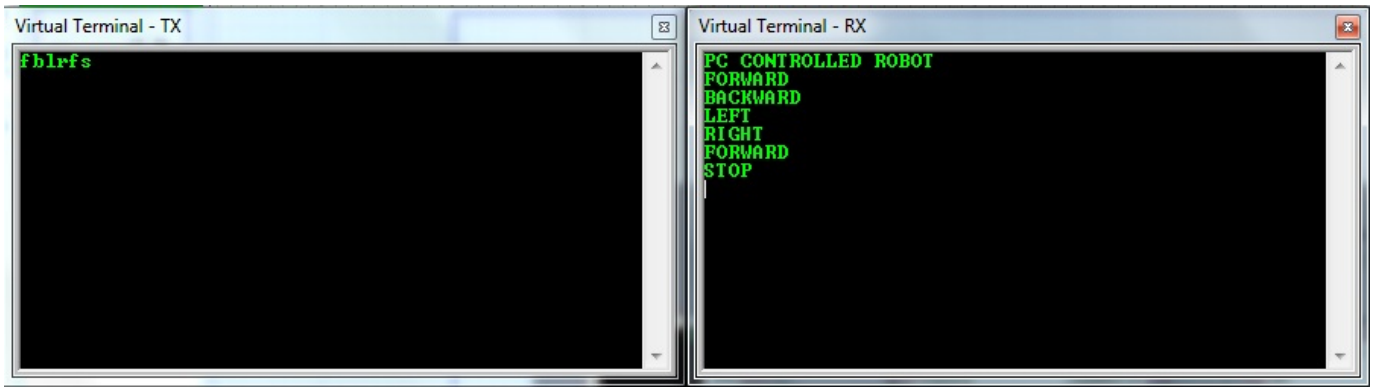
We have already seen this register, except the URSEL bit.

- **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.

```c
//Program to control Robot from PC and Transmitting the same message

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

//Serial Initialization

void USARTInit(unsigned int ubrr_value)//(uint16_t ubrr_value)
{
   UBRRL = ubrr_value; //Baud Rate value (Refer Baud Rate Tabular Column)
   UCSRA = ( 1<<U2X); //Doubling Baud Rate
   UCSRB = ( 1<<TXEN)|(1<<RXEN); //Enabling Transmitter and Receiver
}

//Reading a Character
unsigned char USARTReadChar()
{
 while(!(UCSRA & (1<<RXC))); //waiting for the data to be received
 return UDR; //reading data
}

//Transmitting a Character
void USARTWriteChar(char data)
{
 while(!(UCSRA & (1<<UDRE))); //Waiting for the buffer gets cleared
 UDR=data; //loading data
 while(!(UCSRA & (1<<TXC))); //Waiting for the data to be transmitted
}

//Transmitting a String
void USARTString(char *str)
{
    while(*str!='\0') //loop continues till variable reads null character in the string
    {
        USARTWriteChar(*str); //calling transmit data function
        str++; //updating address
    }
}

//Main function
void main()
{
 unsigned char data;
 DDRB=0xFF;
 DDRD=0x00;
 USARTInit(12);  //Serial intitialization with 9600 Baud Rate
 USARTString("PC CONTROLLED ROBOT\r"); //Transmitting Intial Message
 while(1) //Forever loop
 {
    data = USARTReadChar(); //Reading character and loading information to variable
    if(data=='f') //if 'f' received
    {
        PORTB = 6; //Motor Forward
        USARTString("FORWARD\r"); //Transmitting message
```

```
    }
    if(data=='b') //if 'b' received
    {
        USARTString("BACKWARD\r"); //
        PORTB = 9; //Motor Backward
    }
    if(data=='l') //if 'l' received
    {
        PORTB = 4; //LEFT
        USARTString("LEFT\r");
    }
    if(data=='r') //if 'r' received
    {
        PORTB = 2; //RIGHT
        USARTString("RIGHT\r");
    }
    if(data=='s')
    {
        PORTB = 0;//STOP
        USARTString("STOP\r");
    }
  }
}
```



# Serial Peripheral Interface – SPI Basics

Hey all! It's time to continue with our tutorials on serial communication. Till now, we have covered the following:

- An introduction to serial communication
- Basics of RS232 communication
- The UART/USART of the AVR

Apart from this, there are few other serial transfer protocols like SPI, I2C, etc. In this post, we will discuss about SPI and its bus transactions – no programming, just the concepts. Programming the SPI of (AVR) microcontrollers will be discussed in upcoming post.

# Contents



# Serial Peripheral Interface (SPI)

Serial Peripheral Interface, often shortened as SPI (pronounced as *spy*, or *ess-pee-eye*), is a synchronous serial data transfer protocol named by Motorola. Here two or more serial devices are connected to each other in full-duplex mode. The devices connected to each other are either *Master* or *Slave*. In a SPI link there could as many Masters and Slaves as required, but it's very rare to find more than one Master in a SPI link.

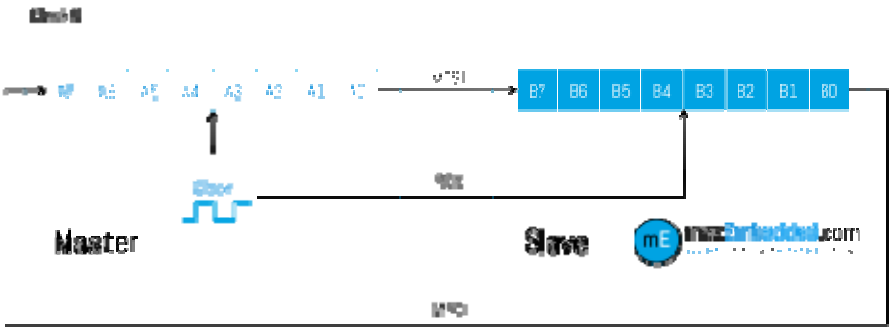
The *Master* device is the one which initiates the connection and controls it. Once the connection is initiated, then the *Master* and one or more *Slave(s)* can transmit and/or receive data. As mentioned earlier, this is a full-duplex connection, which means that *Master* can send data to *Slave(s)* and the *Slave(s)* can also send the data to the *Master* at the same time.

# SPI Bus Transaction

Now that we have a basic knowledge of what SPI is, let's look into the operation of SPI Bus. The SPI operation is based upon shift registers. Every device, whether *Master* or *Slave* has an 8-bit shift register inside it. The size of the shift register could be more than 8-bit as well (like 10-bit, 12-bit, etc), but it should be the same for both *Master* and *Slave*, and the protocol should support it.

## Hardware Setup

The *Master* and *Slave* are connected in such a way that the two shift registers form an inter-device circular buffer. The following diagram should explains the hardware setup. Please click on the images to enlarge it and view it in high resolution.




Hardware Setup of Master-Slave Device and Shift Registers (Click to Enlarge)

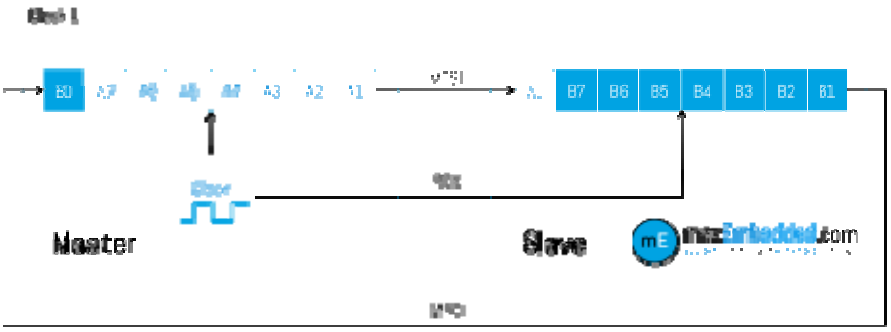
As you can see, there is an 8-bit shift register inside each of the *Master* and *Slave* devices. These shift registers operate in Serial-In/Serial-Out (SISO) fashion. The output of the *Master's* shift register is connected to the input of the *Slave's* shift register; and the output of the *Slave's* shift register is connected to the input of *Master's* shift register. This makes the connection operate like a circular/ring buffer. Don't bother about the names MISO, MOSI and SCK now. We will discuss about them a little later in this post.

As mentioned earlier, SPI is a synchronous serial data transfer protocol, which means that there must be a clock to synchronize the data transfer. It has also been stated that the *Master* is responsible for initiating and controlling the connection. Thus, we use the clock source of the *Master* device to synchronize the data transfer. That's why you see the clock source inside the *Master*, which controls the operation of *both* the shift registers.

# Data Transfer Operation

Alright, now let's see how SPI bus transfers data among *Master* and *Slave*. Let's refer to the diagram shown in the above section. Let's say that the data in the *Master's* shift register is A7 through A0 (MSB through LSB) whereas the data in the *Slave's* shift register is B7 through B0 (MSB through LSB). This is the initial state before any clock pulse arrives.

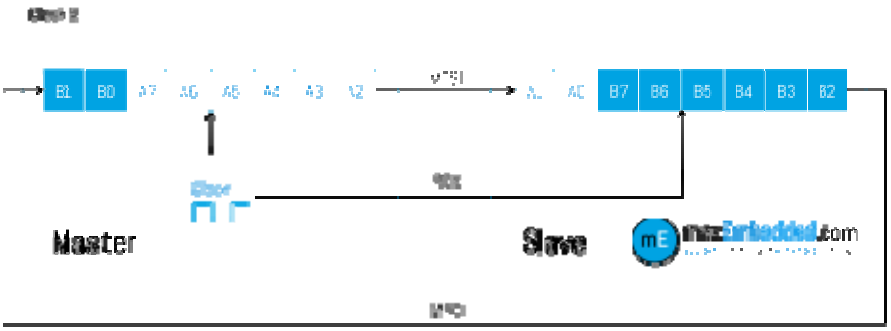
Now as soon as a clock pulse arrives, the shift registers come into operation and the data in the registers in shifted by one bit towards the right. This evicts bit A0 from *Master* and bit B0 from *Slave*. Since the *Master* and *Slave* are connected to form a ring/circular buffer, the evicted bit occupies the MSB position of the other device. Which means, bit A0 gets evicted from *Master* and occupies MSB position in *Slave's* shift register; whereas bit B0 gets evicted from *Slave* and occupies MSB position in *Master's* shift register. This can be seen in the following image. Bits are color coded for better viewing. Please click on the image to enlarge it.




Clock Pulse 1 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

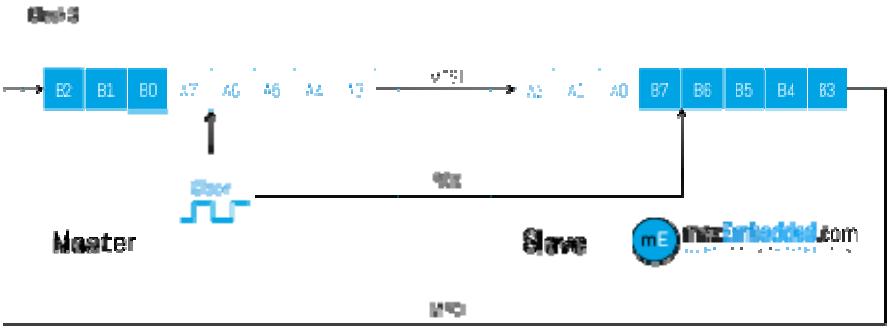
Now once again, when the clock generates another pulse, the data in the registers are shifted by another bit towards right, which evicts bits A1 and B1 from *Master* and *Slave* respectively. The evicted bits A1 and B1 occupy the MSB position of *Slave's* and *Master's* shift registers respectively. This can be seen in the following image. Please click on the image to enlarge it.



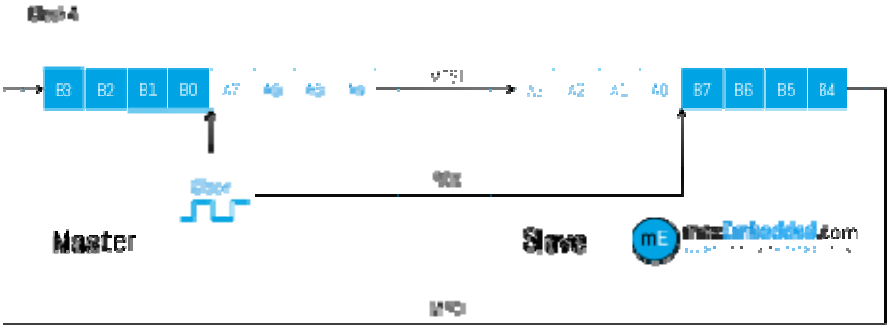


Clock Pulse 2 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

This continues for six more clock pulses. The following images depict the data transfer operation. Please click on the images to enlarge them for better viewing.

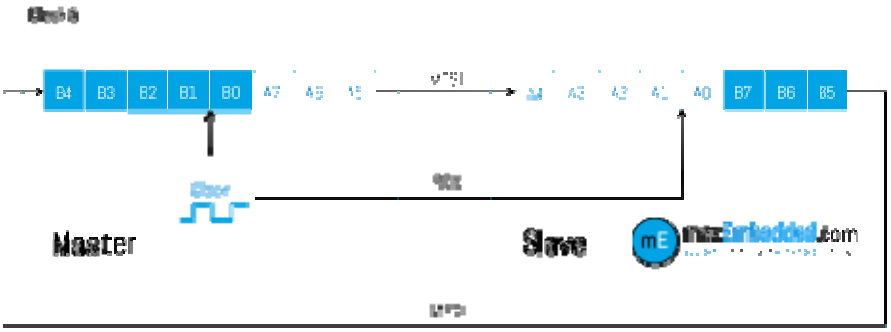



Clock Pulse 3 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

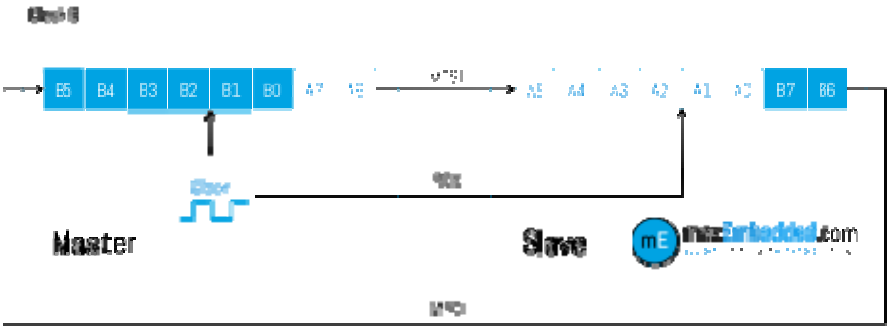



Clock Pulse 4 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

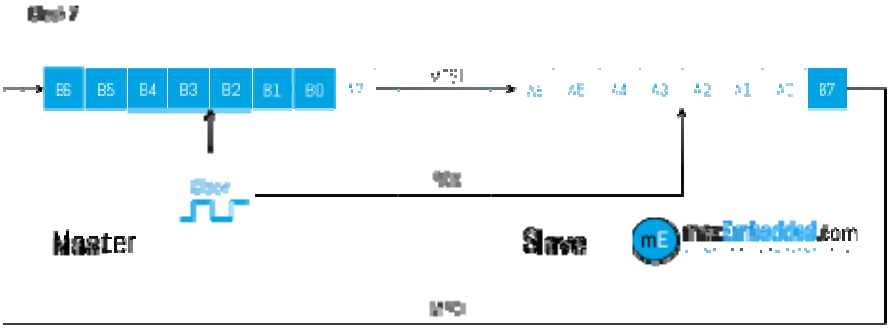





Clock Pulse 5 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)



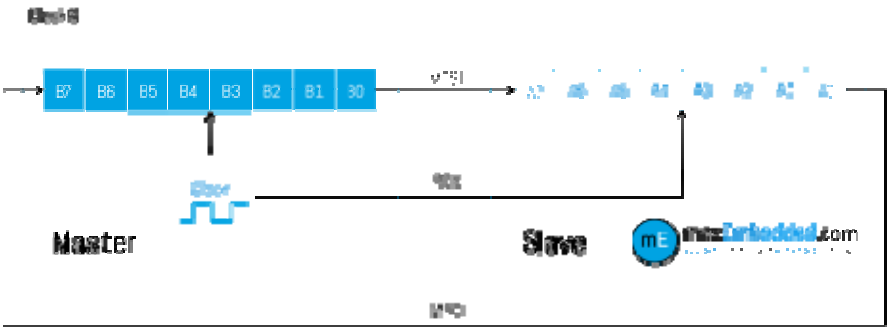

Clock Pulse 6 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)




Clock Pulse 7 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

And finally,






Clock Pulse 8 – SPI Bus Transaction showing Master Slave Shift Registers (Click to Enlarge)

## Short Summary

To sum up,

- Both, *Master* and *Slave* place the data (byte) they wish to transfer in their respective shift registers before the communication starts.

- *Master* generates 8 clock pulses. After each clock pulse, one bit of information is transfer from *Master* to *Slave* and vice-versa.
- After 8 clock pulses, *Master* would have received *Slave's* data, whereas *Slave* would have *Master's* data. And that's why this is a full-duplex communication.

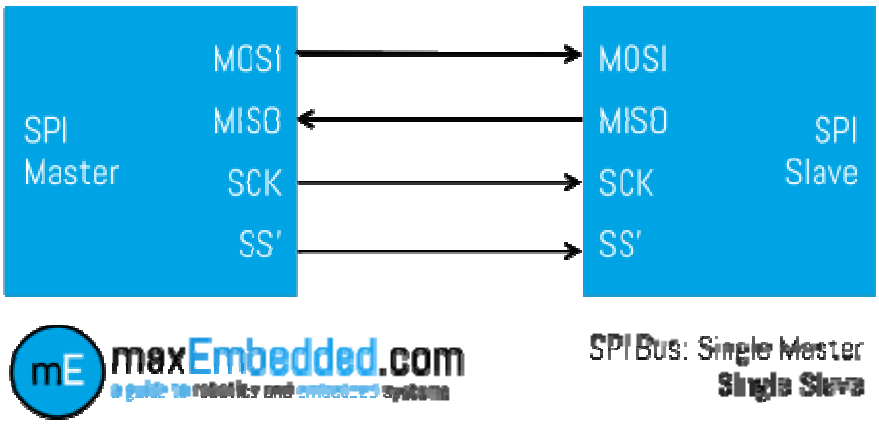
# SPI Bus Interface

Now that we are conceptually clear how the data transfer takes place, let's look into the SBI bus description and the interface between *Master* and *Slave*.

The *Master* and *Slave* are connected by means of four wires. Each of these wires carries a particular signal defined by the SPI bus protocol. These four signals/wires are–

1. **MOSI – Master Out Slave In:** This is the wire/signal which goes from the output of *Master's* shift register to the input of the *Slave's* shift register.
2. **MISO – Master In Slave Out:** This is the wire/signal which goes from the output of *Slave's* shift register to the input of the *Master's* shift register.
3. **SCK/SCLK – Serial Clock:** This is the output of the clock generator for *Master* and clock input for *Slave*.
4. **SS' – Slave Select:** This is discussed in the next section of this post.

The MOSI, SCK and SS' signals are directed from *Master* to *Slave* whereas the MISO signal is directed from *Slave* to *Master*. The following diagram represents this interface having single *Master* and single *Slave*.




SPI Bus – Single Master Single Slave (Click to Enlarge)

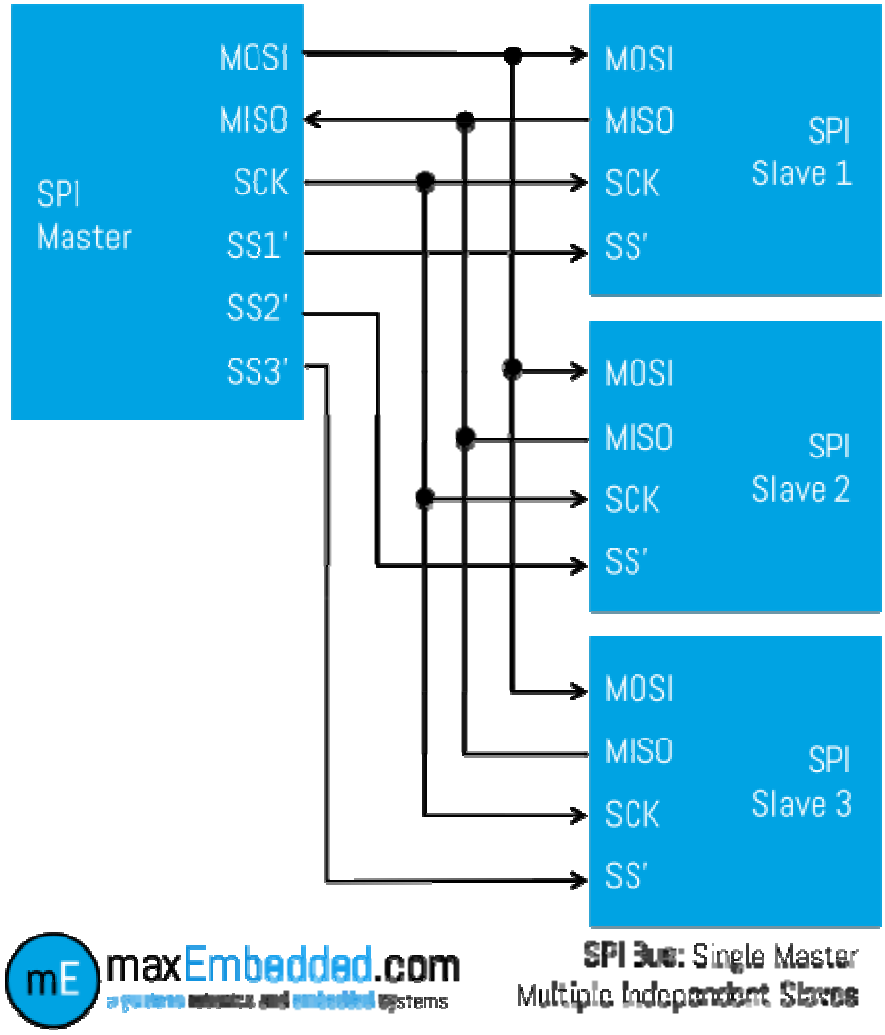
Thus, it should be noted again that during each SPI clock cycle, a full duplex transmission occurs as follows–

- *Master* sends a bit to the MOSI line; *Slave* reads it from the same line.
- *Slave* sends a bit to the MISO line; *Master* reads it from the same line.

# Multiple Slaves – Slave Select (SS') Signal

As mentioned earlier, SPI can be used to connect one *Master* to multiple *Slaves* as well. Having multiple *Masters* is also possible, but it does nothing but increase the complexity due to clock synchronization issues, and is very very rare. Having multiple *Slaves* is where the Slave Select (SS') signal comes into effect.

SS' (which means SS complemented) signal is in active low configuration i.e. to select a particular *Slave*, we need to provide a LOW signal level to SS' of the *Slave*. The SPI bus interface is pretty simple for this one, and is shown in the diagram shown below.




SPI Bus – Single Master Multiple Slaves (Click to Enlarge)

All the *Slaves* share the same MOSI, MISO and SCK signals. The SS' signal is responsible for choosing a particular *Slave*. The *Slave* gets enabled only when its input SS' signal goes LOW.

In the above case, each of the *Slaves* is independent since they are separately selected via independent SS' signals from the *Master*. However, there is another way to link the Slaves together – by using Daisy chain configuration. In this configuration, all the Slaves are selected at a time, and the output of one *Slave* goes to the input of another *Slave*, and so on. However we will not be discussing this here (and in upcoming posts as well) since most of the applications don't require this type of configuration.

# Clock Polarity and Phase

Keeping synchronization in mind, *Master's* role doesn't end with simply generating clock pulses at a particular frequency (usually within the range of 10 kHz to 100 MHz). In fact, *Master* and *Slave* should agree on a particular synchronization protocol as well, or else everything will go wrong and data will get lost. This is where the concept of clock polarity (CPOL) and clock phase (CPHA) comes in.

- **CPOL – Clock Polarity:** This determines the base value of the clock i.e. the value of the clock when SPI bus is idle.
    - When CPOL = 0, base value of clock is zero i.e. SCK is LOW when idle.
    - When CPOL = 1, base value of clock is one i.e. SCK is HIGH when idle.
- **CPHA – Clock Phase:** This determines the clock transition at which data will be sampled/captured.
    - When CPHA = 0, data is sampled at clock's rising/leading edge.
    - When CPHA = 1, data is sampled at clock's falling/trailing edge.

This results in four SPI modes, shown in the table below taken from the ATmega32 datasheet page 139. We will discuss more about these modes and how to choose them in our next post where we will learn how to program the SPI of the AVR.

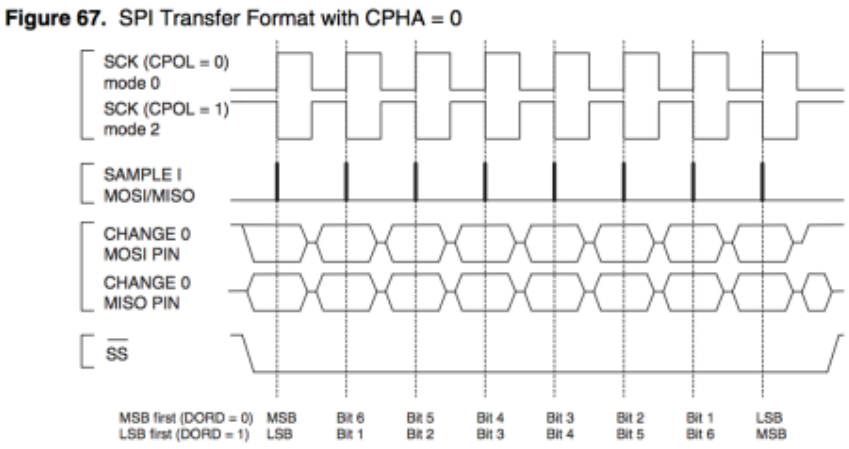


**Table 59.** CPOL and CPHA Functionality

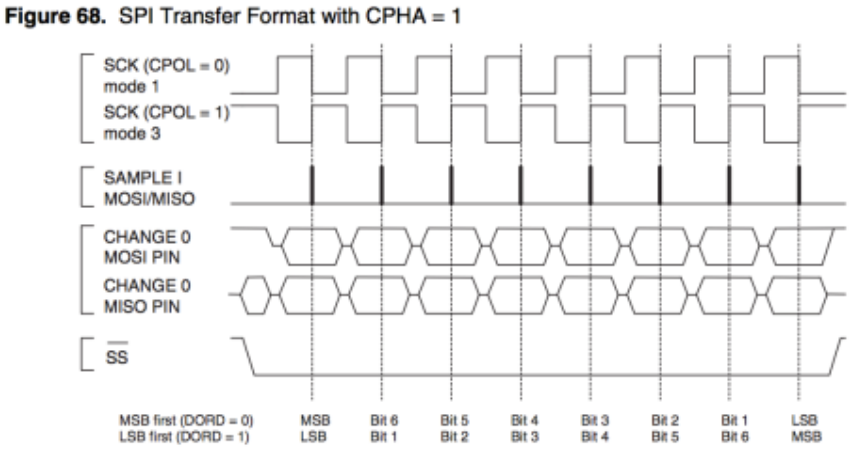
| | Leading Edge | Trailing Edge | SPI Mode |
|---|---|---|---|
| CPOL = 0, CPHA = 0 | Sample (Rising) | Setup (Falling) | 0 |
| CPOL = 0, CPHA = 1 | Setup (Rising) | Sample (Falling) | 1 |
| CPOL = 1, CPHA = 0 | Sample (Falling) | Setup (Rising) | 2 |
| CPOL = 1, CPHA = 1 | Setup (Falling) | Sample (Rising) | 3 |

CPOL and CPHA Functionality

We can also look into the timing diagrams provided in the same page of the datasheet. By now I guess you should be able to decode the timing diagrams yourself. Don't worry about the DORD setting at the bottom, we will discuss about it in the next post. Just focus on what effect CPOL and CPHA has in these figures.




SPI Transfer Format with CPHA = 0 (Click to Enlarge)

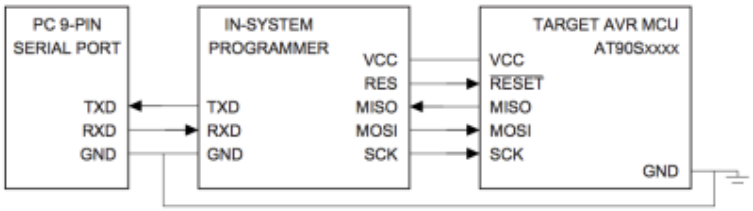



SPI Transfer Format with CPHA = 1 (Click to Enlarge)

# AVR In-System Programming

The In-System Programming (ISP) exploits the technique of SPI to transfer the hex code from the PC to the target AVR microcontroller. We won't go into the details of how it happens. Since we are discussing about SPI, I thought to bring up this small point as well. The following figure taken from AVR Application Note 910 page 2 shows the simplified hardware connections.

**Figure 1.** Six-wire Connection Between Programmer and Target System




Connections between ISP and AVR MCU (Click to Enlarge)

This means that if you have connected some SPI device to your AVR microcontroller, and at the same time you are trying to program your microcontroller, there could be some issues (like driver contention). We will discuss about this issue and other hardware considerations in our next post.

# Interesting Reads

You might be interested in reading the following documents–

- Motorola's SPI Block Guide v03.06
- AVR910: In-System Programming

# Summary

Let's look at what we have learnt in this post.

- SPI is a full-duplex synchronous serial data transfer protocol.
- Data transfer takes place in between *Master* and *Slave* devices.
- Each *Master/Slave* device has an internal 8 bit shift register, which is connected to other devices so as to form a circular/ring buffer.
- At each clock pulse, data gets right shifted in the circular/ring buffer.
- After 8 clock pulses, data is completely exchanged in between devices.
- SPI bus consists of four wires/signals – MOSI, MISO, SCK and SS'.
- When we connect more than one *Slave* devices, then we choose them using the SS' signal.
- CPOL and CPHA must be set so that *Master* and *Slave* devices sync properly.
- AVR ISP uses SPI to program the microcontroller.

So this was all about the basics of SPI. If you want to know about the pros and cons of SPI, I would suggest you to read this Wikipedia section. **In the next post, we will learn how to implement SPI in an AVR microcontroller. So subscribe to stay updated! And don't forget to write your views about this post below.**

# The SPI of the AVR

Continuing with the series of tutorials on Serial Communication, here is another one, and much awaited, the Serial Peripheral Interface (SPI) of AVR! Before proceeding ahead, I would suggest you to read Mayank's tutorial on the basics of SPI.

# Contents



# Serial Peripheral Interface (SPI) – Basics Revisited

Here we will discuss some basics of Serial Peripheral Interface (SPI, pronounced *spy* or *ess-pee-eye*). Mayank has already dealt with the basics of SPI and SPI bus transactions in the previous tutorial, but I will go over some of the nitty-gritties here again.

Serial Peripheral Interfacing is one of the most used serial communication protocols, and very simple to use! As a matter of fact, I find this one much simpler than USART! 

Since SPI has been accepted as a *de facto* standard, it is available in almost all architectures, including 8051, x86, ARM, PIC, AVR, MSP etc., and is thus widely used. This means that

there shouldn't be any portability issues and you can connect devices of two different architectures together as well!

So here are the most popular **applications of SPI**:

1. Wired transmission of data (though the first preference is mostly USART, but SPI *can* be used when we are using multiple slave or master systems, as addressing is much simpler in SPI).
2. Wireless transmissions through Zigbee, 2.4GHz etc.
3. Programming your AVR chips (Yes! They are programmed through the SPI! You'll would have read about it in Mayank's Post on SPI).
4. It is also used to talk to various peripherals – like sensors, memory devices, real time clocks, communication protocols like Ethernet, etc.

# Advantages of SPI

SPI uses 4 pins for communications (which is described later in this post) while the other communication protocols available on AVR use lesser number of pins like 2 or 3. Then why does one use SPI? Here are some of the advantages of SPI:

1. Extremely easy to interface! (It took me much less time to setup and transmit data through SPI as compared to I2C and UART!)
2. Full duplex communication
3. Less power consumption as compared to I2C
4. Higher hit rates (or throughput)

And a lot more!

But there are some disadvantages as well, like higher number of wires in the bus, needs more pins on the microcontroller, etc.

# Master and Slave

In SPI, every device connected is either a *Master* or a *Slave*.

The *Master* device is the one which initiates the connection and controls it. Once the connection is initiated, then the *Master* and one or more *Slave(s)* can transmit and/or receive data. As mentioned earlier, this is a full-duplex connection, which means that *Master* can send data to *Slave(s)* and the *Slave(s)* can also send the data to the *Master* at the same time.

As I said earlier, SPI uses 4 pins for data communication. So let's move on to the pin description.

## Pin Description

The SPI typically uses 4 pins for communication, wiz. MISO, MOSI, SCK, and SS. These pins are directly related to the SPI bus interface.

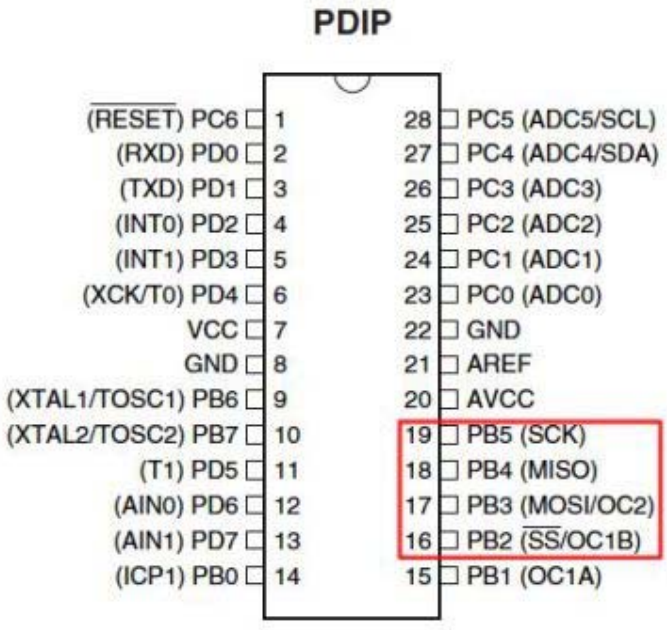
1. **MISO –** MISO stands for Master InSlave Out. MISO is the input pin for *Master* AVR, and output pin for *Slave* AVR device. Data transfer from *Slave* to *Master* takes place through this channel.
2. **MOSI** - MOSI stands for Master OutSlave In. This pin is the output pin for *Master* and input pin for *Slave*. Data transfer from *Master* to *Slave* takes place through this channel.
3. **SCK** - This is the SPI clock line (since SPI is a synchronous communication).
4. **SS** - This stands for *Slave Select*. This pin would be discussed in detail later in the post.

Now we move on to the SPI of AVR!

# The SPI of the AVR

The SPI of AVRs is one of the most simplest peripherals to program. As the AVR has an 8-bit architecture, so the SPI of AVR is also 8-bit. In fact, usually the SPI bus is of 8-bit width. It is available on PORTB on all of the ICs, whether 28 pin or 40 pin.

Some of the images used in this tutorial are taken from the AVR datasheets.



SPI pins on 28 pin ATmega8

# Register Descriptions

The AVR contains the following three registers that deal with SPI:

1. **SPCR – SPI Control Register** – This register is basically the master register i.e. it contains the bits to initialize SPI and control it.
2. **SPSR - SPI Status Register** - This is the status register. This register is used to read the status of the bus lines.
3. **SPDR – SPI Data Register** - The SPI Data Register is the read/write register where the actual data transfer takes place.

## The SPI Control Register (SPCR)

As is obvious from the name, this register controls the SPI. We will find the bits that enable SPI, set up clock speed, configure master/slave, etc. Following are the bits in the SPCR Register.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | SPIE | SPE | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 | SPCR |
| 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: SPIE – SPI Interrupt Enable**
The SPI Interrupt Enable bit is used to enable interrupts in the SPI. Note that global interrupts must be enabled to use the interrupt functions. Set this bit to '1' to enable interrupts.

**Bit 6: SPE – SPI Enable**
The SPI Enable bit is used to enable SPI as a whole. When this bit is set to 1, the SPI is enabled or else it is disabled. When SPI is enabled, the normal I/O functions of the pins are overridden.

**Bit 5: DORD – Data Order**
DORD stands for Data ORDer. Set this bit to 1 if you want to transmit LSB first, else set it to 0, in which case it sends out MSB first.

**Bit 4: MSTR – Master/Slave Select**
This bit is used to configure the device as *Master* or as *Slave*. When this bit is set to 1, the SPI is in *Master*mode (i.e. clock will be generated by the particular device), else when it is set to 0, the device is in SPI *Slave* mode.

**Bit 3: CPOL – Clock Polarity**
This bit selects the clock polarity when the bus is idle. Set this bit to 1 to ensure that SCK is HIGH when the bus is idle, otherwise set it to 0 so that SCK is LOW in case of idle bus.

This means that when CPOL = 0, then the leading edge of SCK is the rising edge of the clock. When CPOL = 1, then the leading edge of SCK will actually be the falling edge of the clock. Confused? Well, we will get back to it a little later in this post again.

| CPOL | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Rising | Falling |
| 1 | Falling | Rising |

CPOL Functionality

**Bit 2: CPHA – Clock Phase**
This bit determines when the data needs to be sampled. Set this bit to 1 to sample data at the leading (first) edge of SCK, otherwise set it to 0 to sample data at the trailing (second) edge of SCK.

| CPHA | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Sample | Setup |
| 1 | Setup | Sample |

CPHA Functionality

**Bit 1,0: SPR1, SPR0 – SPI Clock Rate Select**
These bits, along with the SPI2X bit in the SPSR register (discussed next), are used to choose the oscillator frequency divider, wherein the $f_{OSC}$ stands for internal clock, or the frequency of the crystal in case of an external oscillator.

The table below gives a detailed description.



| SPI2X | SPR1 | SPR0 | SCK Frequency |
|---|---|---|---|
| 0 | 0 | 0 | $f_{osc}/4$ |
| 0 | 0 | 1 | $f_{osc}/16$ |
| 0 | 1 | 0 | $f_{osc}/64$ |
| 0 | 1 | 1 | $f_{osc}/128$ |
| 1 | 0 | 0 | $f_{osc}/2$ |
| 1 | 0 | 1 | $f_{osc}/8$ |
| 1 | 1 | 0 | $f_{osc}/32$ |
| 1 | 1 | 1 | $f_{osc}/64$ |

Frequency Divider

# The SPI Status Register (SPSR)

The SPI Status Register is the register from where we can *get* the status of the SPI bus and interrupt flag is also set in this register. Following are the bits in the SPSR register.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | SPIF | WCOL | – | – | – | – | – | SPI2X | SPSR |
| Read/Write | R | R | R | R | R | R | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

SPSR Register

**Bit 7: SPIF – SPI Interrupt Flag**
The SPI Interrupt Flag is set whenever a serial transfer is complete. An interrupt is also generated if SPIE bit (bit 7 in SPCR) is enabled and global interrupts are enabled. This flag is cleared when the corresponding ISR is execute

**Bit 6: WCOL – Write Collision Flag**
The Write COLlision flag is set when data is written on the SPI Data Register (SPDR, discussed next) when there is an impending transfer or the data lines are busy.

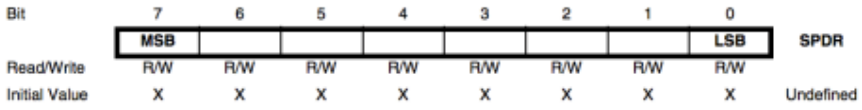
This flag can be cleared by first reading the SPI Data Register when the WCOL is set. Usually if we give the commands of data transfer properly, this error does not occur. We will discuss about how this error can be avoided, in the later stages of the post.

**Bit 5:1**
These are reserved bits.

**Bit 0: SPI2x – SPI Double Speed Mode**
The SPI double speed mode bit reduces the frequency divider from 4x to 2x, hence doubling the speed. Usually this bit is not needed, unless we need very specific transfer speeds, or very high transfer speeds. Set this bit to 1 to enable SPI Double Speed Mode. This bit is used in conjunction with the SPR1:0 bits of SPCR Register.

# The SPI Data Register (SPDR)

The SPI Data register is an 8-bit read/write register. This is the register from where we read the incoming data, and write the data to which we want to transmit.



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | MSB | | | | | | | LSB | SPDR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | X | X | X | X | X | X | X | X | Undefined |

SPDR Register

The $7^{th}$ bit is obviously, the Most Significant Bit (MSB), while the $0^{th}$ bit is the Least Significant Bit (LSB).

Now we can relate it to bit 5 of SPCR – the DORD bit. When DORD is set to 1, then LSB, i.e. the $0^{th}$ bit of the SPDR is transmitted first, and vice versa.

# Data Modes

The SPI offers 4 data modes for data communication, wiz SPI Mode 0,1,2 and 3, the only difference in these modes being the clock edge at which data is sampled. This is based upon the selection of CPOL and CPHA bits.

The table below gives a detailed description and you would like to refer to this for a more detailed explanation and timing diagrams.

| | Leading Edge | Trailing Edge | SPI Mode |
|---|---|---|---|
| CPOL = 0, CPHA = 0 | Sample (Rising) | Setup (Falling) | 0 |
| CPOL = 0, CPHA = 1 | Setup (Rising) | Sample (Falling) | 1 |
| CPOL = 1, CPHA = 0 | Sample (Falling) | Setup (Rising) | 2 |
| CPOL = 1, CPHA = 1 | Setup (Falling) | Sample (Rising) | 3 |

# The Slave Select (SS') Pin

As you would see in the next section, the codes of SPI are fairly simple as compared to those of UART, but the major headache lies here: the SS' pin!

SS' (means SS complemented) works in active low configuration. Which means to select a particular slave, a LOW signal must be passed to it.

***When set as input, the SS' pin should be given as HIGH (Vcc) on as Master device, and a LOW (Grounded) on a Slave device.***

When as an output pin on the *Master* microcontroller, the SS' pin can be used as a GPIO pin.

The SS pin is actually what makes the SPI very interesting! But before we proceed, one question is that why do we need to set these pins to some value?

The answer is, that when we are communicating between multiple devices working on SPI through the same bus, the SS' pin is used to select the slave to which we want to communicate with.

Let us consider the following two cases to understand this better:

1. **When there are multiple slaves and a single master.**
   In this case, the SS' pins of all the slaves are connected to the master microcontroller. Since we want only a specific slave to receive the data, the master microcontroller would give a low signal to the SS' pin of that specific microcontroller, and hence only that slave microcontroller would receive data.
2. **When there are multiple masters and a single slave.**
   A similar setup as above can be used in this case as well, the difference being that the SS'

lines of all the masters is controlled by the slave, while the slave SS' line is always held low. The slave would select the master through which it has to receive data by pulling its SS' high.Alternatively, a multiplexed system can be used where each master microcontroller can control every other master microcontroller's SS' pin, and hence when it has to transmit data, it would pull down every other master microcontroller's SS' Pin, while declaring its own SS' as output.
You can also refer to this if you are still confused regarding Slave Select.

# Problem Statement

Let's assume a problem statement. Say the given problem statement is to send some data from *Master* to *Slave*. The *Slave* in return sends an acknowledgement (ACK) data back to the *Master*. The *Master* should check for this ACK in order to confirm that the data transmission has completed. This is a typical example of full duplex communication. While the *Master* sends the data to the *Slave*, it receives the ACK from the *Slave* simultaneously.

# Methodology

We would use the primary microcontroller (ATmega8 in this case) as the *Master* device, and a secondary microcontroller (ATmega8 in this case) as the *Slave* device. A counter increments in the *Master* device, which is being sent to the *Slave* device. The *Master* then checks whether the received data is the same as ACK or not (ACK is set as 0x7E in this case). If the received data is the same as ACK, it implies that data has been successfully sent and received by the *Master* device. Thus, the *Master* blinks an LED connected to it as many number of times as the value of the counter which was sent to the *Slave*. If the *Master* does not receive the ACK correctly, it blinks the LED for a very long time, thus notifying of a possible error.

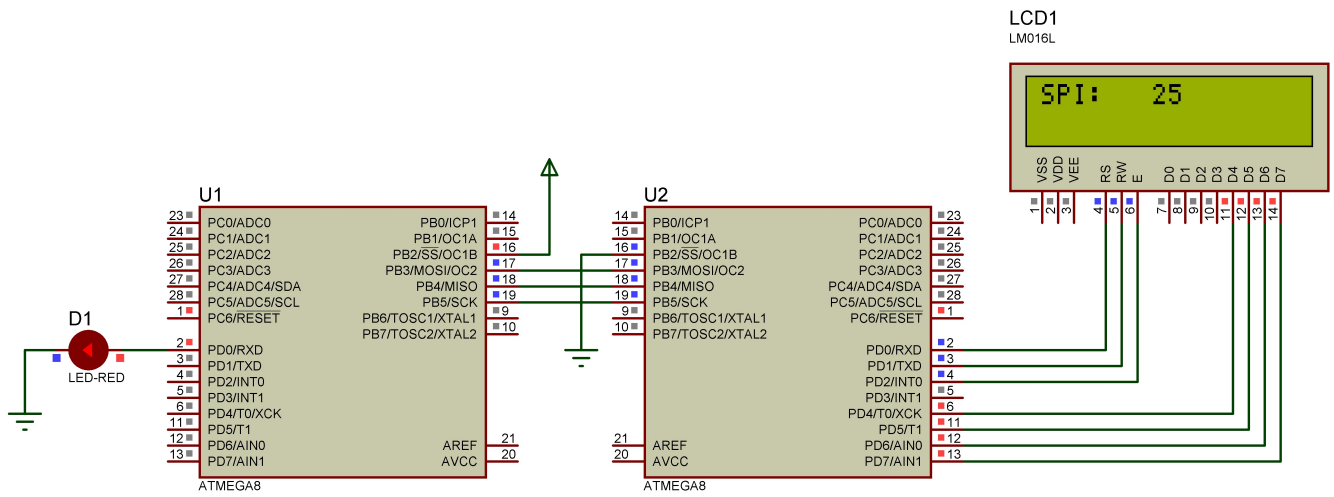
On the other hand, *Slave* waits for data to be received from the *Master*. As soon as data transmission begins (from *Master* to *Slave*, the *Slave* sends ACK (which is 0x7E in this case) to the *Master*. The *Slave* then displays the received data in an LCD.

# Hardware Connections

Hardware connections are simple. Both the MOSI pins are connected together, MISO pins are connected together and the SCK pins are also connected together. The SS' pin of the slave is grounded whereas that of master is left unconnected. And then we have connected the LCD to the slave as well.

```
//MASTER
//Program to send count and receive acknowledgment (MASTER)
#include <avr/io.h>
#include <util/delay.h>

#define ACK 0x7E

#define MOSI PB3
#define MISO PB4
#define SCK PB5
//Initialize SPI Master Device
void spi_init_master (void)
{
    DDRB = (1<<SCK)|(1<<MOSI);              //Set MOSI, SCK as Output
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0); //Enable SPI, Set as Master    //Prescaler: Fosc/16
}
//Function to send and receive data
char spi_tranceiver (char data)
{
    SPDR = data;                        //Load data into the buffer
    while(!(SPSR & (1<<SPIF)));          //Wait until transmission complete
    return SPDR;      //Return received data
}
//Function to blink LED
void led_blink (int i)
{     //Blink LED "i" number of times
    for (; i>0; --i)
    {
        PORTD = 1;
        _delay_ms(100);
        PORTD= 0;
        _delay_ms(100);
    }
}
//Main
void main()
{
    spi_init_master();                  //Initialize SPI Master
    DDRD |= 0x01;                       //PD0 as Output
    char data;                 //Received data stored here
    int x = 0;                      //Counter value which is sent
    _delay_ms(1000);
    while(1)
    {
        data = 0x00;                    //Reset ACK in "data"
        data = spi_tranceiver(++x);     //Send "x", receive ACK in "data"
        if(data == ACK)
        {               //Check condition
                        //If received data is the same as ACK, blink LED "x" number of times
            led_blink(x);
        }
        _delay_ms(1000);                //Wait
    }
}
```

```c
//SLAVE
//Program to recieve count value, displaying it on LCD and sending Acknowledgement to MASTER
#include <avr/io.h>
#include <util/delay.h>
//LCD Header File
#include "lcd.h"
#include "lcd.c"

#define ACK 0x7E

#define MOSI PB3
#define MISO PB4
#define SCK PB5
//Slave Initialization
void spi_init_slave ()
{
    DDRB=(1<<MISO);                                  //MISO as OUTPUT
    SPCR=(1<<SPE);                                  //Enable SPI
}
//Function to send and receive data
char spi_tranceiver (char data)
{
    SPDR = data;                                    //Load data into buffer
    while(!(SPSR &(1<<SPIF)));                       //Wait until transmission complete
    return SPDR;                                   //Return received data
}
//Main function
void main()
{
    lcd_init(LCD_DISP_ON);           //Initialize LCD
    spi_init_slave();                               //Initialize slave SPI
    char data, buffer[10];
    while(1)
    {
        lcd_gotoxy(0,0);                             //Fixing Position
        data = spi_tranceiver(ACK);                 //Receive data, send ACK
        sprintf(buffer,"SPI: %4d",data);                 //Convert integer into string
        lcd_puts(buffer);                           //Display received data
        _delay_ms(20);                              //Wait
    }
}
```



# Inter-Integrated Circuits – I2C Basics

Hello folks, it's time for I2C! Currently, related to Serial Communication, maxEmbedded features RS232 and SPI communication. It's time to move beyond! In this post, we will discuss all the theoretical concepts that you need to know regarding I2C before programming/testing it on real devices. We will deal with how to program the TWI/I2C of (AVR) microcontrollers in the next post.

# Contents

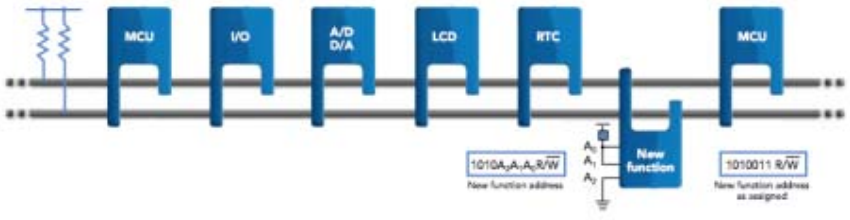
# Inter-Integrated Circuit ($I^2C$)

As the name suggests, Inter-IC (or the Inter-Integrated Circuit), often shortened as *I2C* (pronounced *eye-two-see*), *$I^2C$* (pronounced *eye-squared-see*), or *IIC*, was developed as a communication protocol to interact between different ICs on a motherboard, a simple internal bus system. It is a revolutionary technology developed by Philips Semiconductor (now NXP Semiconductors) in 1982, and is used to connect low speed peripherals (like keyboard, mouse, memory, IO/serial/parallel ports, etc.) to the motherboard (containing the CPU) operating at much higher speed.

These days you can find a lot of devices which are I2C compatible manufactured by a variety of companies (like Intel, TI, Freescale, STMicroelectronics, etc). Somewhere around the mid-1990s, Intel devised the SMBus protocol, a subset of I2C with strict protocols. Most modern day I2C devices  support both, I2C and SMBus with little reconfiguration.

# $I^2C$ Bus Interface

The most compelling thing about the I2C interface is that the devices are hooked up to the I2C bus with just two pins (and hence it is sometimes referred to as ***Two Wire Interface***, or the ***TWI***). Well of course, we do need two more pins for Vcc and ground, but that goes without saying.

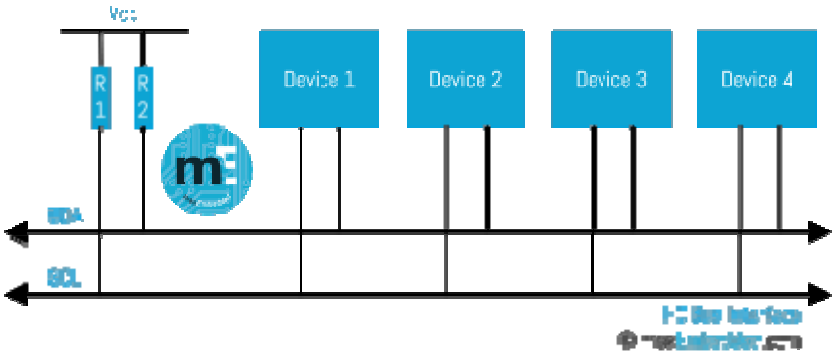



I2C Bus Interface (Image source eeweb.com)

As you can see in the above diagram (taken from eeweb.com), all the devices are hooked up to the same I2C bus with just two pins. These devices could be the CPU, or IO devices, or ADC, or any other device which supports the I2C protocol. All the devices connected to the

bus are classified as either being *Master* or *Slave* (just like SPI). We will discuss about it in a little while.

For now, let's get to know more about the bus itself. The I2C bus consists of two bidirectional "open-drain" lines – SDA and SCL – pulled up with resistors as shown below.




I2C Bus Interface

## Serial Data Line (SDA)

The **Serial Data Line (SDA)** is the data line (of course!). All the data transfer among the devices takes place through this line.

## Serial Clock Line (SCL)

The **Serial Clock Line (SCL)** is the serial clock (obviously!). I2C is a synchronous protocol, and hence, SCL is used to synchronize all the devices and the data transfer together. We'll learn how it works a little later in this post.

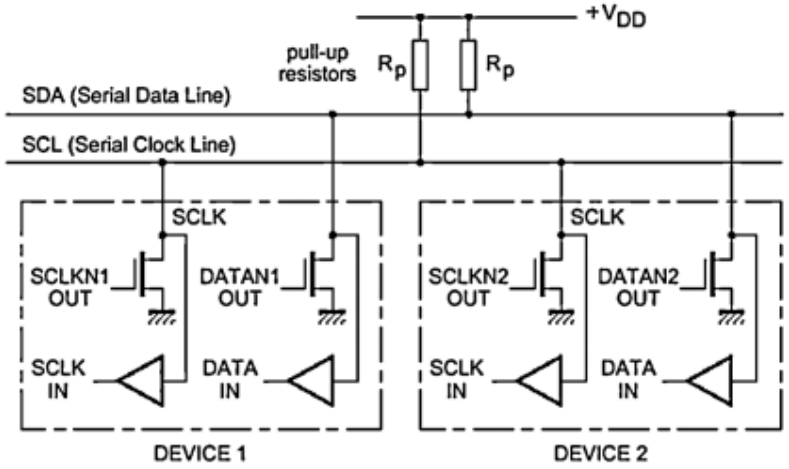
## Open-Drain Lines

A little while ago (just above the previous image), I mentioned that SDA and SCL are *open-drain* (also called *open-collector*) lines pulled up with resistors. What does that mean? It means that the devices connected to the I2C bus are capable of pulling any of these two lines

low, but they cannot drive them high. If any of the devices would ever want to drive the lines high, they would simply need to *let go* of that line, and it would be driven high by the pull up resistors (R1 and R2 in the previous image, or Rp in the next image).

For those who are interested, let's have a closer look. Others, please skip this section and go to the next to next section (voltage levels and resistor values).



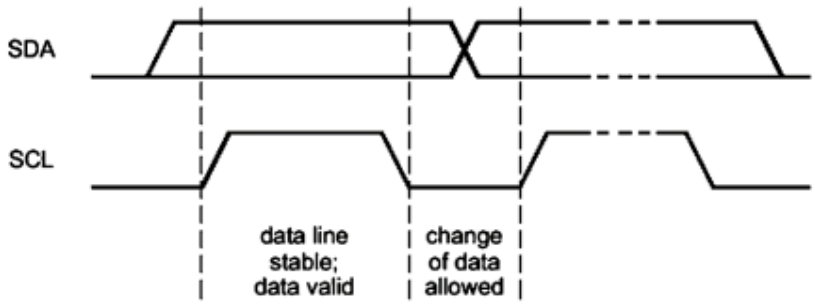



I2C Bus Interface – A Closer Look (Image source infoindustrielle.free.fr)

In the above image, you can clearly see the NMOS transistors inside the devices. In order for the device to pull any of the two lines low, it needs to provide a high voltage to the gate of the transistor (that's how an NMOS transistor operates, right?). If the gate voltage is low, the NMOS transistor is not activated and the corresponding line is driven high.

## $I^2C$ Data Validity

For the data to be valid on the SDA line, it must not change while the SCL is high. The data on the SDA line should change only and only when the SCL line goes low. If this standard is not followed, the data transfer becomes flawed, in which case it becomes a start/stop sequence (discussed later in this post). The following image illustrates the same.




I2C Data Validity (Image source infoindustrielle.free.fr)

### Voltage Levels and Resistor Values

I2C supports a wide range of voltage levels, hence you can provide +5 volts, or +3.3 volts as Vcc easily, and other lower/higher voltages as well. This gives us a wide range of choices for the values of the pull-up resistors (R1 and R2). Anything within the range of 1k to 47k should be fine, however values lower than 10k are usually preferred.

# Master and Slave

The concept of *Master* and *Slave* in I2C is quite similar to that of SPI. Just like SPI, all the devices are either *Master* or *Slave*. *Master* is the device which initiates the transfer and drives the clock line SCL. On a single I2C bus, there are usually multiple *Slaves* connected to a single *Master*.

However, just like SPI, we can also have multiple *Masters* connected to the same I2C bus. Since we want our lives to be a little simpler, we usually avoid such cases, but however I2C supports multi-bus master collision detection and arbitration for such cases (doesn't make sense? Let's forget about it for now!).

# Speed

I2C supports serial 8-bit bi-directional data transfers up to a speed of 100 kbps, which is the standard clock speed of SCL. However, I2C can also operate at higher speeds – Fast Mode (400 kbps) and High Speed Mode (3.4 Mbps). Most of the devices are built to operate up to

speeds of 100 kbps (remember that we discussed that I2C is used to connect low-speed devices?).

# $I^2C$ Bus Transaction

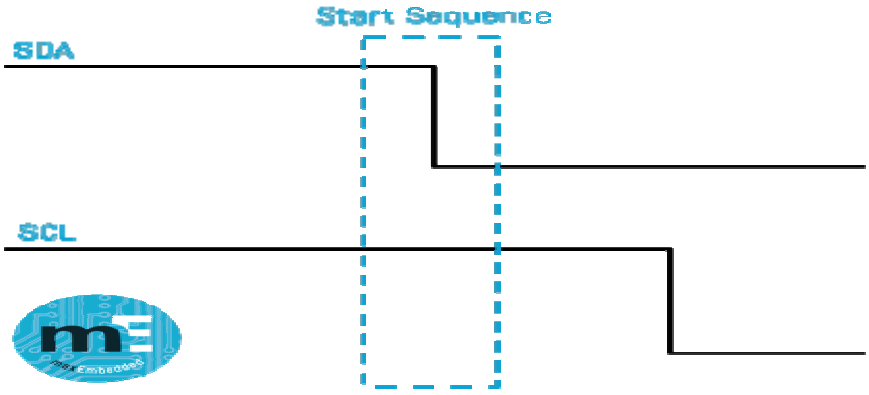
Alright, now that we are familiar with the I2C bus interface, let's look into how the data transfer actually takes place through that interface. I2C supports unidirectional as well as bidirectional data transfer as mentioned below. We will discuss about them in detail towards the end of the post.

- Unidirectional Data Transfer
    - *Master-transmitter* to *Slave-receiver* (Case 1)
    - *Slave-transmitter* to *Master-receiver* (Case 2)
- Bidirectional Data Transfer
    - *Master* to *Slave* and *Slave* to *Master* (Case 3)

## Start/Stop Sequence

In order for the *Master* to start talking to the *Slave(s)*, it must notify the *Slave(s)* about it. This is done using a special start sequence. Remember a little while ago we discussed about I2C data validity – that the SDA should not change while the SCL is high? Well, it doesn't hold good for the start/stop sequence, which is why it makes them special sequences!

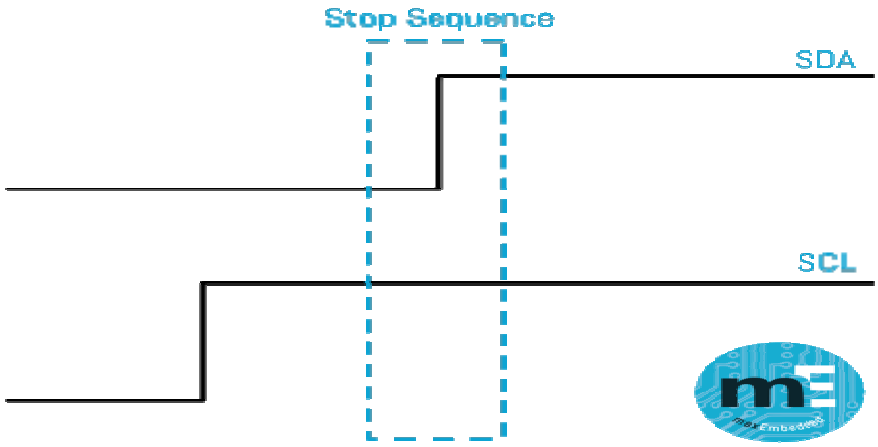
When the SCL is high and SDA goes from high to low (as shown in the following diagram), it marks the beginning of the transaction of *Master* with the *Slave(s)*.



I2C Start Sequence

And when the SDA goes from low to high while the SCL is still high (as shown in the following diagram), it marks the end of the transaction of that *Master* with the *Slave(s)*.





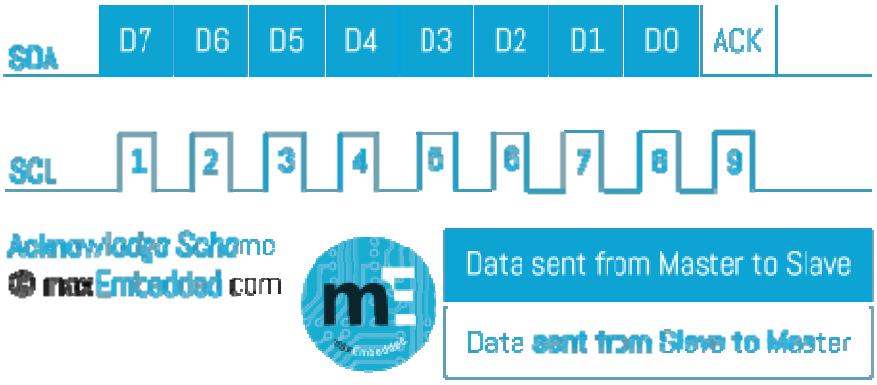

I2C Stop Sequence

**NOTE:** In between the start and stop sequences, the bus is busy and no other *Master(s)* (if any) should try to initiate a transfer.

## Acknowledge Scheme

As mentioned earlier, I2C transfers 8 bits (1 byte) of data at a time. After the transfer of each byte is complete, the receiver must acknowledge it. To acknowledge, the receiver sends an ACK bit back to the transmitter. Here's how it goes–

- The transmitter (could be either *Master* or *Slave*) transmits 1 byte of data (MSB first) to the receiver during 8 clock pulses of SCL, after which it releases the SDA line i.e. the SDA line becomes HIGH for the ACK clock pulse.
- The receiver (could be either *Master* or *Slave*, it depends) is obliged to generate an acknowledge after each byte sent by the transmitter by pulling the SDA line LOW for the ACK clock pulse (9th clock pulse) of SCL.
- So overall, there are 9 SCL clock pulses required to transmit a byte of data. This is shown in the diagram below with the assumption that *Master* is the transmitter.




I2C Acknowledgement Scheme (Assumption: Master is the transmitter)
Note: The legend shown at the bottom is only for SDA. SCL is always generated by the Master (whether transmitter or receiver).

So far so good. But what if the receiver does not (or could not) acknowledge the data sent to it? What happens then? Does the entire system break down?

Well, there are two cases to that situation.

### Case 1: Slave is at the receiver's end

Even in this case, there are two possible cases–

- **CASE 1A:** The *Slave-receiver* does not acknowledge the *Slave* address (hey wait, what's an address? We'll get to it shortly). In that case, it simply leaves the SDA line HIGH. Now the *Master-transmitter* either generates a Stop sequence or attempts a repeated Start sequence.
- **CASE 1B:** The *Slave-receiver* acknowledges the *Slave* address, but after some time it is unable to receive any data and leaves the SDA line HIGH during the ACK pulse. Even in this case, the *Master-transmitter* does the same – either generate a Stop sequence, or attempt a repeated Start sequence.

### Case 2: Master is at the receiver's end

Now this is a tricky situation. In this case, the *Master* is the one generating ACK, as well as responsible for generating Start/Stop sequence. Now how does that work out, especially when the transaction ends?

In this case, in order to signal the *Slave-transmitter* the end of data, the *Master-receiver* does NOT generate any ACK on the last byte clocked out of the *Slave-transmitter*. In this case, the *Slave-transmitter* must let go of the SDA line to allow *Master* to generate a Stop or a repeated Start sequence.

Makes sense? If it is confusing, hopefully it will make more sense when we actually program it for real.

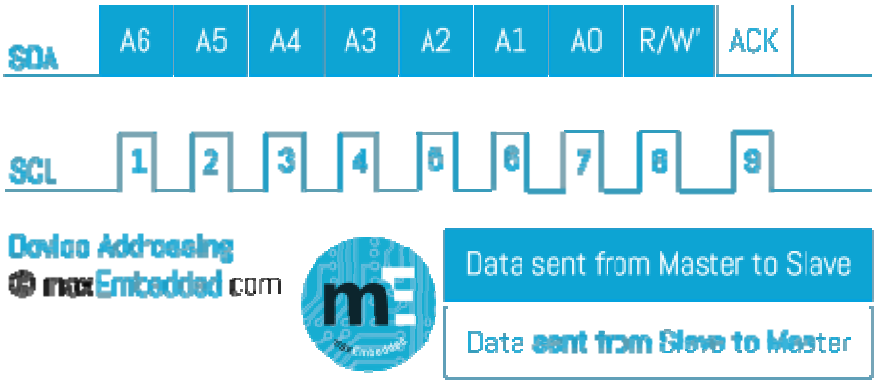
# I2C Device Addressing

Somewhere in the beginning of the tutorial, I mentioned that we can hook up a lot of devices to the I2C bus. But the *Master* can talk with only one of the *Slaves* at a time. Now how does that happen?

This is pretty similar to the situation inside a classroom. There is one teacher (*Master*) and a ridiculous number of students (*Slaves*). The teacher wants to ask a question to one of the students. How does she do that? Well, we do have a name, right? All the students have a unique name (address), and the teacher calls out the name first, right? *Hey Max, could you explain why do we need casex statements in Verilog?* Sounds familiar? Good old school days, eh? 

Well, this is exactly what happens in case of I2C bus transaction. Every device hooked up to the bus has a unique address. As per the I2C specifications, this address is either 7 bits long, or 10 bits long. 10-bit addresses are rare, and since I am lazy, I am gonna skip it for now.

When we have 7-bit address, we can have up to a maximum of $2^7 = 128$ devices hooked up to the I2C bus, with addresses 0 to 127. Now when the *Master* calls out to the *Slave(s)*, it still needs to send out 8 bits of data. In this case, the *Master* appends an extra Read/Write (R/W') bit to the 7 bits of address (note that W' means Write complemented). Thus, the R/W' bit is added to the LSB of the data byte. So now, the data sent looks somethings like this–




I2C Device Addressing

Note: The legend shown at the bottom is only for SDA. SCL is always generated by the Master (whether transmitter or receiver).

## Why should I bother about it?

Well, this is a question which I expect all the newbies to ask. Unfortunately most of them don't and then end up being frustrated. *Why the heck is my I2C not working?!*

Let's take a scenario. You have an external EEPROM which you want to interface using I2C with your processor. You know that the address of the EEPROM is `0x50`. You send this address to the bus expecting the EEPROM device to acknowledge. Does it acknowledge? *Heck, NO!*

So what's the problem here? Yes, you guessed it right (hopefully). You forgot about the R/W' bit! The address `0x50` is actually the 7-bit address (`0b1010000`).

Let's make it right. Say you wanna perform page write operation on the EEPROM device. This means that you wish to *write* to the device and hence the R/W' bit must be set to `0`. Why you ask? Because the write is complemented. For read, `R/W' = 1`, whereas for write, `R/W' = 0`. Makes sense?

So what should be the correct (modified) address?

```
0x50 &lt;&lt; 1 + 0x0 = 0xA0
0b1010000 &lt;&lt; 1 + 0b0 = 0b10100000
(Note: &lt;&lt; refers to left shift operation
```

If you wanna perform sequential read operation on the same EEPROM device, what would be your (modified) address?

```
0x50 &lt;&lt; 1 + 0x1 = 0xA1
0b1010000 &lt;&lt; 1 + 0b1 = 0b10100001
(Note: &lt;&lt; refers to left shift operation)
```

So, we can generalize that for write operations, the 8 bit address is even, whereas for read operations, the 8 bit address is odd.

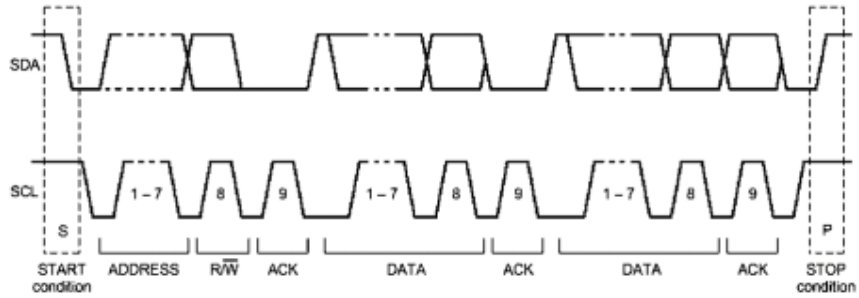
# I2C Data Transfer Protocol

Now that we are familiar with the I2C bus transactions and device addressing, let's see how to transfer data using the I2C protocol and have a 10,000 foot view of the entire bus transaction.

## Timing Diagram

Let's look at the timing diagram of an entire transaction – from start to stop!






Data Transfer Timing Diagram (Image source infoindustrielle.free.fr)

Let's analyze it first. Before we begin, we all know what these slanted lines mean, right? The slanted lines are a representation of slew rate of the device/bus/system. Ideally, whenever a signal changes its state (from high to low or vice-versa), it is supposed to do so immediately. But in real scenario, it is almost impossible for it to happen without a time lag. The slanted lines refer to these time lags, also known as slew.

Alright, back to where we were.

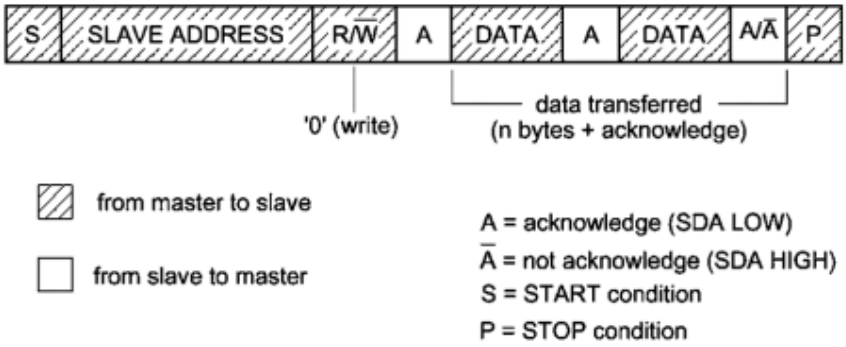
- The transaction starts with a START sequence.
- After the START sequence, the *Master* has to send the address of the *Slave* it wants to talk to. That's the 7-bit ADDRESS (MSB first) followed by R/W' bit determining whether you want to read from the device or write into it.
- The *Slave* responds by acknowledging the address. Yay! If it doesn't send the ACK, we know what the *Master* does, right?
- Once the *Slave* acknowledges the address, it means that it is now ready to send/receive data to/from the *Master*. Thus begins the data transfer. The DATA is always of 8 bits (MSB first), and the receiver has to send the ACK signal after each byte of data received.
- When the transaction is over, the *Master* ends it by generating a STOP sequence. Alternatively, *Master* could also begin with a repeated START.

There are three possible cases of data transfer–

- Case 1: *Master-transmitter* to *Slave-receiver*
- Case 2: *Slave-transmitter* to *Master-receiver*
- Case 3: Bi-directional (R/W) in same data transfer

# Case 1: Master (Transmitter) to Slave (Receiver) Data Transfer

Let's have a look at the entire transaction first and then analyze it.




Master to Slave Data Transfer (Image source infoindustrielle.free.fr)

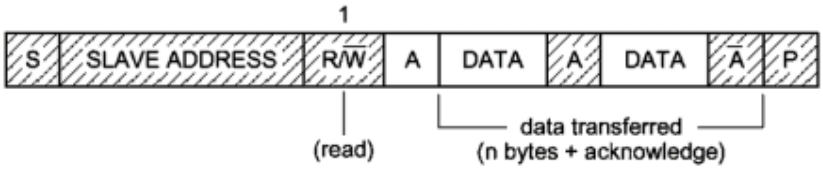
- The *Master* sends the START sequence to begin the transaction.
- It is followed by *Master* sending 7-bit *Slave* address and the R/W’ bit set to zero. We set it to zero because the *Master* is *writing* to the *Slave*.
- The *Slave* acknowledges by pulling the ACK bit low.
- Once the *Slave* acknowledges the address, *Master* can now send data to the *Slave* byte-by-byte. The *Slave has to* send the ACK bit after every byte it receives.
- This goes on till *Slave* can no longer receive data and does NOT send the ACK bit.
- This is when the *Master* realizes that the *Slave* has gone crazy (not accepting anymore) and then STOPs the transaction (or reSTART).

We see that the data transfer never changes its direction. Data always flows from *Master* to *Slave*, which makes the setup quite easy.

An example of this case would be like performing page write operations on a EEPROM chip.

# Case 2: Slave (Transmitter) to Master (Receiver) Data Transfer

Let’s look at the entire transaction again.




Slave to Master Data Transfer (Image source infoindustrielle.free.fr)

- The *Master* sends the START sequence, followed by the 7-bit *Slave* address and the R/W’ bit set to 1.
- We set R/W’ bit to 1 because the *Master* is *reading* from the *Slave*.
- The *Slave* acknowledges the address, thus ready to send data now.
- *Slave* keeps on sending data to the *Master*, and the *Master* keeps on sending ACK to the *Slave* after each byte until it can no longer accept any more data.

- When the *Master* feels like ending the transaction, it *does not* send the ACK, thus ending with the STOP sequence.

An example of this case could be an Analog to Digital Converter (ADC) sending data to the microcontroller continuously. The microcontroller accepts data as long as it wants to, after which it stops/finishes execution.

# Case 3: Bi-directional Read and Write in same Data Transfer

Once again, let's look at the entire transaction first!




Bi-directional Data Transfer (Image source infoindustrielle.free.fr)

- The *Master* sends out the START sequence, followed by the 7-bit *Slave* address and the R/W' bit.
- The *Slave* acknowledges the address.
- Depending upon the value of the R/W' bit, read/write operations are performed (like the above two cases).
- Whatever the case it may be, it always ends with the receiver *not* sending the ACK.
- Until now, in the previous two cases, we have seen that the *Master* would close the connection. But in this case, the *Master* attempts a repeated START.
- And the entire process repeats again, until the *Master* decides to STOP.

As we can see, a change of direction of data transfer might happen depending upon the R/W' bits in the entire transaction.

An example of this case could be performing sequential read from a EEPROM chip. It is bi-directional because the CPU first *writes* the address from where it would like to start reading, followed by *reading* from the device. It is like, unless you tell the device from where you would like to start reading, how would it start sending you the data?

# Clock Stretching

So far so good. Now let's look at a very plausible complication. Suppose *Master* is reading data from the *Slave*. Everything goes on good as long as the *Slave* returns the data. What if… what if the *Slave* is not just ready yet? This is not an issue with devices like ADC or EEPROM, but with devices like a microcontroller. What if the *Slave* is a microcontroller, and the *Master* requests for a data which is not there in its cache. This would require the microcontroller to perform context switching, force it to search for it in the RAM, store it back in cache and then send it to the *Master*. This could (and definitiely would) take a much longer time than the clock pulses of the SCL, and everything would just go wrong!

Fortunately, there is a way, called *Clock Stretching*. The *Slave* is allowed to hold the clock line low until it is ready to give out the result. The *Master* must wait for the clock to go back to high before continuing with its work.

This is the only instance in the entire I2C protocol where the *Slave* is drives the clock line SCL. In many processors and microcontrollers, the low level hardware does this for us, so that we don't have to worry about it while writing the code.

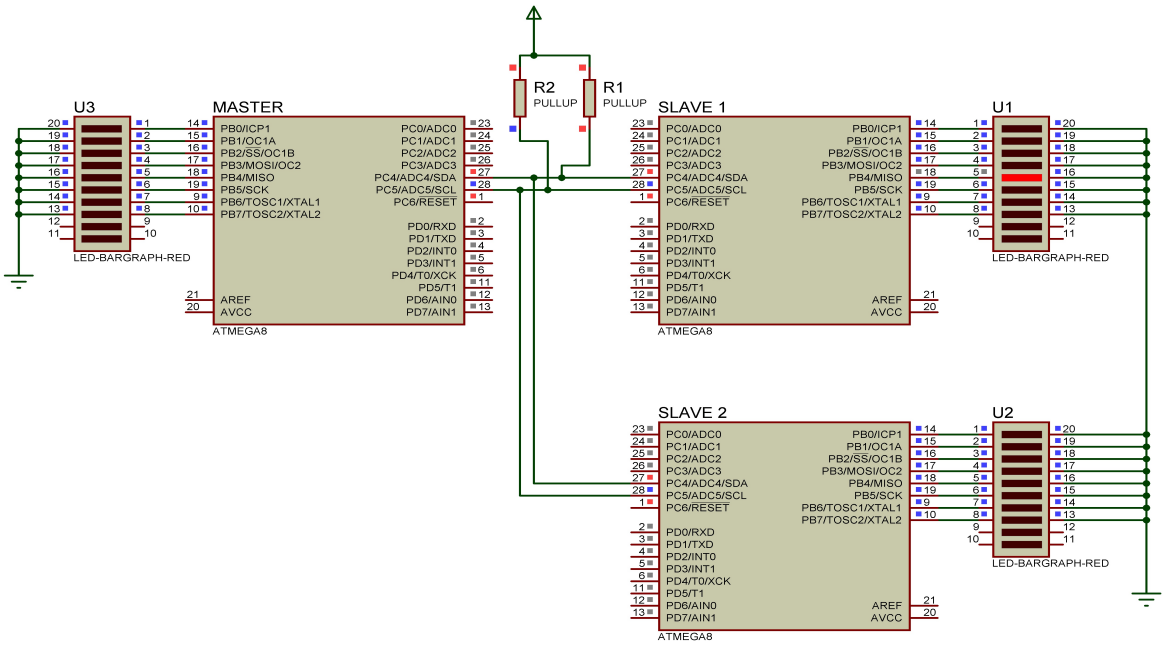
# Why I2C?

Now that we are almost done with the basics of I2C communications, let's take a moment to jot down some advantages of I2C.

- I2C requires least number of pins (just two pins) to perform serial data transfer.
- The receiver always sends feedback to the transmitter (ACK) conveying a successful transmission, which leads to higher noise immunity as well.
- Even though it has a slow standard speed of 100 kHz, modern I2C specifications support up to 3.4 MHz clock speed.

# Summary

- I2C is an 8-bit bidirectional synchronous serial communication protocol requiring only two wires for operation.
- The I2C bus consists of two open-drain lines – SDA (data) and SCL (clock).
- Several devices, being either *Master* or *Slave*, can be connected to the bus. The *Master* device must initiate the transfer and drive the clock line (SCL).
- I2C supports the standard speed of 100 kbps, up to a maximum speed of 3.4 Mbps.
- *Master* must generate unique Start and Stop conditions in order to mark the beginning and end of a transaction.
- The receiver must send the ACK bit after every byte that it receives, failing which the *Master* may either Stop the transaction or attempt a repeated Start.
- Every device connected to the I2C bus has either 7-bit or 10-bit address. An additional R/W' bit is added to the address by the *Master* to determine whether it wants to read or write from/to the device.
- Data transfer can be unidirectional (*Master* to *Slave* OR vice-versa) or bidirectional.
- *Slave* can hold the clock line low until it is ready with the result to be sent to the *Master*, called Clock Stretching




```
//1. I2C MASTER PROGRAM

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

#define SLAVE1_ADD 0x02
#define SLAVE2_ADD 0x04

/*
    Note 1: To send any signal, TWINT flag bit should be cleared by providing 1 to it and TWEN
    bit should be enabled.
    Note 2: Wait till TWINT flag bit goes high which indicates completion of signal transmission
*/


//STATUS Code

#define START 0x08
#define RESTART 0x10
#define ADDRESS_WRITE_ACK 0x18
#define ADDRESS_READ_ACK 0x40
#define DATA_WRITE_ACK 0x28
#define DATA_READ_ACK 0x58

#define READ 1
#define WRITE 0


//Error Function

void error(char i)
{
    DDRB = 255;
    PORTB |= i;
}
```

```
//I2C Initialisation

void twi_init_master()
{
    TWBR = 0xFF; //Setting Minimum Frequency
    TWSR|=(1<<TWPS0); //Prescalar: 4
}

//I2C Start Condition

void twi_start()
{
    TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //Clearing Flag, Sending Start Signal, Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Waiting till transmission completes
    if((TWSR & 0xF8)!= START)  //Masking upper 5 bits of TWSR as it also has prescalar bits
    //Checking Acknowledgment
        error(1); //Making 1st LED ON to indicate Start Signal error
}

//I2c Restart Condition

void twi_restart()
{
    TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //Clearing Flag, Sending Start Signal, Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Waiting till transmission completes
    if((TWSR & 0xF8)!= RESTART)  //Masking upper 5 bits of TWSR as it also has prescalar bits
    //Checking Acknowledgment
        error(2); //Making 1st LED ON to indicate Start Signal error
}

//I2C Stop Condition

void twi_stop()
{
    TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN); //Clearing Flag bit, Sending Stop Signal, Enabling
    TWI
    while(!(TWCR & (1<<TWINT))); //Waiting till transmission completes
}

//I2C for Addressing and write Signal

void twi_write_address(unsigned char address)
{
    TWDR = (address<<1) + WRITE;
    TWCR = (1<<TWINT)|(1<<TWEN); //Clearing Flag bit, Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till tranmission completes
    if((TWSR & 0xF8)!=ADDRESS_WRITE_ACK) //Checking status after acknowledgment
        error(4);
}

//I2C for Addressing and Read Signal

void twi_read_address(unsigned char address)
```

```
{
    TWDR = address+READ;
    TWCR = (1<<TWINT)|(1<<TWEN); //Clearing Flag bit, Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till tranmission completes
    if((TWSR & 0xF8)!=ADDRESS_READ_ACK) //Checking status after acknowledgment
        error(8);
}

//I2C for Writing Data

void twi_write_data(unsigned char data)
{
    TWDR = data; //Loading data
    TWCR = (1<<TWINT)|(1<<TWEN); //Clearing Flag bit, Enable TWI
    while(!(TWCR & (1<<TWINT))); //Wait till transmission completes
    if((TWSR & 0xF8)!=DATA_WRITE_ACK) //Checking status after acknowledgment
        error(16);
}

//I2C for Reading Data

unsigned char twi_read_data()
{
    TWCR = (1<<TWINT)|(1<<TWEN); //Clearing flag bit, Enable TWI
    while(!(TWCR & (1<<TWINT))); //Wait Till Transmission completes
    if((TWSR & 0xF8)!=DATA_READ_ACK) //Checking status after acknowledgment
        error(32);
    return TWDR; //Returning received data
}

void main()
{
    int i;
    twi_init_master(); //Initializing TWI configured as Master
    twi_start(); //Sending Start Signal
//  twi_write_address(SLAVE1_ADD); //Sending Write signal to slave
    while(1)
    {
        twi_write_address(SLAVE1_ADD); //Sending Write signal to slave
        for(i=1;i<=128;i*=2)
        {
            twi_write_data(i); //Sending data
            _delay_ms(300);
        }
        twi_restart();
        twi_write_address(SLAVE2_ADD); //Sending Write signal to slave
        for(i=128;i>=1;i/=2)
        {
            twi_write_data(i);
            _delay_ms(300);
        }
        twi_restart();
    }
}
```

```
//2. SLAVE 1 PROGRAM
#include <avr/io.h>
#include <util/delay.h>
#define SLAVE_ADDRESS 0x02
//Status Code
#define ADDRESS_ACK 0x40
#define DATA_ACK 0x80
#define STOP 0xA0

//Error Function
void error(unsigned char i)
{
    DDRB = 255;
    PORTB |= i;
}

//I2C Initialization
void twi_init_slave(unsigned char address)
{
    TWAR = (address<<1); //Setting slave address value by loading it in TWI Address Register
    (TWAR)
}

//I2C Address Ack
void twi_address_ack()
{
    TWCR = (1<<TWINT)|(1<<TWEA)|(1<<TWEN); //Clearing Flag bit, Enabling Acknoledgment,
    Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till transmission completes
    if((TWSR & 0xF8)!=ADDRESS_ACK) //Checking Status
        error(1);
}

//I2C Data Read
unsigned char twi_read_data()
{
    TWCR = (1<<TWINT)|(1<<TWEA)|(1<<TWEN); //Clearing Flag bit, Enabling Acknoledgment,
    Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till transmission completes
    if(((TWSR & 0xF8)!=DATA_ACK))//||((TWSR & 0xF8)!=STOP)) //Checking Status
        error(2);
    return TWDR;
}

void main()
{
    DDRB = 255;
    twi_init_slave(SLAVE_ADDRESS); //Initialising I2C in Slave Mode
    twi_address_ack(); //Sending address acknowledgment
    while(1)
    {
        PORTB = twi_read_data();
    }
}
```

```
//3. SLAVE 2 PROGRAM
#include <avr/io.h>
#include <util/delay.h>
#define SLAVE_ADDRESS 0x04
//Status Code
#define ADDRESS_ACK 0x60
#define DATA_ACK 0x80
#define STOP 0xA0

//Error Function
void error(unsigned char i)
{
    DDRB = 255;
    PORTB |= i;
}

//I2C Initialization
void twi_init_slave(unsigned char address)
{
    TWAR = (address<<1); //Setting slave address value by loading it in TWI Address Register
    (TWAR)
}

//I2C Address Ack
void twi_address_ack()
{
    TWCR = (1<<TWINT)|(1<<TWEA)|(1<<TWEN); //Clearing Flag bit, Enabling Acknoledgment,
    Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till transmission completes
    if((TWSR & 0xF8)!=ADDRESS_ACK) //Checking Status
        error(1);
}

//I2C Data Read
unsigned char twi_read_data()
{
    TWCR = (1<<TWINT)|(1<<TWEA)|(1<<TWEN); //Clearing Flag bit, Enabling Acknoledgment,
    Enabling TWI
    while(!(TWCR & (1<<TWINT))); //Wait till transmission completes
    if(((TWSR & 0xF8)!=DATA_ACK)||((TWSR & 0xF8)!=STOP)) //Checking Status
        error(2);
    return TWDR;
}

void main()
{
    DDRB = 255;
    twi_init_slave(SLAVE_ADDRESS); //Initialising I2C in Slave Mode
    twi_address_ack(); //Sending address acknowledgment
    while(1)
    {
        PORTB = twi_read_data();
    }
}
```