2009

Beginners Guide To

Embedded Systems & Robotics

Sourabh Sankule
www.sourabh.sankule.com

June 2009

Contribution

- Arpit Mathur
- Abhijeet Gupta
- Ankur Agrawal
- Tanmay Gangwani

Acknowledgement

- Robocon Team, IIT Kanpur
- Robotics Club, IIT Kanpur
- Electronics Club, IIT Kanpur
- Centre for Mechatronics, IIT Kanpur

Contents

Section A: Digital Electronics

Section B: Embedded Systems

Section A

Digital Electronics

Chapter 1 Basics

1.1 What is a Digital system?

In most general terms, this system's behavior is sufficiently explained by using only two of its states can be Voltage(more than x volts or less?),distance covered(more than 2.5 km or less?],true-false or weight of an elephant(will my weighing machine withstand it?))

Note that although in every case, the all the intermediate states **ARE POSSIBLE AND DO EXIST,** our point of interest are such that we don't require their explicit description**.** In electronic systems we mostly deal with Voltage levels as digital entities.

1.2 Assigning States

There is no specific fixed definition of logic levels in electronics. Most commonly used level designation is the one used in CMOS and TTL (transistor transistor logic) families:

Logic high –> designated as '1'

Logic low –> designated as '0'

Where high and low are actually 'higher' and 'lower' with respect to a reference voltage level (ideally taken as 2.5V)

GOOGLY: Why assign '0'and '1' and not 'a 'and 'b', 'x' and 'y ,'cat 'and 'dog'?
ANS: Computational ease!

1.3 Number Systems in digital electronics

1. **Binary:** Only '0' and '1'.
2. **Hexadecimal:** 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Dec	Hex	Bin
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

1.4 Types of Digital Circuits

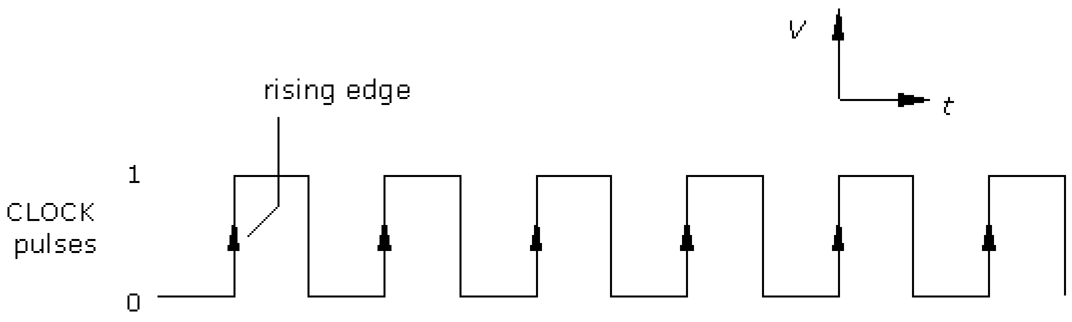
Combinatorial Circuits: In these circuits, the past states are immaterial and the output depends only upon the present state. Example logic gates

Sequential circuits: In these circuits, the next state is completely determined by the past states. Hence these follow a predictable structure and essentially require a timing device. Ex. counters, flip flops.

1.5 **Clock:** Building block of a sequential circuit

A clock is simply alternate high and low states of voltage with time i.e. essentially a square wave. Important terms related to clock are its duty cycle and its frequency:

Duty cycle: It is the ratio of T_h and T_h+T_l $D = \frac{Th}{Tl+Th}$



1.6 **Logic Gates:** Building block of a combinatorial circuitry

These are essentially combinatorial circuits used to implement logical Boolean operations like AND, NAND, OR, XOR and NOT. NOT and NAND are called universal gates as any other gate can be formed using either of them!

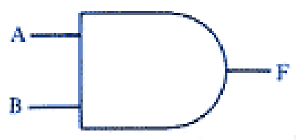
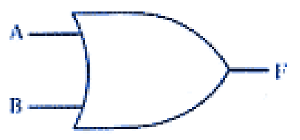
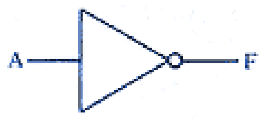
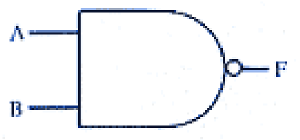
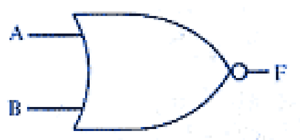
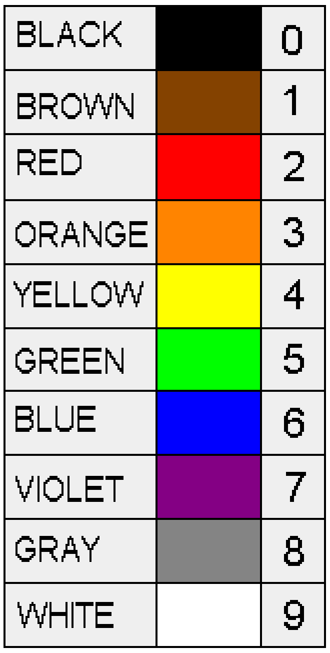
Name	Graphic Symbol	Algebraic Function	Truth Table
AND		$F = A \cdot B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = (\overline{AB})$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (\overline{A + B})$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

Figure 2: Table of Logic Gates

1.7 Practical Circuiting Elements

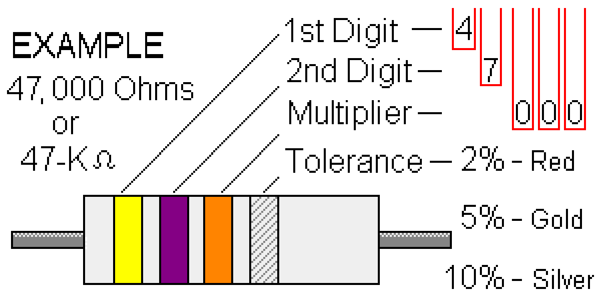
1.7.1 Resistor:

A color scheme is followed to give the specifications of a resistor. The table for color code is shown below:



BLACK	⬛	0
BROWN	🟫	1
RED	🟥	2
ORANGE	🟧	3
YELLOW	🟨	4
GREEN	🟩	5
BLUE	🟦	6
VIOLET	🟪	7
GRAY	◼	8
WHITE	⬜	9

Figure 3: Table of Resistance



The 1st two bands specify the 2 digits of the resistor value whereas the 3rd band specifies the multiplier in terms of the power to which 10 is raised and multiplied to the 2 digits.

The tolerance tells the possible % variation of the resistor value about the value indicated by bands.

1.7.2 Capacitor:

The 2 types of capacitors we frequently use in circuits are ceramic and electrolytic capacitors. While ceramic capacitors do not have a fixed polarity; electrolytic capacitors should be connected in their specified polarities only else they might blow off! This polarity is usually provided on the side of the capacitors ‘corresponding leg.

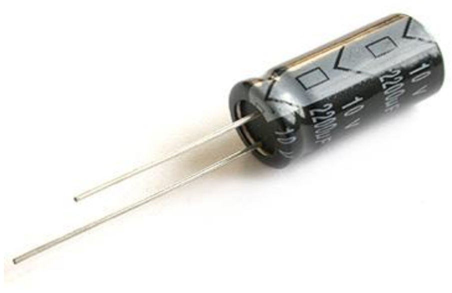


Figure 5: Electrolytic cap with –ve polarity leg seen

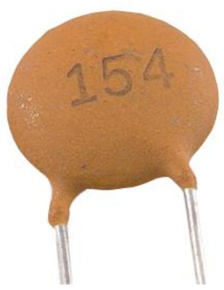


Figure 4: Ceramic cap with value 15x104 pF

1.7.3 Breadboard:

This is the base used for setting up the circuit. This has embedded metal strips in it that form a grid of connections inside its body. This allows us to take multiple connections from a single point without any need of soldering/disordering as in PCBs. It is always a good habit to test the circuit on breadboard before making it on a PCB.

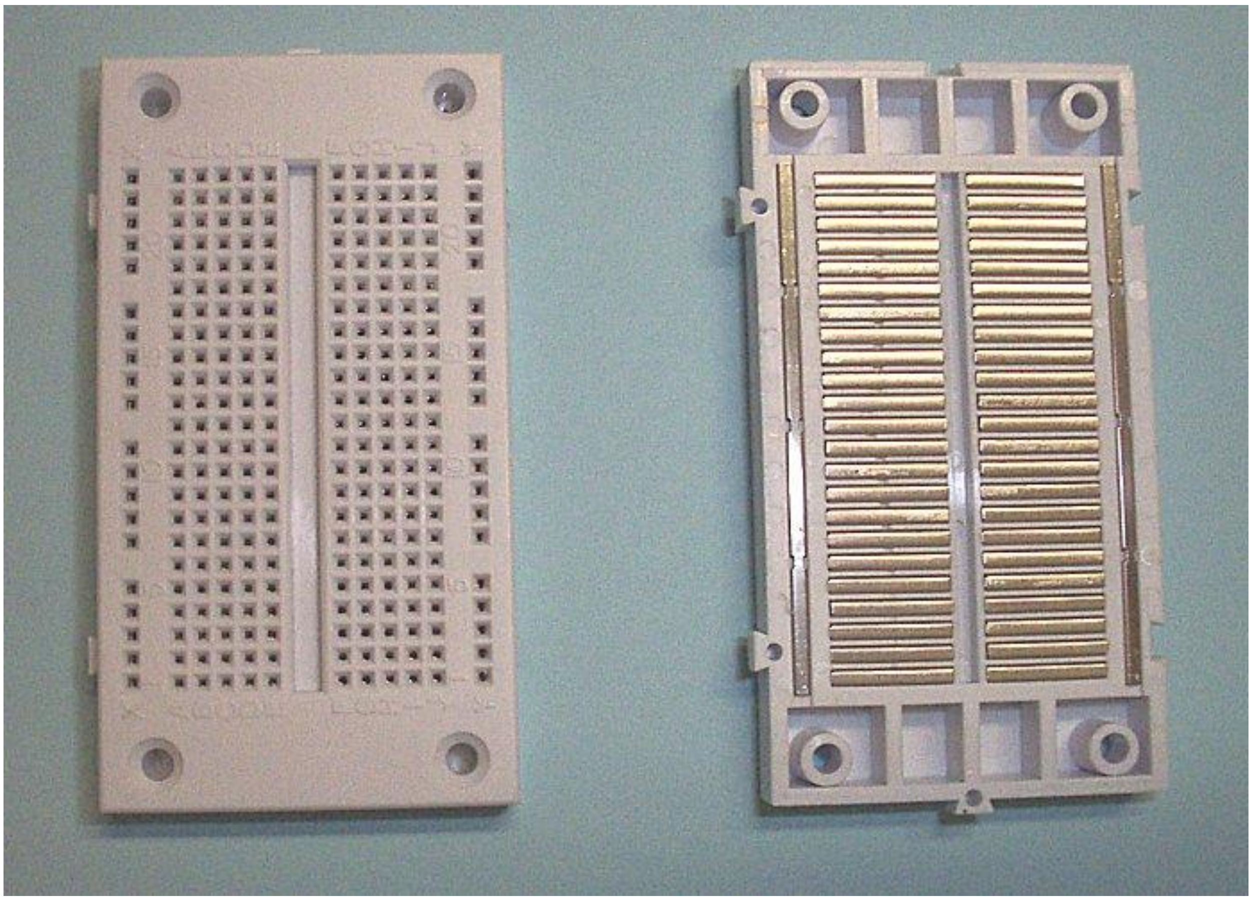


Figure 6: Top view showing the connecting holes. Bottom view shows the contact metal strips

1.7.4 Integrated Circuits (IC)

ICs or Integrated Circuits are packaged circuits designed for some fixed purpose. An IC has its fixed IC name/number that can be used to get catalog of its functions and pin configuration. ICs come in various sizes and packages depending upon the purpose.

NOTE: Numbering scheme of IC pins will be explained in the lab session. Different ICs may have different number of pins.

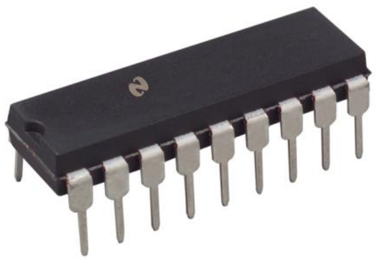


Figure 7: IC

1.7.5 LED

LED (Light Emitting Diode) is frequently used to display the outputs at various stages of the circuit. It is essentially a Diode with the energy released in the form of photons due to electron transitions falling in the visible region. Hence normal diode properties apply to it.

It glows only in fwd bias mode i.e. with p junction connected to +ve voltage and n junction to negative.

Diodes are essentially low power devices. The current through the LED should be less than 20mA.Hence always put a 220 ohm resistor in series with the LED.

Never forget that LEDs consume a significant amount of power of the outputs of the ICs (CMOS based).Hence it is advisable to only use them for checking the voltage level (high or low) and then remove them.

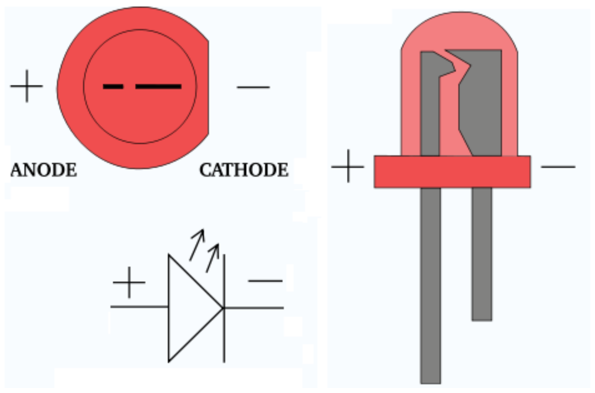


Figure 8: LEDs

Chapter 2 Some Integrated Circuits and Implementation

2.1 555

555 is an IC used to generate a clock .The two attributes of a clock are

- Frequency
- Duty cycle.

Both of these can be changed using this IC, however the duty cycle is always <50%.

There are two modes in which 555 can run.

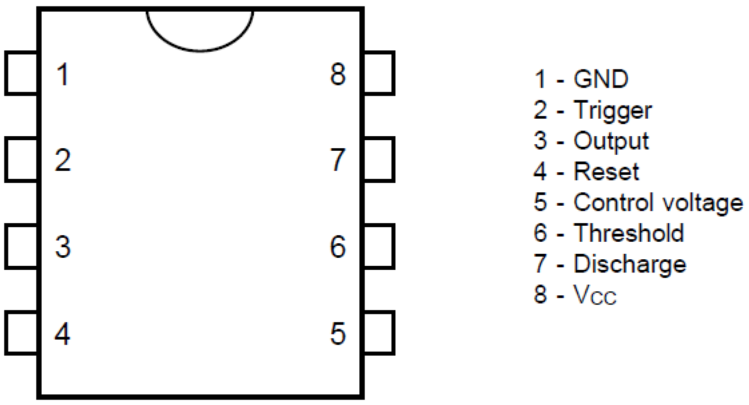


Figure 9: Pin Configuration of 555

2.1.1 Monostable mode

As the name suggests; in this mode the output is stable in only one (mono) state i.e. 'off' state. Thus it can stay only for a finite time, if **triggered,** to the other state i.e. 'on' state. This time can be set choosing appropriate values of resistances in the formula:

T = 1.1 x R1 x C1

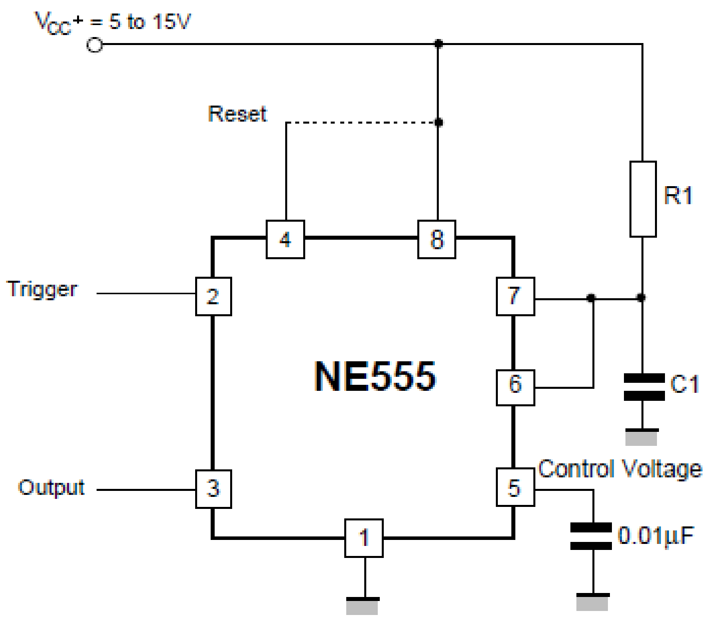


Figure 10: 555 in monostable mode

2.1.2 Astable mode

In this mode; the output is stable neither in 'high' state nor in 'low ' state. Hence it oscillates from one state to another giving us a square wave or clock. We can set the clock frequency and Duty cycle D by the formulae:

$$F = \frac{1.44}{(R1+2R2)C1} \qquad D = \frac{(R1+R2)}{(R1+2R2)}$$

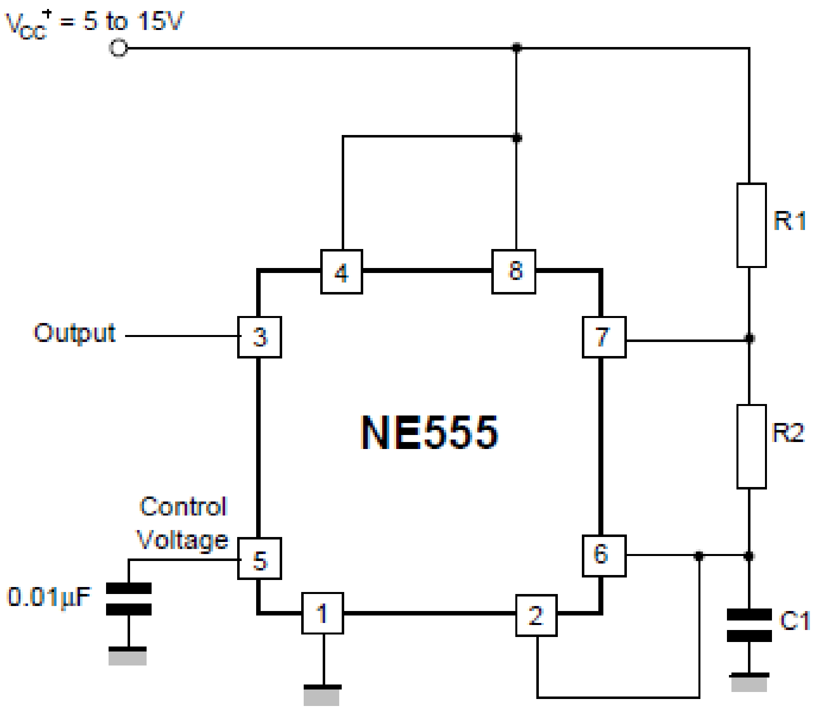


Figure 11: 555 in astable mode

NOTE: Capacitor C2 is just to filter the noise and its value can be suitably chosen to be 0.01µF. It can also be neglected.

2.2 4029 counter

With the clock made, we are ready to count the number of pulses passed into the circuit. Note that any kind of counting requires a **memory** (you got to know that you have just counted '3' to go to '4'!). Hence 4029 can also be used as a memory element that remembers its immediate previous state.

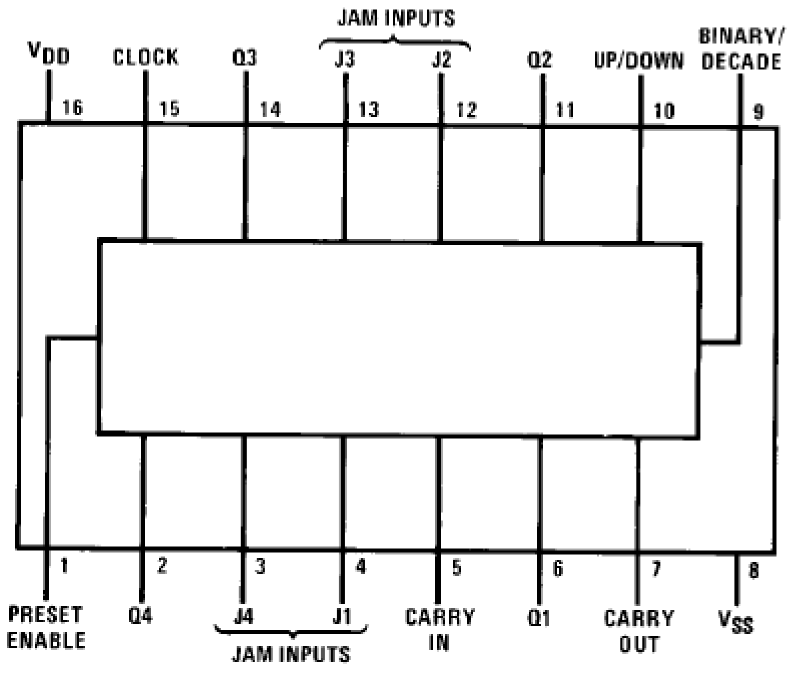


Figure 12: 4029 pin configuration

2.2.1 Pin Description

PIN No.	Name	Pin function	Connection reqd.
1	Parallel load	If given high; loads the value of Parallel bit into the o/p bits	Gnd
2	Output Bit 3	Most significant bit of o/p	To pin 6 of 7447
3	Parallel input bit 3	Most significant bit of parallel i/p	Input
4	Parallel input bit 0	Least significant bit of parallel i/p	Input
5	Clock enable bar	Low on this pin enables counting as per the clock received	Gnd
6	Output Bit 0	Least significant bit of parallel o/p	To pin 7 of 7447
7	TC bar	Output bit that gives a low when the count is complete. Can be used to signal the end of counting.	None if you don't want to use it
8	Gnd	Needed for powering	Gnd
9	Binary/Hex bar	To choose b/w binary and hexadecimal modes	low for count 0-15 high for count 0-9
10	Up/Down bar count	To choose b/w up counting and down counting modes	Low for down count High for up count
11	Output bit 1	2nd bit of o/p	To pin 1 of 7447
12	Parallel input bit 1	2nd bit of i/p	Input
13	Parallel input bit 2	3rd bit of i/p	Input
14	Output bit 2	3rd bit of o/p	To pin 2 of 7447
15	Clock pulse	Clock pulse is given here	Clock from 555
16	V_{dd}	Needed for powering	+5 V

2.3 7447: BCD to 7 segment display decoder

For displaying the number in the counter output on a seven segment display (i.e. 7 LEDs making up a figure of '8' as in a general calculator. See fig.) we need to decode the 4 bits and match them to the 7 pins for lighting the LEDs corresponding to the number. This work is done by 7447.

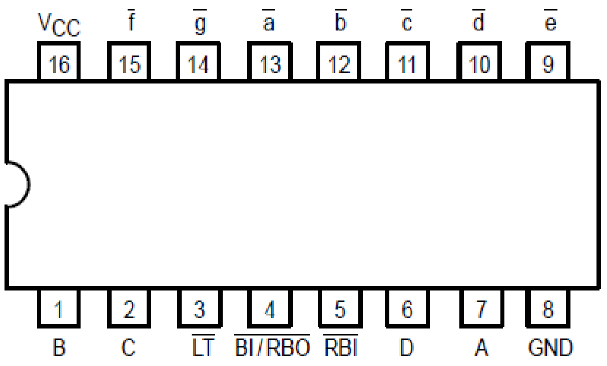
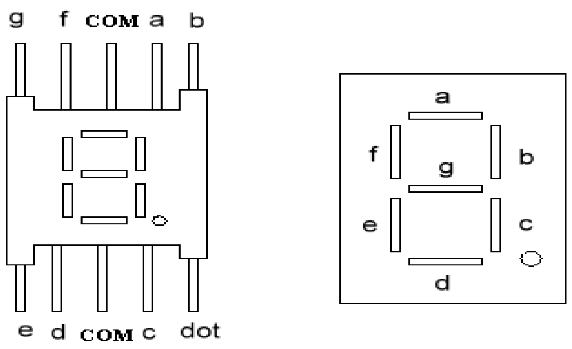


Figure 13: 7447 pin configuration



Seven-Segment Display

2.3.1 Pin Description

PIN no.	Name	Function	Connection reqd.
1	i/p B	2nd bit(O1) of 4029's o/p	To O1 of 4029
2	i/p C	3rd bit(O2) of 4029's o/p	To O2 of 4029
3	Lamp Test bar	Used to check that all LEDs of 7 seg are working.	High for normal fn Low to glow all LEDs
4	BI /RBI	Kept high to allow normal function	Kept high
5	RBI	Blanks '0' from being displayed	Kept high
6	i/p D	Most significant bit(O3) of 4029's o/p	To O2 of 4029
7	i/p A	3rd bit(O2) of 4029's o/p	To O2 of 4029
8	Gnd	For power	Connected to gnd
9-15	a-g as per the fig	The o/p pins to 7segment display	To 7 seg display
16	Vcc	For power	Connected to +5 V

NOTE:

- The COM pins are to be connected to Vcc via 220 ohm resistor. Why resistor is required??
- The dot pin is just for display of decimal point and essentially only makes the upper and lower sides distinguishable from each other for a single display. Without the asymmetry produced by dot how will we be able to see which side is upper and which is lower?

2.4 LDR (Light Dependent Resistor)

Light Dependent Resistor (LDR) or photo resistor is a device that acts like a resistance and its resistance varies with the intensity of light incident on it. In this device, if photons of sufficient energy fall on it, the resistance drops drastically as the electrons in the semiconductor are able to jump from the valence band to the conduction band and are available for conduction. The LDRs used are mostly responsive to visible light. The resistance might drop from as high as 1MΩ in the dark to 1 k Ω in bright light.

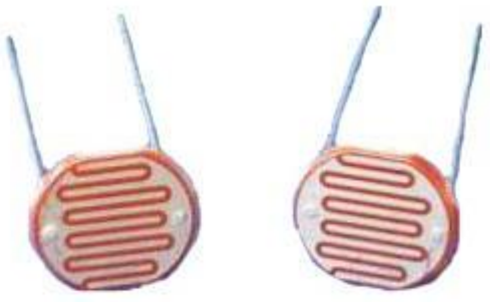


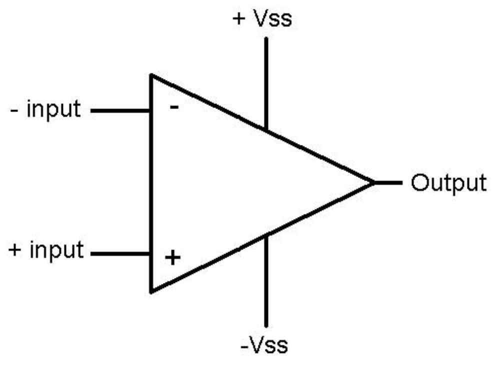
Figure 14: LDRs. The coiled portion is responsive to light

2.5 Operational Amplifier (Opamp)

Opamp is a very important device used in everyday electronics .It is essentially a differential amplifier with a very high gain of the order of 10^5!By differential amplifier I mean that it amplifies the difference of 2 signals and gives the output.

Opamp equation:

$$V_{out} = A\,(V_+ - V_-) \qquad \text{where A is the gain of the order } 10^5.$$



Ironically, this high gain in open loop makes it impossible to use it as a general purpose differential amplifier directly.

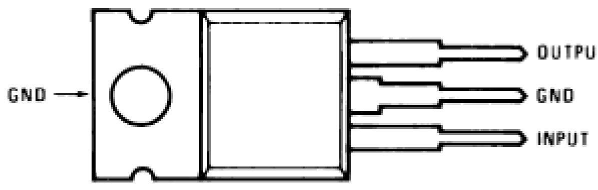
<u>GOOGLY</u>: If $(V_+ - V_-) = 0.005V$; Vss = 12V what will be the output??

2.5.1 Opamp as a comparator

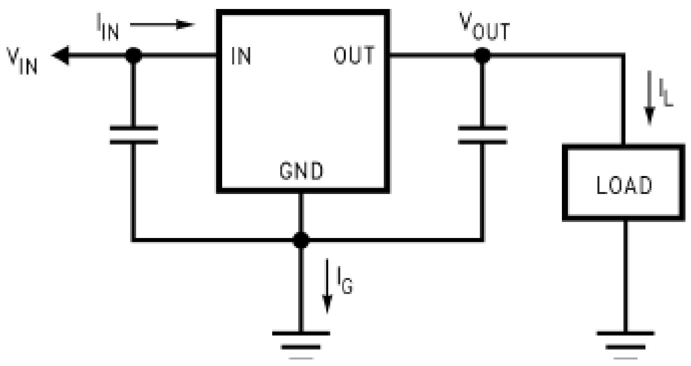
Simplest use of Opamp is as a comparator. It can be used to convert an analog signal to a digital signal defined by a fixed threshold. Set V- as the threshold voltage say 2.5 V and apply the analog signal to be digitized at V+ .What will be the output? Well if you have worked out the googly; this should be a piece of cake!

2.6 7805 Voltage Regulator

7805 voltage regulator is used to get +5 V output out of a higher voltage supply (7.5V-20V).We use adapter's supply to generate +5V here. Connect the gnd and +12V of adapter to the pins as shown and get +5V directly as an output out of the 3rd pin. Current up to 0.5 A can be obtained from this regulator without any significant fall in voltage level.



NOTE: Use 2 capacitors of value say 0.1µF to filter the noise in the input and output of regulator's supply as shown. Also put a capacitor of 10 – 100 uF at output to take care of the low frequency dip in the voltages.



Section B

Embedded Systems

Chapter 3 Introduction to Microcontrollers

You must be knowing about Digital Integrated Circuits (ICs) right ? For example:

- 7404: Hex Inverter
- 7408: Quad 2-input AND gate
- 7410: Triple 3-input NAND Gate
- 7432: Quad 2-input OR Gate
- 7457: 60:1 Frequency divider

There are AND, XOR, NAND, NOR, OR logic gate ICs, Counters, Timers, Seven Segment Display Drivers and much more. Just check out 7400 Series and 4000 Series of Integrated Circuits.

Now let's take Quad 2 input AND gate IC. It has 4 AND gates, each having 2 pins for input and 1 pin for output. The truth table or the function table of each gate is fixed. This is as follows,

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

Similarly all the Integrated circuits have their function tables and input and output pins fixed. You cannot change the function and no input pin act as output and vice versa. So whenever you want to design some circuit you first have to get the output as a function of inputs and then design it using gates or whatever the requirement is.

So once a circuit is built we cannot change its function ! Even if you want to make some changes again you have to consider all the gates and components involved. Now if you are designing any circuit which involves change of the function table every now and then you are in trouble ! For example if I want to design an Autonomous Robot which should perform various tasks and I don't just want to fix the task. Suppose I make it to move in a path then I want to change the path ! How to do that ?

Here comes the use of Microcontrollers ! Now if I give you an Integrated Circuit with 20 pins and tell you that you can make any pin as output or input also you can change the function table by programming the IC using your computer ! Then your reactions will be wow ! that's nice :-) That's what the most basic function of a microcontroller is. It has set of pins called as PORT and you can make any pin either as output or input. After configuring pins you can program it to perform according to any function table you want. You can change the configuration or the function table as many times you wants.

There are many Semiconductor Companies which manufactures microcontrollers. Some of them are:

- Intel
- Atmel
- Microchip
- Motorola

We will discuss about Atmel Microcontrollers commonly known as AVR in this section.

Question: How a microcontroller works !

Answer: Well I cannot go into lot of details about the working because it is a vast topic in itself. I can just give an overview.

Microcontroller consists of an Microprocessor (CPU that is Central processing Unit) which is interfaced to RAM (Random Access Memory) and Flash Memory (one your pen drive has !). You feed your program in the Flash Memory on the microcontroller. Now when you turn on the microcontroller, CPU accesses the instruction from RAM which access your code from Flash. It sets the configuration of pins and start performing according to your program.

Question: How to make the code ?

Answer: You basically write the program on your computer in any of the high level languages like C, C++, JAVA etc. Then you compile the code to generate the machine file. Now you will ask what this machine file is ? All the machines understand only one language, 0 & 1 that is on and off. Now this 0 & 1 both corresponds to 2 different voltage levels for example 0 volt for 0 logic and +5 volt for 1 logic. Actually the code has to be written in this 0, 1 language and then saved in the memory of the microcontroller. But this will be very difficult for us ! So we write the code in the language we understand (C) and then compile and make the machine file (.hex). After we make this machine file we feed this to the memory of the microcontroller.

Question: How to feed the code in the flash of Microcontroller ?

Answer: Assuming you have the machine file (.hex) ready and now you want to feed that to the flash of the microcontroller. Basically you want to make communication between your computer and microcontroller. Now computer has many communication ports such as Serial Port, Parallel Port and USB (Universal Serial Bus).

Let's take Serial Port, it has its own definition that is voltage level to define 0 & 1 (yeah all the data communication is a just collection of 0 & 1) Serial Port's protocol is called as UART (Universal Asynchronous Receiver & Transmitter) Its voltage levels are : +12 volt for 0 logic and -12 volt for 1 logic.

Now the voltage levels of our microcontroller are based on CMOS (Complementary Metal Oxide Semiconductor) technology which has 0 volt for 0 logic and +5 volt for 1 logic.

Two different machines with 2 different ways to define 0 & 1 and we want to exchange information between them. Consider microcontroller as a French and Computer's Serial Port as an Indian person (obviously no common language in between !) If they want to exchange information they basically need a mediator who knows both the language. He will listen one person and then translate to other person. Similarly we need a circuit which converts CMOS (microcontroller) to UART (serial port) and vice versa. This circuit is called as programmer. Using this circuit we can connect computer to the microcontroller and feed the machine file to the flash.

3.1 Compiler / IDE (Integrated Development Environment)

Atmel Microcontrollers are very famous as they are very easy to use. There are many development tools available for them. First of all we need an easy IDE for developing code. I suggest beginners to use CVAVR (Code Vision AVR) Evaluation version is available for free download from the website. It has limitation of code size. It works on computers with Windows platform that is Windows XP & Vista.

Some famous compilers/development tools supporting Windows for Atmel Microcontrollers are:

- WINAVR (AVRGCC for Windows)
- Code Vision AVR (CVAVR)
- AVR Studio (Atmel's free developing tool)

AVRGCC is a very nice open source compiler used by most of the people.

3.2 Programmer

Programmer basically consists of two parts:

- Software (to open .hex file on your computer)
- Hardware (to connect microcontroller)

Hardware depends on the communication port you are using on the computer (Serial, Parallel or USB). I suggest beginners to use Serial Programmer as it is very easy to build. Software for that is Pony Prog. Some famous Windows (XP, Vista) programmers are:

- Pony Prog (Serial, Parallel)
- AVRdude (supports many hardwares)
- AVRStudio (supports Atmel's hardware)
- ATProg (Serial)
- USB-ASP (USB)

An IDE has following functions:

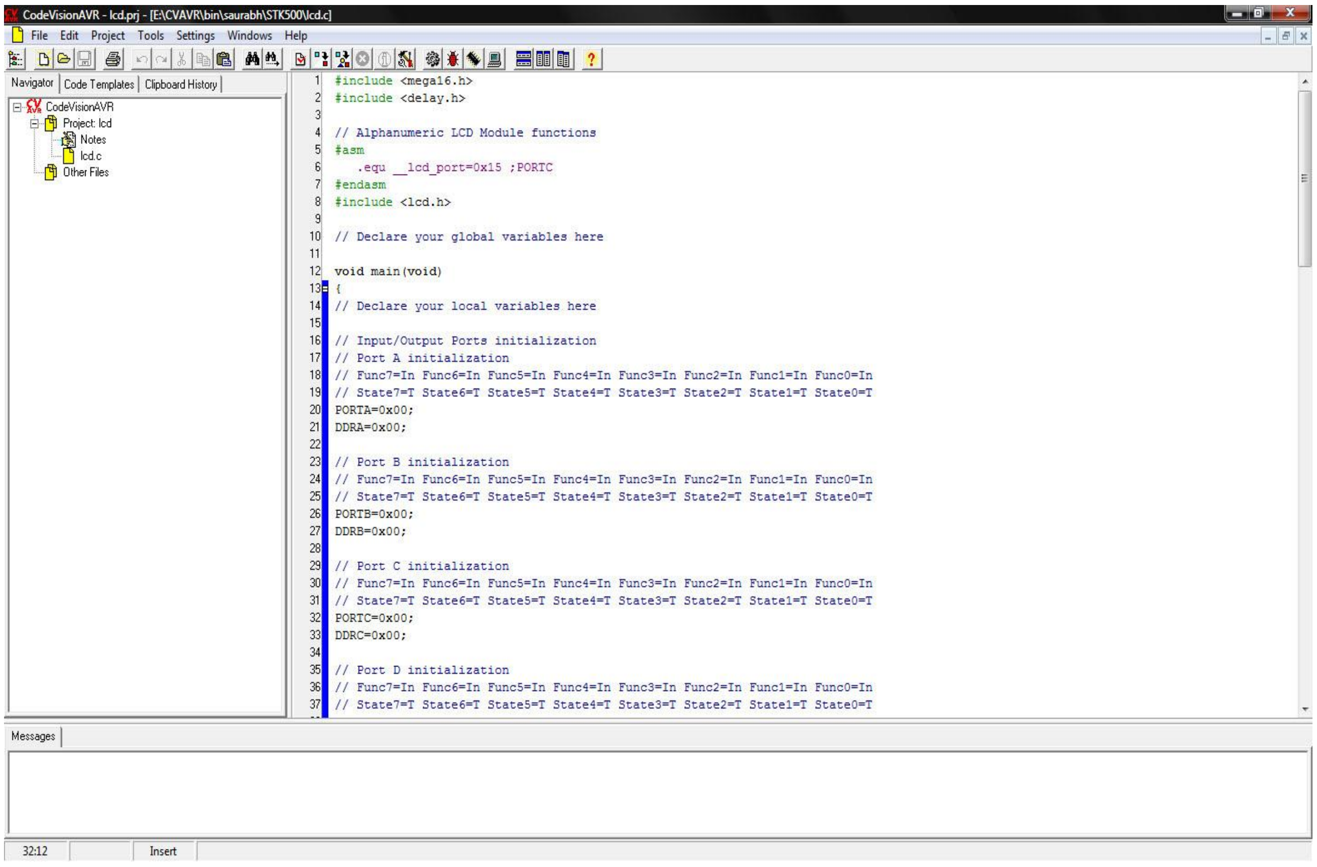
- Preprocessing
- Compilation
- Assembly
- Linking
- Object Translation
- Text Editor

If we just use compiler and linker independently we still need to get a text editor. So combining everything will actually mess things up. So the best way is to get Software which has it all. That's called an Integrated Development Environment, in short IDE.

I consider **Code-Vision-AVR** to be the best IDE for getting started with AVR programming on Windows XP, Vista. It has a very good Code Wizard which generate codes automatically ! You need not mess with the assembly words. So in all my tutorials I will be using CVAVR. You can download evaluation version for free which has code size limitation but good enough for our purpose.

For all my examples I will be using **Atmega-16** as default microcontroller because it very easily available and is powerful enough with sufficient number of pins and peripherals we use. You can have a look on the datasheet of **Atmega-16** in the datasheet section.

Let's take a look on the software. The main window looks like following,


CodeVisionAVR - lcd.prj - [E:\CVAVR\bin\saurabh\STK500\lcd.c]
File Edit Project Tools Settings Windows Help
Navigator
Code Templates
Clipboard History
CodeVisionAVR
Project: lcd
Notes
lcd.c
Other Files
Messages
32:12
Insert

```
#include <mega16.h>
#include <delay.h>

// Alphanumeric LCD Module functions
#asm
   .equ __lcd_port=0x15 ;PORTC
#endasm
#include <lcd.h>

// Declare your global variables here

void main(void)
{
// Declare your local variables here

// Input/Output Ports initialization
// Port A initialization
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
PORTA=0x00;
DDRA=0x00;

// Port B initialization
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
PORTB=0x00;
DDRB=0x00;

// Port C initialization
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
PORTC=0x00;
DDRC=0x00;

// Port D initialization
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
```

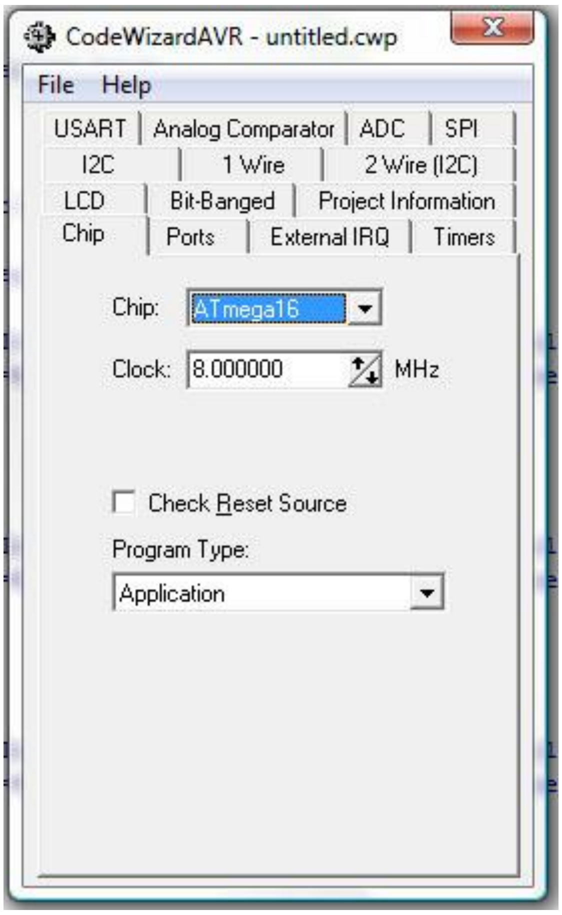
Now click on **File ---> New --->Project**

A pop up window will come asking whether you want to use Code Wizard AVR, obviously select yes because that is the reason we are using CVAVR !

Now have a look on this Wizard. It has many tabs where we can configure PORTS, TIMERS, LCD, ADC etc. I am explaining some of them

4.1 CHIP:

Select the chip for which you are going to write the program. Then select the frequency at which Chip is running. By default all chips are set on Internal Oscillator of 1 MHz so select 1 MHz if that is the case. If you want to change the running clock frequency of the chip then you have to change its fuse bits (I will talk more about this in fuse bits section).

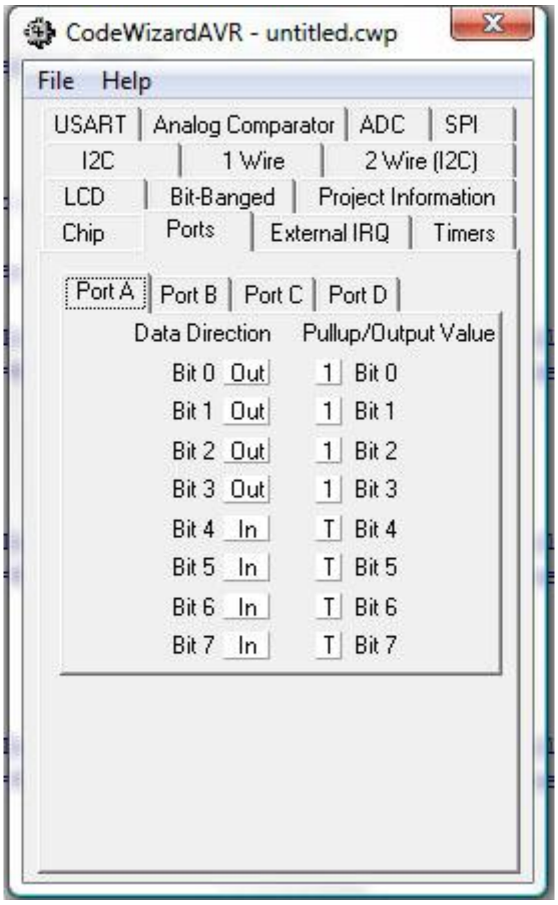


Chip: ATmega16

Clock: 8.000000 MHz

Check Reset Source [ ]

Program Type: Application



Data Direction		Pullup/Output Value	
Bit 0	Out	1	Bit 0
Bit 1	Out	1	Bit 1
Bit 2	Out	1	Bit 2
Bit 3	Out	1	Bit 3
Bit 4	In	T	Bit 4
Bit 5	In	T	Bit 5
Bit 6	In	T	Bit 6
Bit 7	In	T	Bit 7

4.2 PORT:

PORT is usually a collection of 8 pins.

From this tab you can select which pin you want to configure as output and which as input. It basically writes the DDR and PORT register through this setting. Registers are basically RAM locations which configure various peripherals of microcontroller and by changing value of these registers we can change the function it is performing. I will talk more about registers later. All the details are provided in the datasheet.

So you can configure any pin as output or input by clicking the box.

For Atmega-16 which has 4 Ports we can see 4 tabs each corresponding to one Port. You can also set initial value of the Pins you want to assign. or if you are using a pin as input then whether you want to make it as pull-up or tristated, again I will talk in details about these functions later.

Similarly using this code wizard you can very easily configure all the peripherals on the Atmega.

Now for generating code just go to **File ----> Generate, Save and Exit** (of the code wizard)

Now it will ask you name and location for saving three files. Two being project files and one being the .C file which is your program. try to keep same names of all three files to avoid confusion. By default these files are generated in **C:\CVAVR\bin**

The generated program will open in the text editor. Have a look it has some declarations like PORT, DDR, TCCR0 and many more. These are all registers which configures various functions of Atmega and by changing these value we make different functions. All the details about the registers are commented just below them. Now go down and find following infinite while loop there. We can start writing our part of program just before the while loop. And as for most of the applications we want microcontroller to perform the same task forever we put our part of code in the infinite while loop provided by the code wizard !

```
      while                                                    (1)
{
// Place your code here

      };
}
```

See how friendly this code wizard is, all the work (configuring registers) automatically done and we don’t even need to understand and go to the details about registers too !

Now we want to generate the hex file, so first compile the program. Either press F9 or go to **Project ---> Compile**.

It will show compilation errors if any. If program is error free we can proceed to making of hex file. So either press Shift+F9 or go to **Project ----> Make**. A pop up window will come with information about code size and flash usage etc.

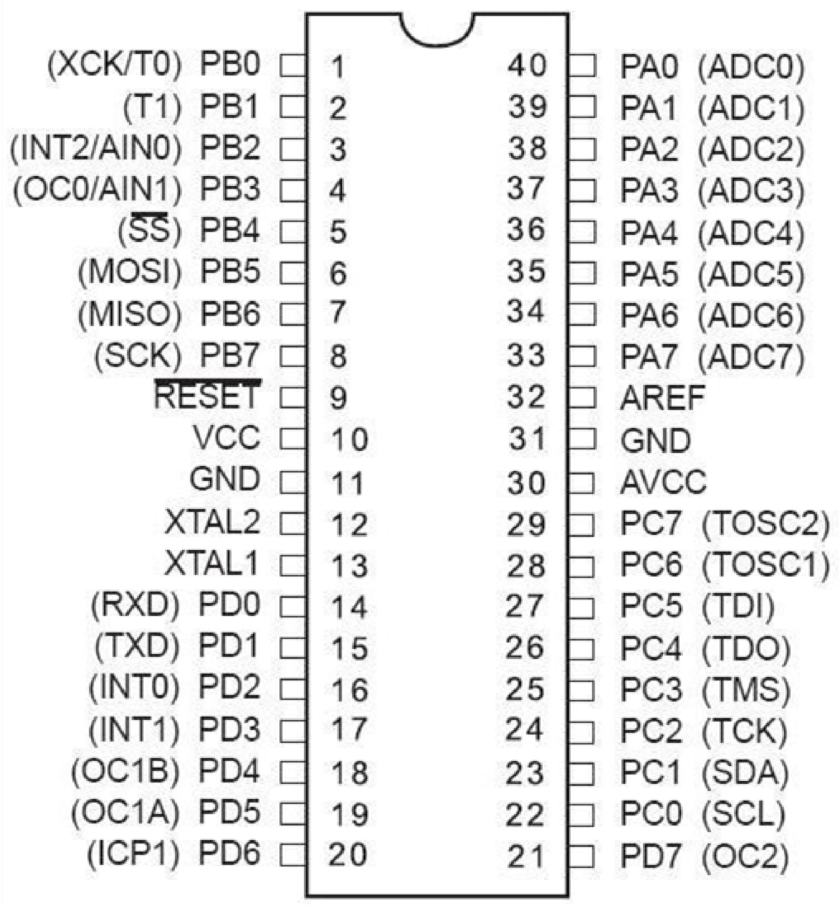
So the machine file is ready now ! It is in the same folder where we saved those 3 files.

Chapter 5 Introduction to Atmega 16 Microcontroller

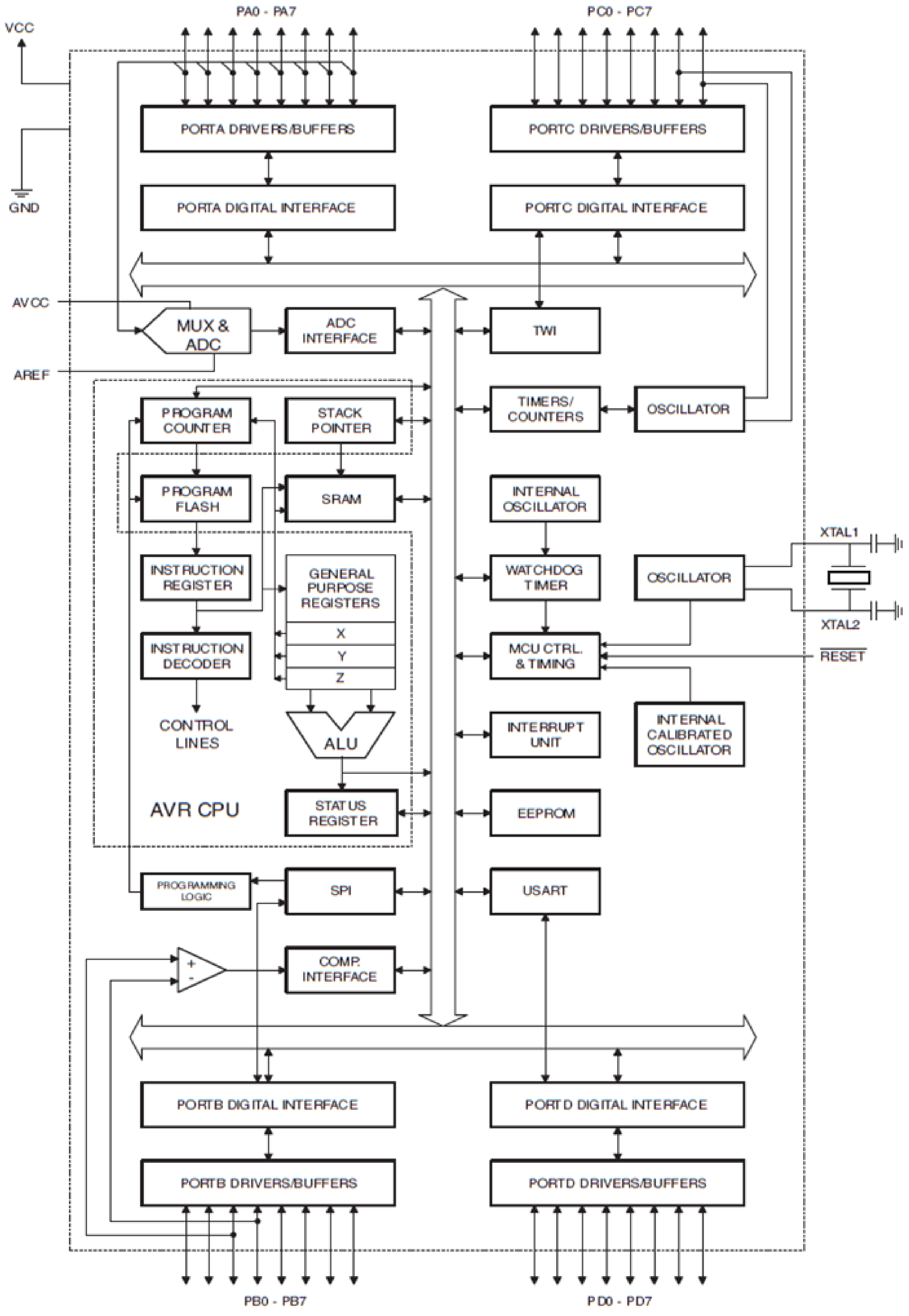
5.1 Features

- Advanced RISC Architecture
- Up to 16 MIPS Throughput at 16 MHz
- 16K Bytes of In-System Self-Programmable Flash
- 512 Bytes EEPROM
- 1K Byte Internal SRAM
- 32 Programmable I/O Lines
- In-System Programming by On-chip Boot Program
- 8-channel, 10-bit ADC
- Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
- One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture
- Four PWM Channels
- Programmable Serial USART
- Master/Slave SPI Serial Interface
- Byte-oriented Two-wire Serial Interface
- Programmable Watchdog Timer with Separate On-chip Oscillator
- External and Internal Interrupt Sources

5.2 Pin Configuration



5.3 Block Diagram



5.4 Pin Descriptions

VCC: Digital supply voltage. (+5V)

GND: Ground. (0 V) Note there are 2 ground Pins.

Port A (PA7 - PA0)
Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port B (PB7 - PB0)
Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port B also serves the functions of various special features of the ATmega16 as listed on page 58 of datasheet.

Port C (PC7 - PC0)
Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port C also serves the functions of the JTAG interface and other special features of the ATmega16 as listed on page 61 of datasheet. If the JTAG interface is enabled, the pull-up resistors on pins PC5 (TDI), PC3 (TMS) and PC2 (TCK) will be activated even if a reset occurs.

Port D (PD7 - PD0)
Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port D also serves the functions of various special features of the ATmega16 as listed on page 63 of datasheet.

RESET: Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running.

XTAL1: External oscillator pin 1

XTAL2: External oscillator pin 2

AVCC: AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to V_{CC}, even if the ADC is not used. If the ADC is used, it should be connected to V_{CC} through a low-pass filter.

AREF: AREF is the analog reference pin for the A/D Converter.

5.5 Digital Input Output Port

So let's start with understanding the functioning of AVR. We will first discuss about I/O Ports. Again I remind you that I will be using and writing about **Atmega-16**. Let's first have a look at the Pin configuration of Atmega-16. Image is attached, click to enlarge.

You can see it has 32 I/O (Input/Output) pins grouped as A, B, C & D with 8 pins in each group. This group is called as PORT.

- PA0 - PA7 (PORTA)
- PB0 - PB7 (PORTB)
- PC0 - PC7 (PORTC)
- PD0 - PD7 (PORTD)

Notice that all these pins have some function written in bracket. These are additional function that pin can perform other than I/O. Some of them are.

- ADC (ADC0 - ADC7 on PORTA)
- UART (Rx,Tx on PORTD)
- TIMERS (OC0 - OC2)
- SPI (MISO, MOSI, SCK on PORTB)
- External Interrupts (INT0 - INT2)

5.6 Registers

All the configurations in microcontroller is set through 8 bit (1 byte) locations in RAM (RAM is a bank of memory bytes) of the microcontroller called as **Registers**. All the functions are mapped to its locations in RAM and the value we set at that location that is at that Register configures the functioning of microcontroller. There are total 32 x 8bit registers in Atmega-16. As Register size of this microcontroller is 8 bit, it called as 8 bit microcontroller.

Chapter 6 I/O Ports:

Input Output functions are set by Three Registers for each PORT.

- DDRX ----> Sets whether a pin is Input or Output of PORTX.
- PORTX ---> Sets the Output Value of PORTX.
- PINX -----> Reads the Value of PORTX.

Go to the page 50 in the datasheet or you can also see the I/O Ports tab in the Bookmarks.

6.1 DDRX (Data Direction Register)

First of all we need to set whether we want a pin to act as output or input. DDRX register sets this. Every bit corresponds to one pin of PORTX. Let's have a look on DDRA register.

Bit	7	6	5	4	3	2	1	0
PIN	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

Now to make a pin act as I/O we set its corresponding bit in its DDR register.

- To make Input set bit 0
- To make Output set bit 1

If I write **DDRA = 0xFF** (0x for Hexadecimal number system) that is setting all the bits of DDRA to be 1, will make all the pins of PORTA as Output.

Similarly by writing **DDRD = 0x00** that is setting all the bits of DDRD to be 0, will make all the pins of PORTD as Input.

Now let's take another example. Consider I want to set the pins of PORTB as shown in table,

PORT-B	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
Function	Output	Output	Input	Output	Input	Input	Input	Output
DDRB	1	1	0	1	0	0	0	1

For this configuration we have to set DDRB as **11010001** which in hexadecimal is **D1**. So we will write **DDRB=0xD1**

Summary

- DDRX -----> to set PORTX as input/output with a byte.
- DDRX.y ---> to set yth pin of PORTX as input/output with a bit (works only with CVAVR).

6.2 PORTX (PORTX Data Register)

This register sets the value to the corresponding PORT. Now a pin can be Output or Input. So let's discuss both the cases.

6.2.1 Output Pin

If a pin is set to be output, then by setting bit 1 we make output **High** that is +5V and by setting bit 0 we make output **Low** that is 0V.

Let's take an example. Consider I have set DDRA=0xFF, that is all the pins to be Output. Now I want to set Outputs as shown in table,

PORT-A	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Value	High(+5V)	High(+5V)	Low(0V)	Low(0V)	Low(0V)	High(+5V)	High(+5V)	Low(0V)
PORTA	1	1	0	0	0	1	1	0

For this configuration we have to set **PORTA** as **11000110** which in hexadecimal is **C6**. So we will write **PORTA=0xC6;**

6.2.2 Input Pin

If a pin is set to be input, then by setting its corresponding bit in PORTX register will make it as follows,

- Set bit 0 ---> Tri-Stated
- Set bit 1 ---> Pull Up

Tristated means the input will ***hang*** (no specific value) if no input voltage is specified on that pin.

Pull Up means input will go to **+5V** if no input voltage is given on that pin. It is basically connecting PIN to +5V through a 10K Ohm resistance.

Summary

- PORTX ----> to set value of PORTX with a byte.
- PORTX.y --> to set value of yth pin of PORTX with a bit (works only with CVAVR).

6.3 PINX (Data Read Register)

This register is used to read the value of a PORT. If a pin is set as input then corresponding bit on PIN register is,

- 0 for Low Input that is V < 2.5V
- 1 for High Input that is V > 2.5V (Ideally, but actually 0.8 V - 2.8 V is error zone !)

For an example consider I have connected a sensor on PC4 and configured it as an input pin through DDR register. Now I want to read the value of PC4 whether it is Low or High. So I will just check 4th bit of PINC register.

We can only read bits of the PINX register; can never write on that as it is meant for reading the value of PORT.

Summary

- PINX ----> Read complete value of PORTX as a byte.
- PINX.y --> Read y^{th} pin of PORTX as a bit (works only with CVAVR).

I hope you must have got basic idea about the functioning of I/O Ports. For detailed reading you can always refer to datasheet of Atmega.

Chapter 7 LCD Interfacing

Now we need to interface an LCD to our microcontroller so that we can display messages, outputs, etc. Sometimes using an LCD becomes almost inevitable for debugging and calibrating the sensors (discussed later). We will use the 16x2 LCD, which means it has two rows of 16 characters each. Hence in total we can display 32 characters.

7.1 Circuit Connection

There are 16 pins in an LCD; See reverse side of the LCD for the PIN configuration.
The connections have to be made as shown below:

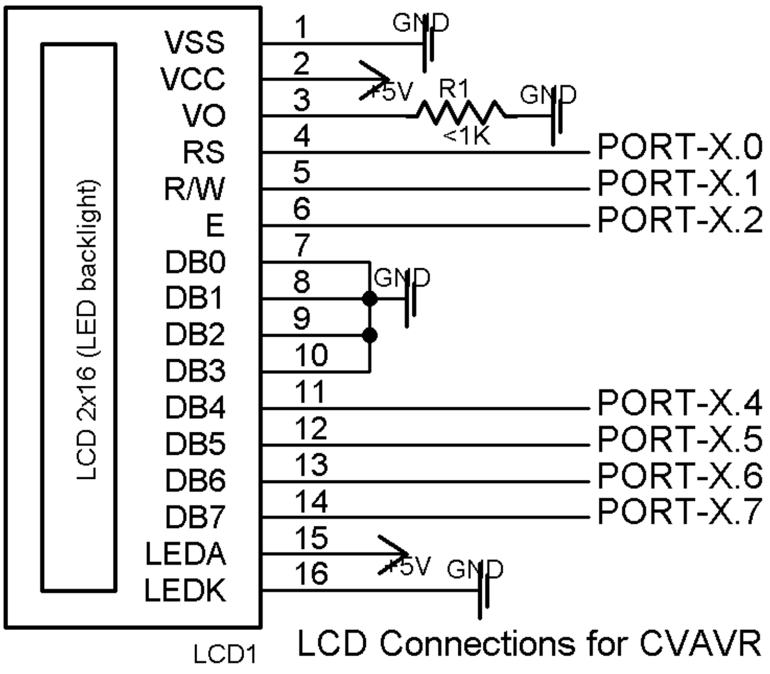
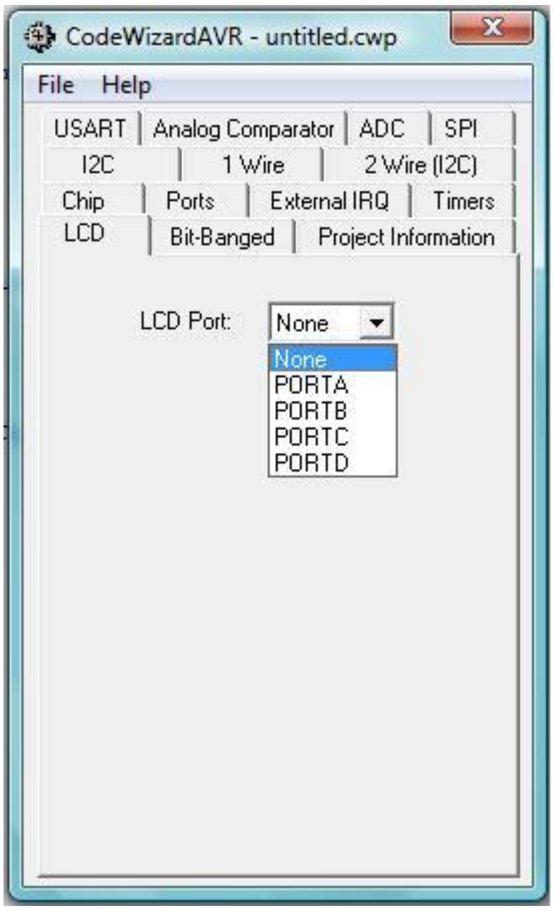


Figure 15: LCD connections

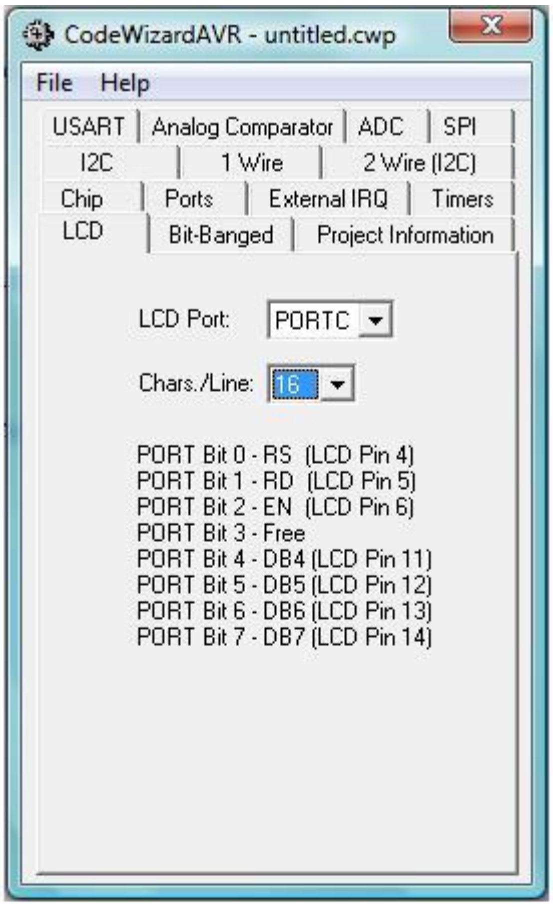
7.2 Setting up in Microcontroller

When we connect an LCD to Atmega16, one full PORT is dedicated to it, denoted by PORT-X in the figure. To enable LCD interfacing in the microcontroller, just click on the LCD tab in the Code Wizard and select the PORT at which you want to connect the LCD. We will select PORTC. Also select the number of characters per line in your LCD. This is 16 in our case. Code Wizard now shows you the complete list of connections which you will have to make in order to interface the LCD. These are nothing but the same as in the above figure for general PORT-X.



LCD Port: None

None
PORTA
PORTB
PORTC
PORTD



LCD Port: PORTC

Chars./Line: 16

```
PORT Bit 0 - RS  (LCD Pin 4)
PORT Bit 1 - RD  (LCD Pin 5)
PORT Bit 2 - EN  (LCD Pin 6)
PORT Bit 3 - Free
PORT Bit 4 - DB4 (LCD Pin 11)
PORT Bit 5 - DB5 (LCD Pin 12)
PORT Bit 6 - DB6 (LCD Pin 13)
PORT Bit 7 - DB7 (LCD Pin 14)
```

Figure 16: LCD settings on CVAVR wizard window.

As you can see, there are some special connections other than those to uC, Vcc and gnd. These are general LCD settings. Pin 3 (VO) is for the LCD contrast, ground it through a <1kΩ resistance/ potentiometer for optimum contrast. Pin 15 & 16 (LEDA and LEDK) are for LCD backlight, give them permanent +5V and GND respectively as we need to glow it continuously.

7.3 Printing Functions

Now once the connections have been made, we are ready to display something on our screen. Displaying our name would be great to start with. Some of the general LCD functions which you must know are:

7.3.1 lcd_clear()

Clears the lcd. Remember! Call this function before the while(1) loop, otherwise you won't be able to see anything!

7.3.2 lcd_gotoxy(x,y)

Place the cursor at coordinates (x,y) and start writing from there. The first coordinate is (0,0). Hence, x ranges from 0 to 15 and y from 0 to 1 in our LCD. Suppose you want to display something starting from the 5th character in second line, then the function would be

lcd_gotoxy(5,1);

7.3.3 lcd_putchar(char c)

To display a single character. E.g.,
lcd_putchar('H');

7.3.4 lcd_putsf(constant string)

To display a constant string. Eg,

lcd_putsf("IIT Kanpur");

7.3.5 lcd_puts(char arr)

To display a variable string, which is nothing but an array of characters (data type char) in C language . e.g., You have an array char c[10] which keeps on changing. Then to display it, the function would be called as
lcd_puts(c);

Now we have seen that only characters or strings (constant or variable) can be displayed on the LCD. But quite often we have to display values of numeric variables, which is not possible directly. Hence we need to first convert that numeric value to a string and then display it. For e.g., if we have a variable of type integer, say **int** k, and we need to display the value of k (which changes every now and then, 200 now and 250 after a second... and so on). For this, we use the C functions **itoa()** and **ftoa()**, but remember to include the header file **stdlib.h** to use these C functions.

7.3.6 itoa(int val, char arr[])

It stores the value of integer val in the character array arr. E.g., we have already defined int i and char c[20], then

itoa(i,c);
lcd_puts(c);

Similarly we have

7.3.7 ftoa(float val, char decimal_places, char arr[])

It stores the value of floating variable f in the character array arr with the number of decimal places as specified by second parameter. E.g., we have already defined float f and char c[20], then

ftoa(f,4,c);   *// till 4 decimal places*
lcd_puts(c);

Now we are ready to display anything we want on our LCD. Just try out something which you would like to see glowing on it!

Chapter 8 UART Communication

UART (Universal Asynchronous Receiver Transmitter) is a way of communication between the microcontroller and the computer system or another microcontroller. There are always two parts to any mode of communication-a Receiver and a Transmitter. Hence, our Atmega can receive data as well as send data to other microcontroller, computer or any other device.

8.1 UART: Theory of Operation

Figure 16 illustrates a basic UART data packet. While no data is being transmitted, logic 1 must be placed in the Tx line. A data packet is composed of 1 start bit, which is always a logic 0, followed by a programmable number of data bits (typically between 6 to 8), an optional parity bit, and a programmable number of stop bits (typically 1). The stop bit must always be logic 1.

Most UART uses 8bits for data, no parity and 1 stop bit. Thus, it takes 10 bits to transmit a byte of data.

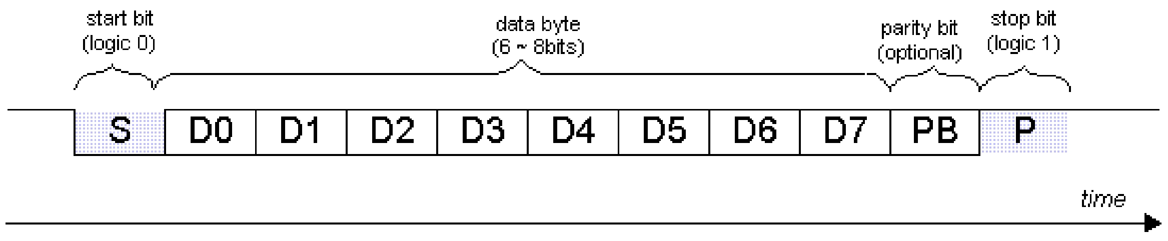


Figure 17: Basic UART packet format: 1 start bit, 8 data bits, 1 parity bit, 1 stop bit.

BAUD Rate: This parameter specifies the desired baud rate (bits per second) of the UART. Most typical standard baud rates are: 300, 1200, 2400, 9600, 19200, etc. However, any baud rate can be used. This parameter affects both the receiver and the transmitter. The default is 2400 (bauds).

In the UART protocol, the transmitter and the receiver do not share a clock signal. That is, a clock signal does not emanate from one UART transmitter to the other UART receiver. Due to this reason the protocol is said to be **asynchronous**.

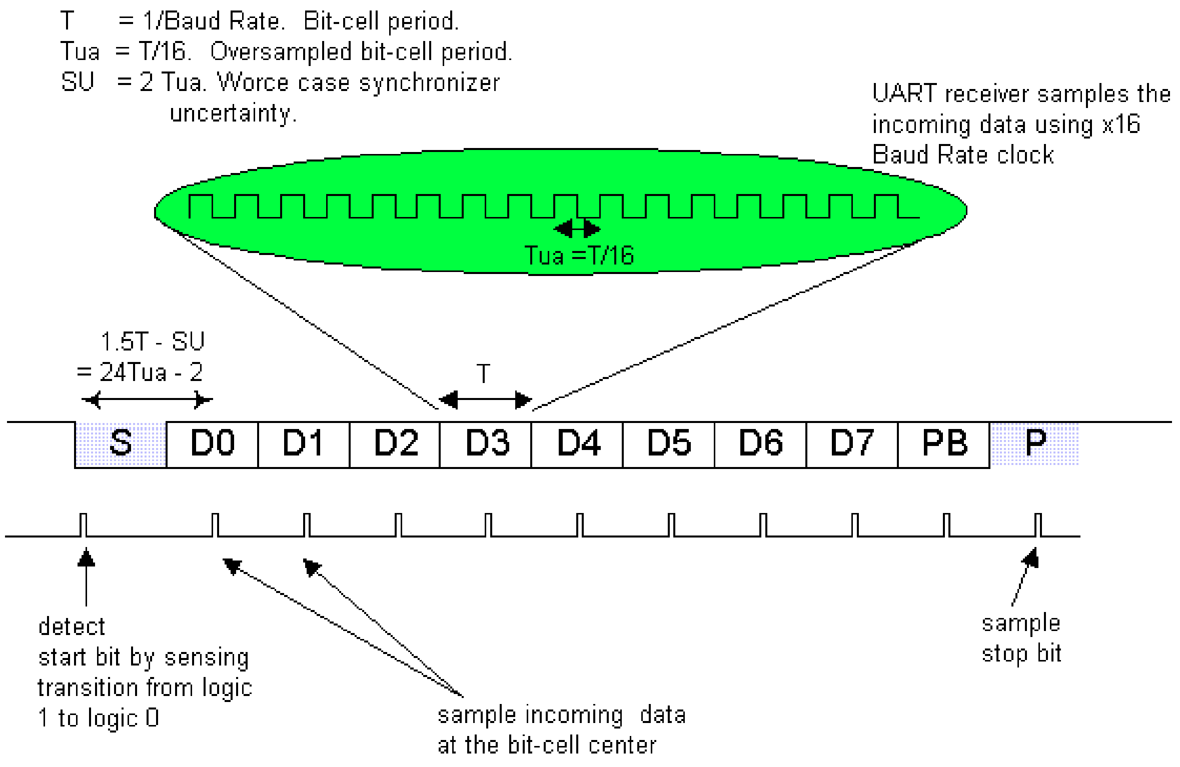
Since no common clock is shared, a known data transfer rate (baud rate) must be agreed upon prior to data transmission. That is, the receiving UART needs to know the transmitting UART's baud rate (and conversely the transmitter needs to know the receiver's baud rate, if any). In almost all cases the receiving and transmitting baud rates are the same. The transmitter shifts out the data starting with the LSB first.

Once the baud rate has been established (prior to initial communication), both the transmitter and the receiver's internal clock is set to the same frequency (though not the same phase). The receiver "synchronizes" its internal clock to that of the transmitter's at the beginning of every data packet received. This allows the receiver to sample the data bit at the bit-cell center.

A key concept in UART design is that UART's internal clock runs at much faster rate than the baud rate. For example, the popular 16450 UART controller runs its internal clock at 16 times the baud rate. This allows the UART receiver to sample the incoming data with granularity of 1/16 the baud-rate period. This "oversampling" is critical since the receiver adds about 2 clock-ticks in the input data synchronizer uncertainty. The incoming data is not sampled directly by the receiver, but goes through a synchronizer which translates the clock domain from the transmitter's to that of the receiver. Additionally, the greater the granularity, the receiver has greater immunity with the baud rate error.

The receiver detects the start bit by detecting the transition from logic 1 to logic 0 (note that while the data line is idle, the logic level is high). In the case of 16450 UART, once the start-bit is detected, the next data bit's "center" can be assured to be 24 ticks minus 2 (worse case synchronizer uncertainty) later. From then on, every next data bit center is 16 clock ticks later. Figure 2 illustrates this point.

Once the start bit is detected, the subsequent data bits are assembled in a de-serializer. Error condition maybe generated if the parity/stop bits are incorrect or missing.



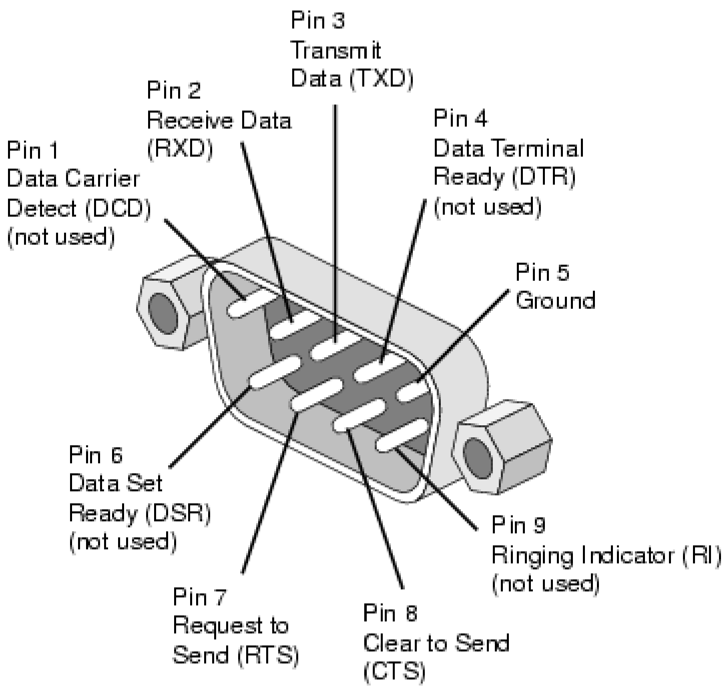
T = 1/Baud Rate. Bit-cell period.
Tua = T/16. Oversampled bit-cell period.
SU = 2 Tua. Worce case synchronizer uncertainty.
UART receiver samples the incoming data using x16 Baud Rate clock
Tua =T/16
1.5T - SU = 24Tua - 2
T
S
D0
D1
D2
D3
D4
D5
D6
D7
PB
P
detect start bit by sensing transition from logic 1 to logic 0
sample incoming data at the bit-cell center
sample stop bit

Figure 18: Data sampling points by the UART receiver.

8.2 Serial Port of Computer

We will be using Serial Port for communication between the uC and the computer. A serial port has 9 pins as shown. If you have a laptop, then most probably there won't be a serial port. Then you can use a USB to serial Converter.

If you have to transmit one byte of data, the serial port will transmit 8 bits as one bit at a time. The advantage is that a serial port needs only one wire to transmit the 8 bits.



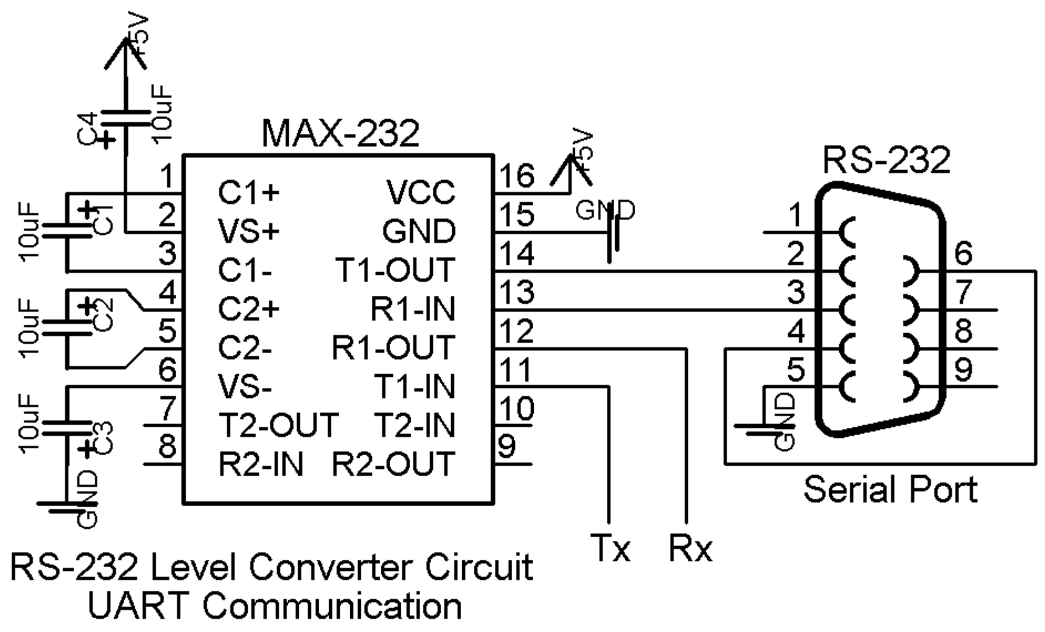
Pin 1 Data Carrier Detect (DCD) (not used)
Pin 2 Receive Data (RXD)
Pin 3 Transmit Data (TXD)
Pin 4 Data Terminal Ready (DTR) (not used)
Pin 5 Ground
Pin 6 Data Set Ready (DSR) (not used)
Pin 7 Request to Send (RTS)
Pin 8 Clear to Send (CTS)
Pin 9 Ringing Indicator (RI) (not used)

Figure 19: A Serial Port

Pin 3 is the Transmit (TX) pin, pin 2 is the Receive (RX) pin and pin 5 is Ground pin. Other pins are used for controlling data communication in case of a modem. For the purpose of data transmission, only the pins 3 and 5 are required.

The standard used for serial communication is **RS-232** (Recommended Standard 232). The RS-232 standard defines the voltage levels that correspond to logical one and logical zero levels. Valid signals are plus or minus 3 to 15 volts. The range near zero volts is not a valid RS-232 level; logic one is defined as a negative voltage, the signal condition is called marking, and has the functional significance of OFF. Logic zero is positive; the signal condition is spacing, and has the function ON.

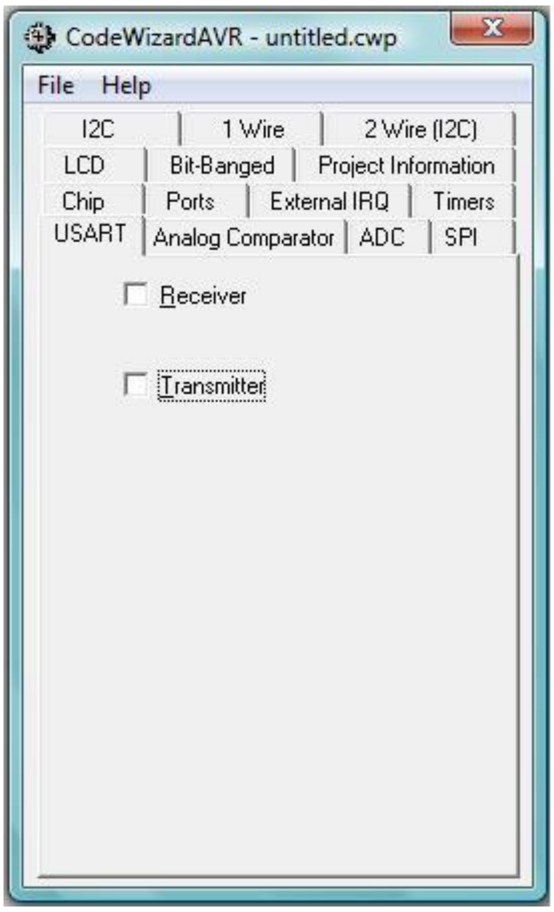
Now we know that this is not the voltage level at which our microcontroller works. Hence, we need a device which can convert this voltage level to that of CMOS, i.e., logic 1 = +5V and logic 0 = 0V. This task is carried out by an IC MAX 232, which is always used with four 10uF capacitors. The circuit is shown:



The Rx and Tx shown in above figure (pins 11 and 12 of MAX232) are the Rx and Tx of Atmega 16 (PD0 and PD1 respectively).

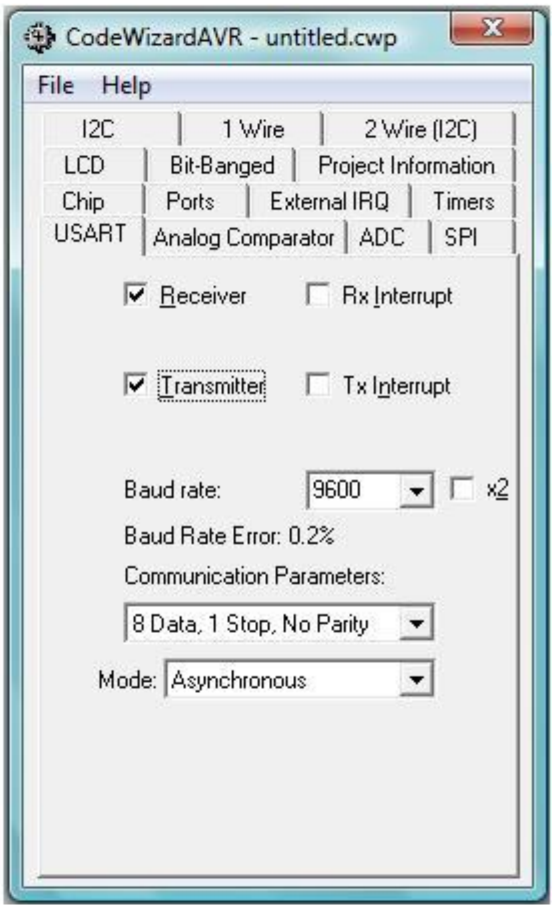
8.3 Setting up UART in microcontroller

Once our electronic circuit is complete, we can start with coding. To enable UART mode in Atmega16, click on the USART tab in Code Wizard. Now depending upon your requirement, you can either check receiver, transmitter or both. You get a list of options to select from for the baud rate. **Baud Rate** is the unit of data transfer, defined as bits per second. We will select 9600 as it is fair enough for our purpose, and default setting in most of the applications (like Matlab, Docklight, etc.). Keep the communication parameters as default, i.e., 8 data, 1 stop and no parity and mode asynchronous. You also have option of enabling Rx and Tx Interrupt functions, but we won't do at this point.



☐ Receiver

☐ Transmitter



☑ Receiver ☐ Rx Interrupt

☑ Transmitter ☐ Tx Interrupt

Baud rate: 9600 ☐ x2

Baud Rate Error: 0.2%

Communication Parameters:

8 Data, 1 Stop, No Parity

Mode: Asynchronous

Once you generate and save the code, all the register values are set by CVAVR and you only need to know some of the C functions to transfer data. Some of them are:

8.3.1 putchar()

To send one character to the buffer, which will be received by the device (uC or computer)? E.g.,

putchar('A'); *//sends character 'A' to the buffer*
putchar(c); *// sends a variable character c*

8.3.2 getchar()

To receive one character from the buffer, which might have been sent by the other uC or the computer. E.g., if we have already defined a variable char c, then

c = getchar(); *// receives the character from buffer and save it in variable c*

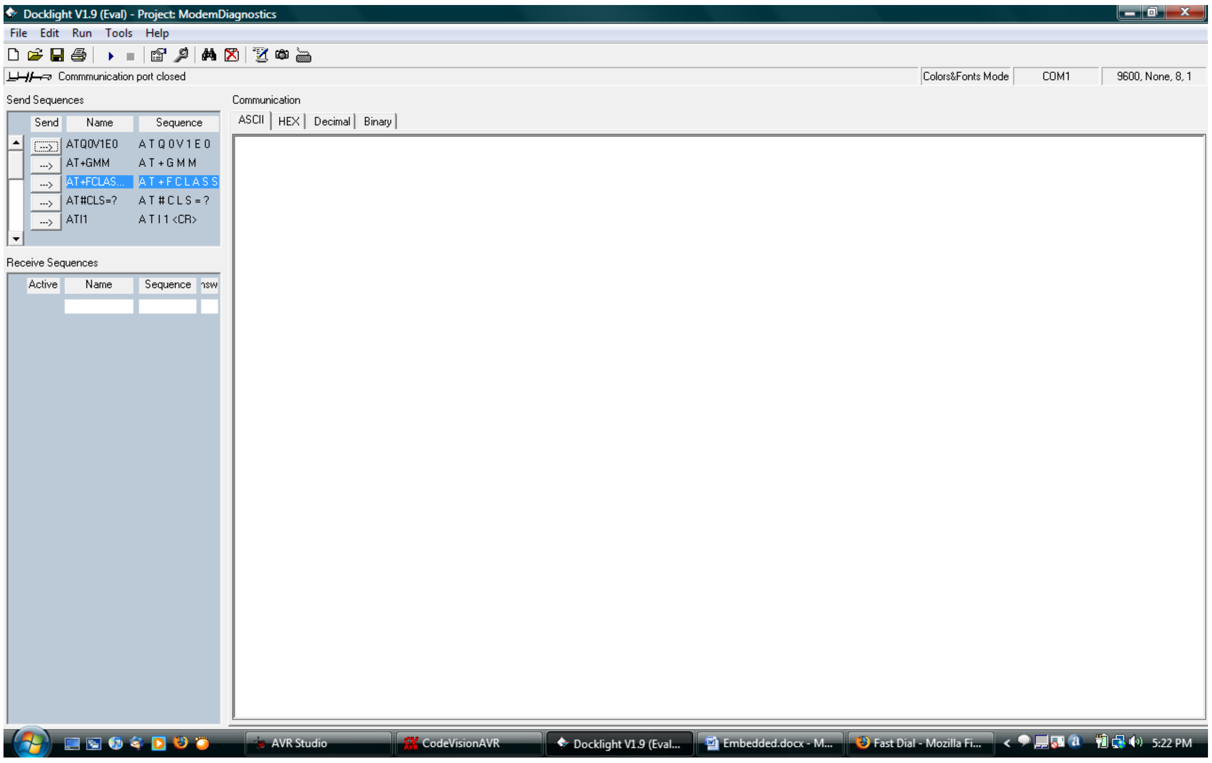
8.3.3 putsf()

To send a constant string. Eg,

putsf("Robocon Team");

8.4 Docklight

To communicate with the computer, you need a terminal where you can send data through keyboard and the received data can be displayed on the screen. There are many softwares which provide such terminal, but we will be using Docklight. Its evaluation version is free for download on internet, which is sufficient for our purpose.



Commmunication port closed | Colors&Fonts Mode | COM1 | 9600, None, 8, 1

Send	Name	Sequence
--->	ATQ0V1E0	A T Q 0 V 1 E 0
--->	AT+GMM	A T + G M M
--->	AT+FCLAS...	A T + F C L A S S
--->	AT#CLS=?	A T # C L S = ?
--->	ATI1	A T I 1 <CR>

Active	Name	Sequence	nsw

To start with, check the Terminal Settings in Docklight. Go to **Tools -> Project Settings**. Select the Send/Receive communication channel, i.e., the name by which the serial port is known in your computer (like COM1…). In the COM port settings, select the **same values** as you had set while coding Atmega 16. So, we will select Baud Rate 9600, Data Bits 8, Stop Bits 1, Parity Bits none. You can select 'none' in Parity Error Character. Click OK.

We are now ready to send/receive data, so, select **Run -> Start Communication**, or, press F5. If your uC is acting as a transmitter, then the characters it sends will appear in the Communication window of Docklight. E.g.,

```
putchar('K');
delay_ms(500);                    // Sends character K after every 500ms
```

Hence what you get on the screen is a KKKKKKKKKKKKKKKKKKKKKKKKK…….. one K increasing every 500ms. To stop receiving characters, select **Run -> Stop Communication**, or press F6.

If the receiver option is also enabled in Atmega16, then whatever you type from keyboard will be received by it. You can either display these received characters on an LCD, control motors depending on what characters you send, etc. e.g.,

```
c = getchar() ;             // receive character
lcd_putchar(c);             // display it on LCD
if (c=='A')
    PORTA=255;              // if that character is A, make all pins of PORTA high.
```

When you are done with sending data, select **Run -> Stop Communication**, or press F6.

Chapter 9 Timers

9.1 Basic Theory

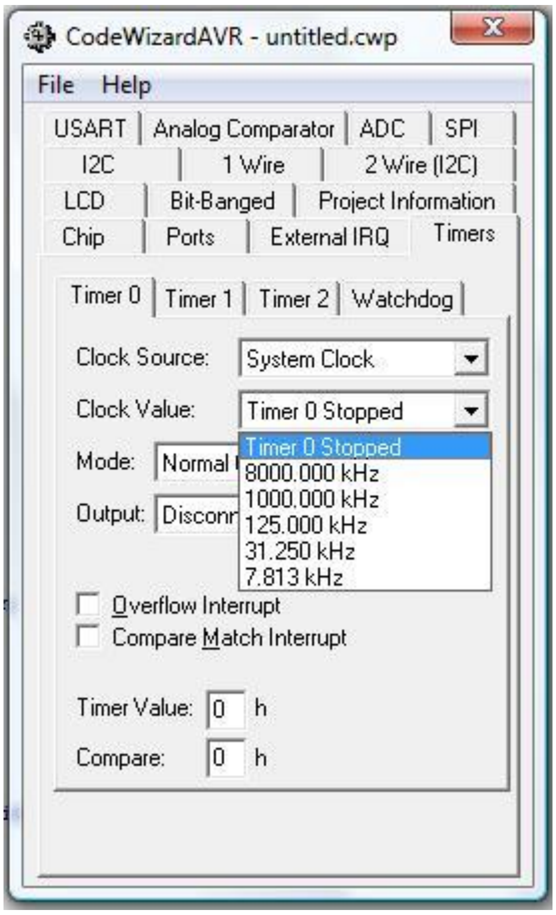
Atmega 16 has following timers,

- Timer 0, 8 bit
- Timer 1, 16 bit consisting of two 8 bit parts, A and B
- Timer 2, 8 bit

Now there are two clocks,

1. **System Clock (fs):** This is the clock frequency at which Atmega is running. By default it is 1 MHz which can be changed by setting fuse bits.

2. **Timer Clock (ft):** This is the clock frequency at which timer module is running. Each timer module has different clocks.



Clock Source: System Clock

Clock Value: Timer 0 Stopped

- Timer 0 Stopped
- 8000.000 kHz
- 1000.000 kHz
- 125.000 kHz
- 31.250 kHz
- 7.813 kHz

Mode: Normal

Output: Disconn

☐ Overflow Interrupt

☐ Compare Match Interrupt

Timer Value: 0 h

Compare: 0 h

Now **ft** can be in ratios of **fs.** That is, ft = fs, fs/8, fs/64 ...

For example, if we keep fs = 8 MHz then available options for ft are: 8 MHz, 1 MHz, 125 KHz ... (look in the image)

There are total 4 settings for Timers,

1. **Clock Source:** Source for timer clock, keep it as system clock. You can also provide external clock. Read datasheet for more information about external clock source.

2. **Clock value:** This is value of **ft.** Drop down for available options of ft. Chose whichever is required

3. **Mode:** There are many modes of timers. We will be discussing following 2 modes,

   a. Fast PWM top = FFh
   b. CTC top=OCRx (x=0, 1A, 2)

4. **Output**: Depending upon the mode we have chosen there are options for output pulse. We will look in detail later.

Basically each Timer has a counter unit with size 8 bit for **Timer 0, 2** and 16 bit for **Timer 1.** I will be talking about Timer 0 and same will follow for other Timers.

Each counter has a register which increments by one on every rising edge of timer clock. After counting to its full capacity, 255 for 8 bit, it again starts from 0. By using this register we can have different modes.

- Timer/Counter (**TCNT0**) and Output Compare Register (**OCR0**) are 8-bit registers.

- **TOP:** The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be the fixed value 0xFF (MAX) or the value stored in the OCR0 Register. The assignment is dependent on the mode of operation.

9.2 Fast PWM Mode

PWM = Pulse Width Modulation.

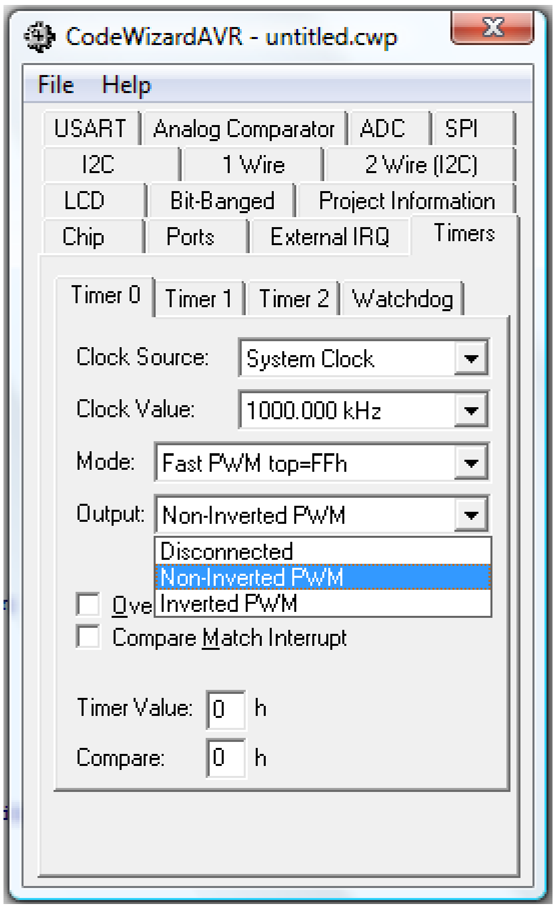
This mode is used to generate pulse with

- Fixed Frequency (F)
- Variable Duty Cycle (D)

F = Ft / 256

D = OCR0 / 255 (non inverted)

D = (255-OCR0) / 255 (inverted)



CodeWizardAVR - untitled.cwp

File Help

USART | Analog Comparator | ADC | SPI | I2C | 1 Wire | 2 Wire (I2C) | LCD | Bit-Banged | Project Information | Chip | Ports | External IRQ | Timers

Timer 0 | Timer 1 | Timer 2 | Watchdog

Clock Source: System Clock

Clock Value: 1000.000 kHz

Mode: Fast PWM top=FFh

Output: Non-Inverted PWM

Disconnected

Non-Inverted PWM

Inverted PWM

☐ Ove

☐ Compare Match Interrupt

Timer Value: 0 h

Compare: 0 h

By changing OCR0 value we can change the duty cycle of the output pulse.

As OCR0 is an 8bit register it can vary from 0 o 255.

$0 \leq OCR0 \leq 255$

Pins for Output pulse,

Timer 0	:	OC0,	pin 4
Timer 1A	:	OC1A,	pin 19
Timer 1B	:	OC1B,	pin 18
Timer 2	:	OC2,	pin 21

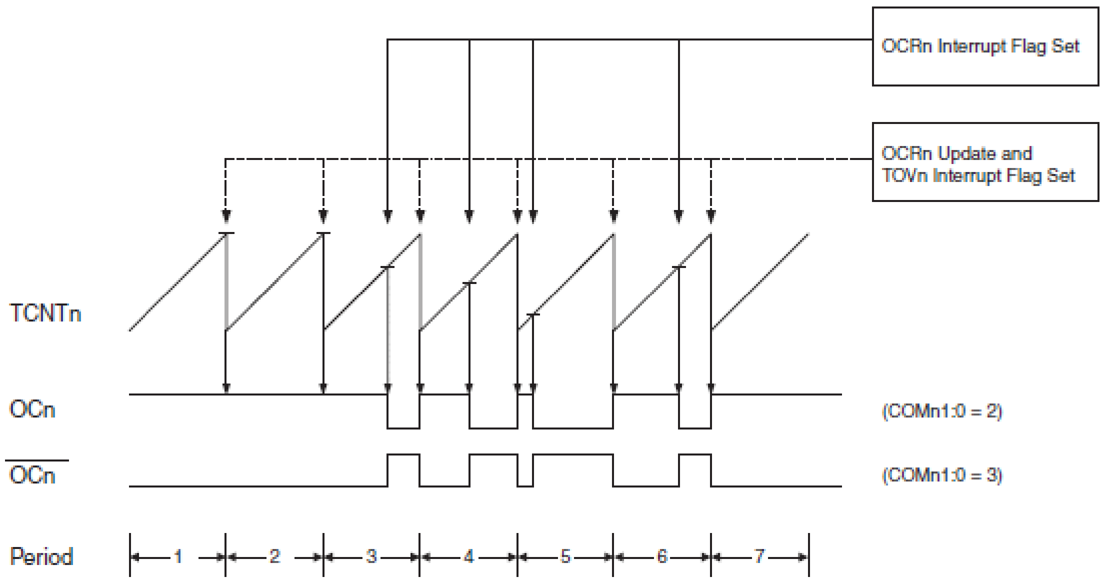


Figure 20: Fast PWM mode Timing Diagram

9.3 CTC Mode

CTC = Clear Timer on Compare Match.

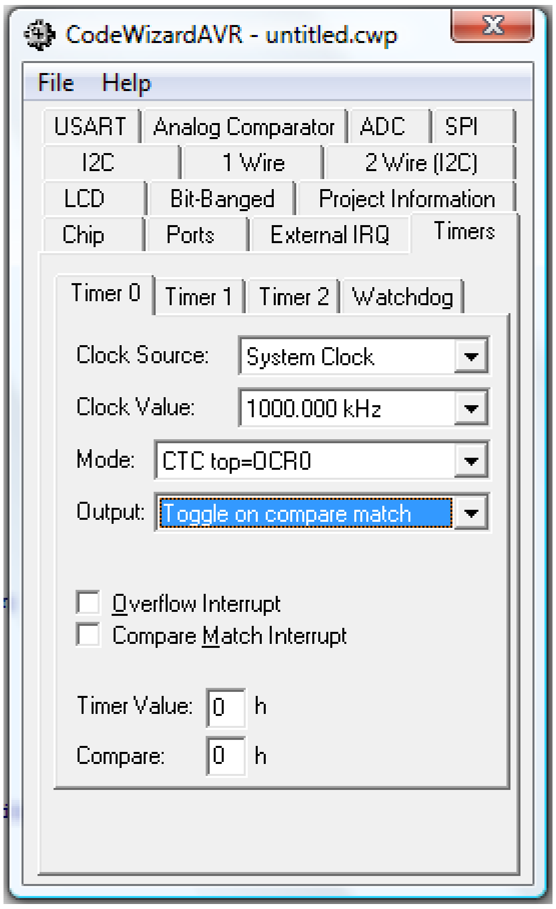
This mode is to generate pulse with,

- Fixed Duty Cycle (D = 0.5)
- Variable Frequency (F)

$$\boxed{\begin{array}{l} \mathbf{F} = \dfrac{\mathbf{ft}}{2\,(OCR0+1)} \\[2ex] \mathbf{D = 0.5} \end{array}}$$

By changing value of OCR0 we can change the value of output pulse frequency. As OCR0 is an 8bit register it can vary from 0 to 255.

$$0 \leq OCR0 \leq 255$$


CodeWizardAVR - untitled.cwp

File Help

USART | Analog Comparator | ADC | SPI | I2C | 1 Wire | 2 Wire (I2C) | LCD | Bit-Banged | Project Information | Chip | Ports | External IRQ | Timers

Timer 0 | Timer 1 | Timer 2 | Watchdog

Clock Source: System Clock

Clock Value: 1000.000 kHz

Mode: CTC top=OCR0

Output: Toggle on compare match

☐ Overflow Interrupt

☐ Compare Match Interrupt

Timer Value: 0 h

Compare: 0 h

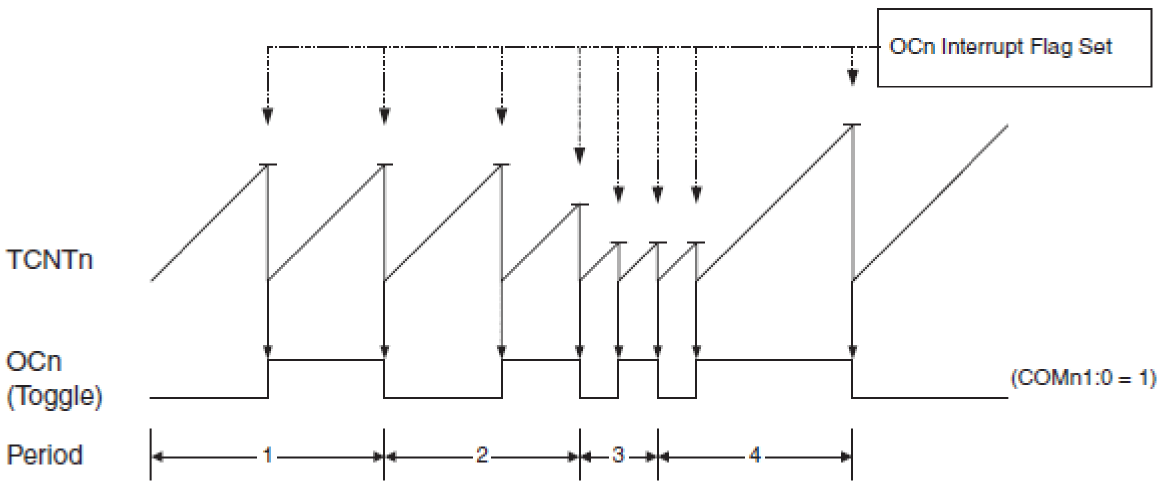


Figure 21: CTC Mode timing diagram

Chapter 10 SPI: Serial Peripheral Interface

10.1 Theory of Operation

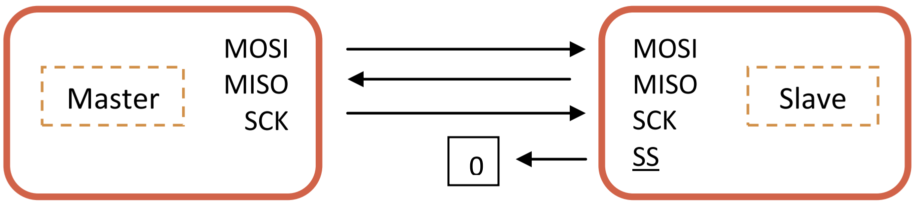
The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link used to communicate between two or more microcontroller and devices supporting SPI mode data transfer. Devices communicate in master/slave mode where the master device initiates the data frame. Multiple slave devices are allowed with individual slave select (chip select) lines.

This communication protocol consists of folliwng lines or pins,

1. **MOSI** : **M**aster **O**ut **S**lave **I**n (Tx for Master and Rx for Slave)
2. **MISO** : **M**aster **I**n **S**lave **O**ut (Rx for Master Tx for Slave)
3. **SCK** : **S**erial **C**lock (Clock line)
4. **$\overline{\text{SS}}$** : **S**lave **S**elect (To select Slave chip) (if given 0 device acts as slave)

Master: This device provides the serial clock to the other device for data transfer. As a clock is used for the data transfer, this protocol is Synchronous in nature. **$\overline{\text{SS}}$** for Master will be disconnected.

Slave: This device accepts the clock from master device. **$\overline{\text{SS}}$** for this has to be made 0 externally.



Pin connections for SPI protocol

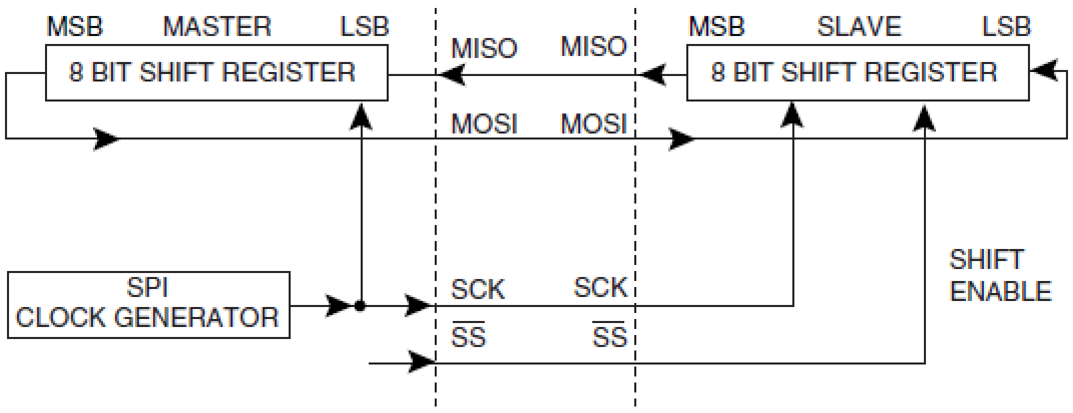


Figure 22: SPI Data Communication

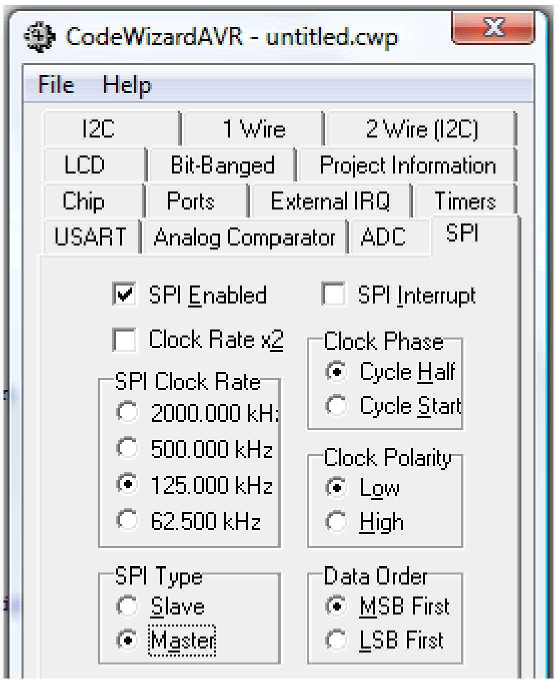
10.2 Setting up SPI in Microcontroller

10.2.1 Master Microcontroller

Open Code Wizard and go to **SPI** tab. **Enable SPI** and choose Master in SPI Type.

Select clock rate depending upon your data transfer speed requirement.

Keep other settings as default.



[✓] SPI Enabled [ ] SPI Interrupt

[ ] Clock Rate x2

SPI Clock Rate: 2000.000 kH: / 500.000 kHz / [✓ 125.000 kHz] / 62.500 kHz

Clock Phase: [✓ Cycle Half] / Cycle Start

Clock Polarity: [✓ Low] / High

SPI Type: Slave / [✓ Master]

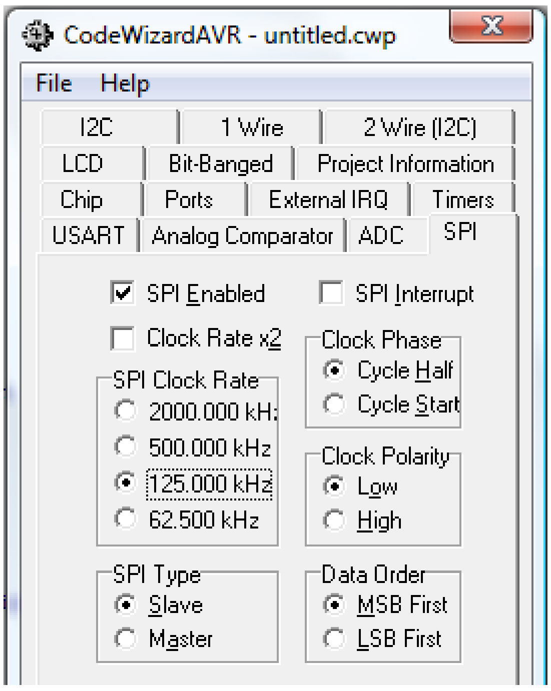
Data Order: [✓ MSB First] / LSB First

10.2.2 Slave Microcontroller

Open Code Wizard and go to **SPI** tab. **Enable SPI** and choose Slave in SPI Type.

Select clock rate depending upon your data transfer speed requirement.

Keep other settings as kept in the master microcontroller.



[✓] SPI Enabled [ ] SPI Interrupt

[ ] Clock Rate x2

SPI Clock Rate: 2000.000 kH: / 500.000 kHz / [✓ 125.000 kHz] / 62.500 kHz

Clock Phase: [✓ Cycle Half] / Cycle Start

Clock Polarity: [✓ Low] / High

SPI Type: [✓ Slave] / Master

Data Order: [✓ MSB First] / LSB First

10.3 Data Functions

You can transmit or receive 1 byte of data at a time.

10.3.1 Transmit Data

spi(1 byte data); Example: spi('A');

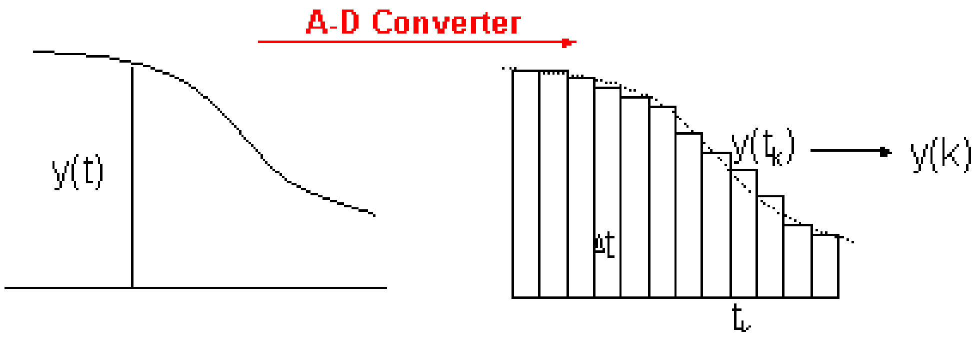
10.3.2 Receive Data

c = spi(0); // c is 1 byte variable Example: char c = spi(0);

Chapter 11 ADC: Analog to Digital Converter

11.1 Theory of operation

What we have seen till now that the input given to uC was digital, i.e., either +5 V (logic 1) or 0V (logic 0). But what if we have an analog input, i.e., value varies over a range, say 0V to +5V? Then we require a tool that converts this analog voltage to discrete values. Analog to Digital Converter (ADC) is such a tool.



ADC is available at PORTA of Atmega16. Thus we have 8 pins available where we can apply analog voltage and get corresponding digital values. The ADC register is a 10 bit register, i.e., the digital value ranges from 0 to 1023. But we can also use only 8 bit out of it (0 to 255) as too much precision is not required.

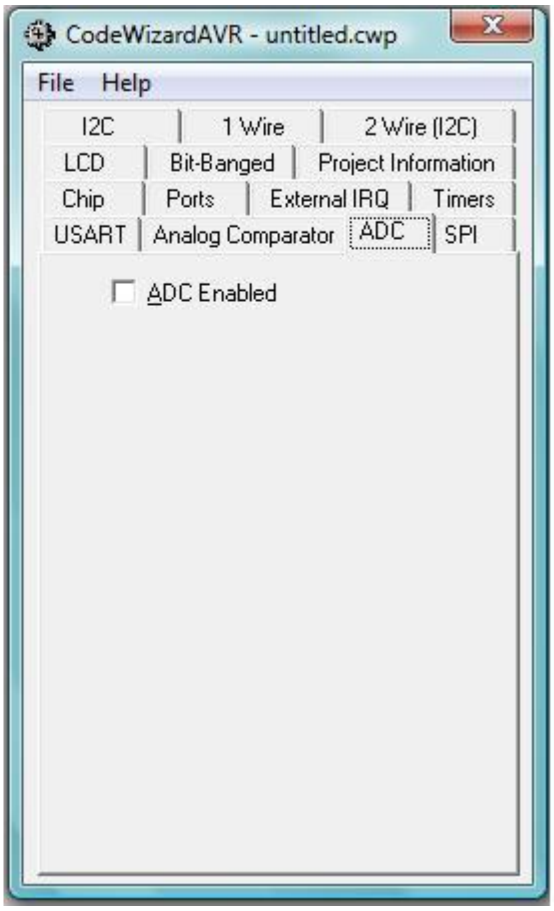
Reference voltage is the voltage to which the ADC assigns the maximum value (255 in case of 8 bit and 1023 for 10 bit). Hence, the ADC of Atmega16 divides the input analog voltage range (0V to Reference Voltage) into 1024 or 256 equal parts, depending upon whether 10 bit or 8 bit ADC is used. For example, if the reference voltage is 5V and we use 10bit ADC, 0V has digital equivalent 0, +5V is digitally 1023 and 2.5V is approximately equal to 512.

$\text{ADC} = V_{in} \times 255/V_{ref}$ (8 bit)

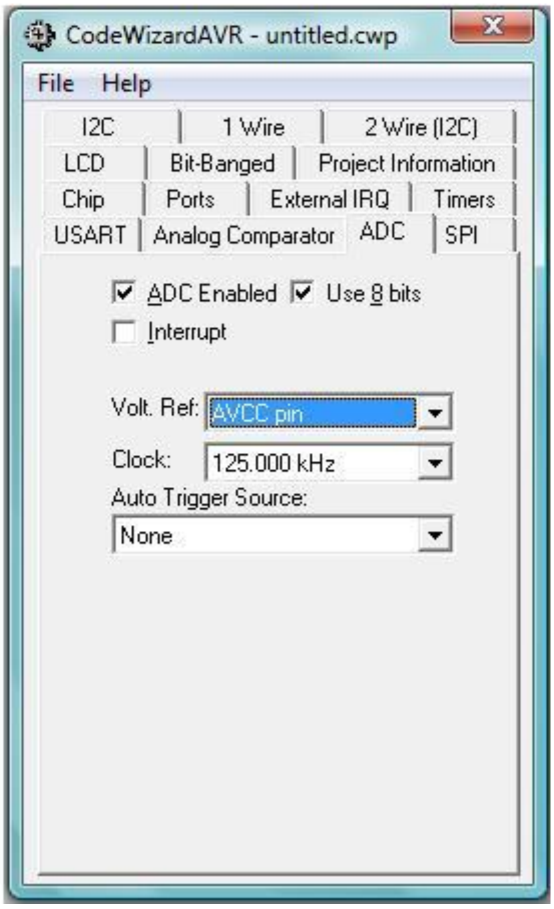
$\text{ADC} = V_{in} \times 1023/V_{ref}$ (10 bit)

11.2 Setting up Microcontroller

To enable ADC in Atmega16, click on the ADC tab in Code Wizard and enable the checkbox. You can also check “use 8 bits” as that is sufficient for our purpose and 10 bit accuracy is not required. If the input voltage ranges from 0 to less than +5V, then apply that voltage at AREF (pin 32) and select the Volt. Ref. as AREF pin. But if it ranges from 0 to +5 V, you can select the Volt. Ref. as AVCC pin itself. Keep the clock at its default value of 125 kHz and select the Auto Trigger Source as Free Running. You can also enable an interrupt function if you require.



☐ ADC Enabled



☑ ADC Enabled ☑ Use 8 bits
☐ Interrupt

Volt. Ref: AVCC pin
Clock: 125.000 kHz
Auto Trigger Source: None

11.3 Function for getting ADC

Now when you generate and save the code, all the register values are set automatically along with a function:

unsigned char read_adc(unsigned char adc_input).

This function returns the digital value of analog input at that pin of PORTA whose number is passed as parameter, e.g., if you want to know the digital value of voltage applied at PA3, and then just call the function as,

read_adc(3);

If the ADC is 8 bit, it will return a value from 0 to 255. Most probably you will need to print it on LCD. So, the code would be somewhat like

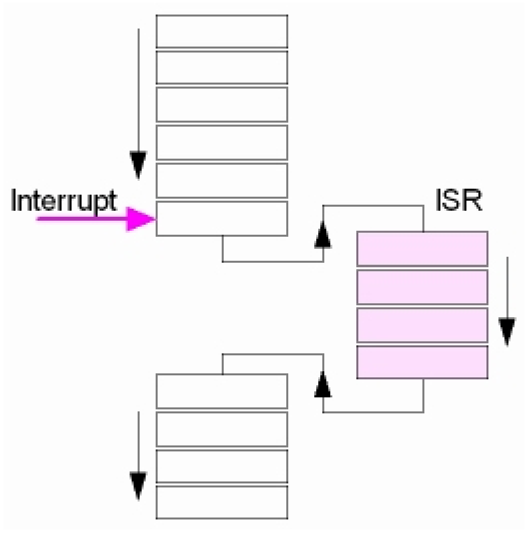
```
int a; char c[10];          // declare in the section of global variables

a=read_adc(3);
itoa(a,c);
lcd_puts(c);
```

Chapter 12 Interrupts

An interrupt is a signal that stops the current program forcing it to execute another program immediately. The interrupt does this without waiting for the current program to finish. It is unconditional and immediate which is why it is called an interrupt.

The whole point of an interrupt is that the main program can perform a task without worrying about an external event.



Example

For example if you want to read a push button connected to one pin of an input port you could read it in one of two ways either by polling it or by using interrupts.

12.1 Polling

Polling is simply reading the button input regularly. Usually you need to do some other tasks as well e.g. read an RS232 input so there will be a delay between reads of the button. As long as the delays are small compared to the speed of the input change then no button presses will be missed.

If however you had to do some long calculation then a keyboard press could be missed while the processor is busy. With polling you have to be more aware of the processor activity so that you allow enough time for each essential activity.

12.2 Hardware interrupt

By using a hardware interrupt driven button reader the calculation could proceed with all button presses captured. With the interrupt running in the background you would not have to alter the calculation function to give up processing time for button reading.

The interrupt routine obviously takes processor time – but you do not have to worry about it while constructing the calculation function.

You do have to keep the interrupt routine small compared to the processing time of the other functions - it's no good putting tons of operations into the ISR. Interrupt routines should be kept short and sweet so that the main part of the program executes correctly e.g. for lots of interrupts you need the routine to finish quickly ready for the next one.

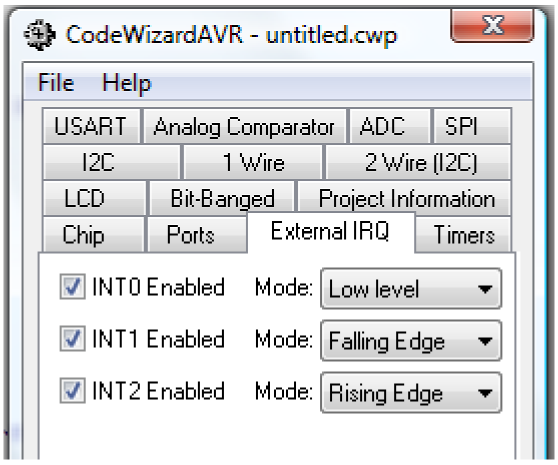
12.3 Hardware Interrupt or polling?

The benefit of the hardware interrupt is that processor time is used efficiently and not wasted polling input ports. The benefit of polling is that it is easy to do.

Another benefit of using interrupts is that in some processors you can use a wake-from-sleep interrupt. This lets the processor go into a low power mode, where only the interrupt hardware is active, which is useful if the system is running on batteries.
Hardware interrupt Common terms

Terms you might hear associated with hardware interrupts are ISR, interrupt mask, non maskable interrupt, an asynchronous event, and interrupt vector and context switching.

12.4 Setting up Hardware Interrupt in Microcontroller



INT0 Enabled ✓ Mode: Low level
INT1 Enabled ✓ Mode: Falling Edge
INT2 Enabled ✓ Mode: Rising Edge

There are 3 external interrupts in Atmega 16. They are

- INT0 : PD2, Pin 16
- INT1 : PD3, Pin 17
- INT2 : PB2, Pin 3

There are 3 modes of external Interrupts,

1. **Low Level:** In this mode interrupt occurs whenever it detects a '0' logic at INT pin. To use this, you should put an external pull up resistance to avoid interrupt every time.

2. **Falling Edge:** In this mode interrupt occurs whenever it detects a falling edge that is '1' to '0' logic change at INT pin.

3. **Rising Edge:** In this mode interrupt occurs whenever it detects a falling edge that is '0' to '1' logic change at INT pin.

12.5 Functions of Interrupt Service Routine

After generating the code, you can see the following function,

```
// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
// Place your code here

}
```

You can place your code in the function, and the code will be executed whenever interrupt occurs.

Section C

Robotics

Chapter 13 Introduction to Autonomous Robots

Any Autonomous Robot consists of following essential parts.

1. Robot Chassis and actuators

Includes wheeled or any type of chassis with all the necessary actuators fitted on the chassis to achieve desired goal. We mostly use DC geared motors as actuators.

2. Electronics

Electronics includes Sensors, motion control circuits, power management system etc.

3. Power Source

Usually battery pack consisting of Lead acid, Nickel cadmium, Nickel metal hydride or Lithium batteries is used.

4. Intelligence

This is the most important part of the autonomous robots. Usually intelligence is achieved by using Microcontroller.

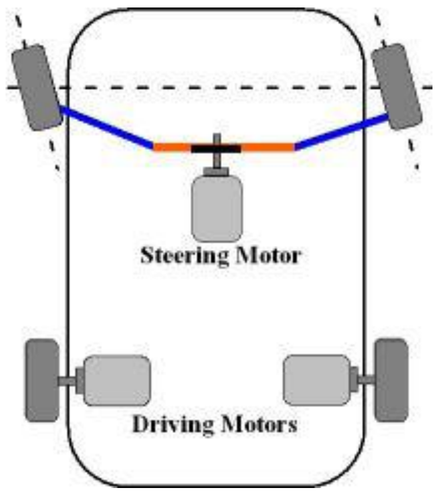
First step in making an autonomous robot is to chalk out what tasks we are expecting the robot to perform. After gauging these we get a vague idea about the design and appearance of the robot.

13.1 Robot Chassis Designing

Selecting the Drive Mechanism for the wheeled motion:

13.1.1 Robot with steering wheel:

- Power for motion is provided by back wheels and turning is achieved using front wheels.
- This scheme is similar to that of cars.



Advantages:

1. When path to be followed is straight in nature with curved turns this configuration gives fastest speed and graceful path following.

2. Don't need to modify left or right wheels velocity to follow the path. This is very advantageous when we want precision velocity control. In this case back wheels take care of velocity control and front wheels take care of direction control.

Disadvantages:

1. It will not able to take very sharp turns. Hence it is difficult to move robot on the grid of lines.

2. Somewhat difficult and expensive to make.

3. Front wheels will need position feedback to control turning control.

13.1.2 Robot with differential drive:

- A method of controlling a robot where the left and right wheels are powered independently.
- The Three Wheel Differential drive uses two motors and a caster or an omni-directional wheel easiest to design and program.

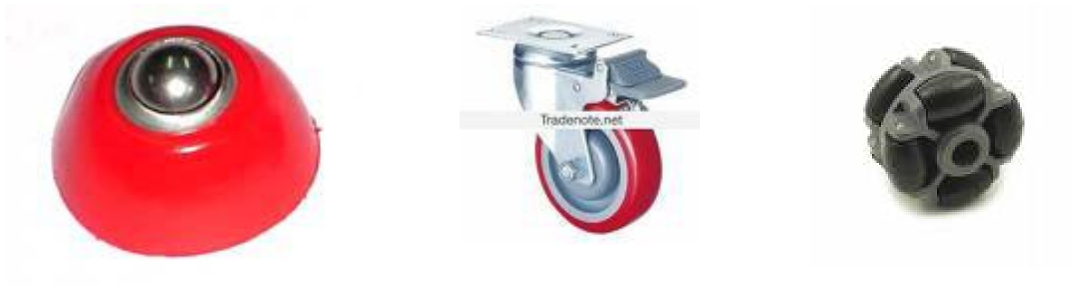
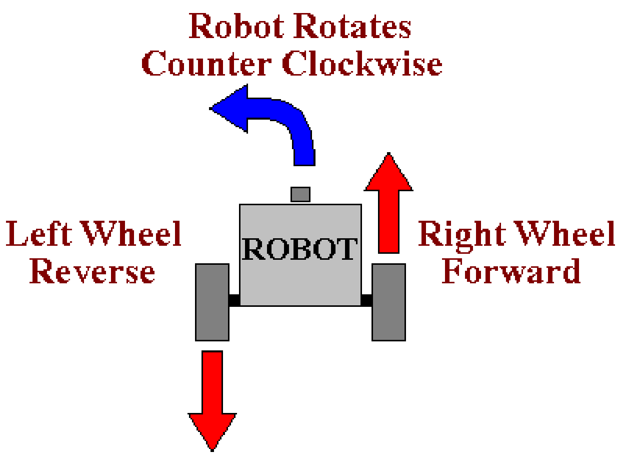
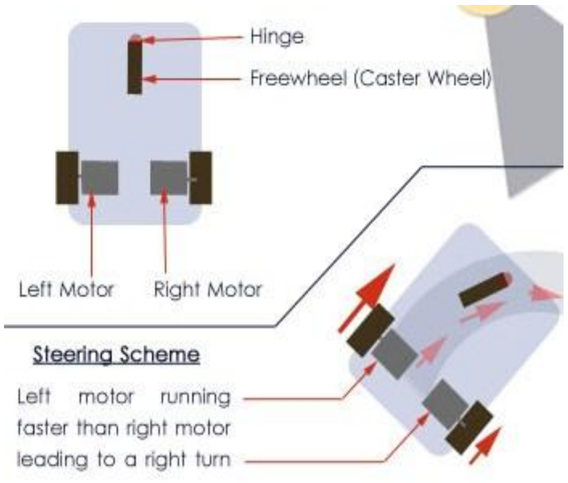


Figure 23: Ball bearing caster, wheel based swivel caster and omni directional wheel

- The radius and centre of rotation can be varied by the varying the relative speed of rotation between the two motors.




Hinge
Freewheel (Caster Wheel)
Left Motor
Right Motor
Steering Scheme
Left motor running faster than right motor leading to a right turn

- Rotating the wheels in different directions provides a sharp turn.
- For a smooth turn, rotate the wheels in the same direction but with different speeds. Greater the difference in speeds, smaller the radius of rotation.

Advantages:

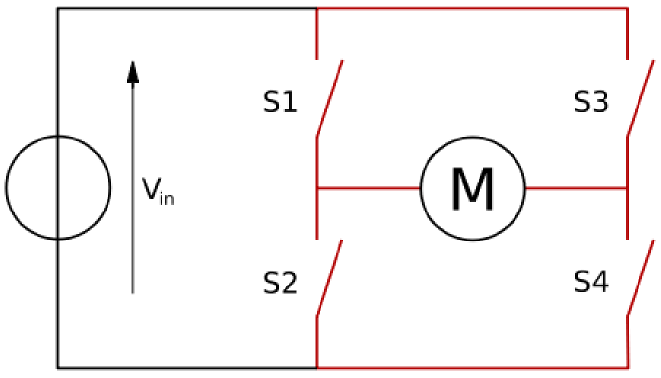
1. Zero turning radius achievable.
2. Easy to move when path to be followed is contoured and zigzag in nature. E.g., navigating along the maze of lines.

Disadvantages:

1. If we want to move along curved path we have to control left and right motor's velocity independently. Hence precision velocity control becomes difficult as actual velocity of the robot will be average of the both wheels.

14.1 H- Bridge:

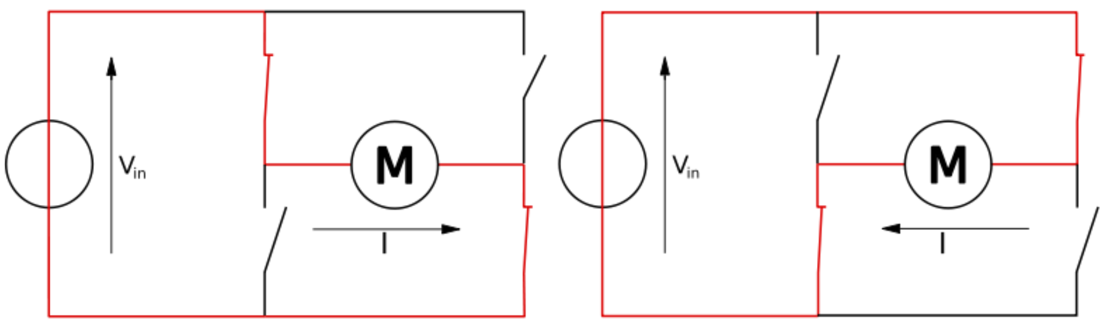
- It is an electronic circuit which enables a voltage to be applied across a load in either direction.
- It allows a circuit full control over a standard electric DC motor. That is, with an H-bridge, a microcontroller, logic chip, or remote control can electronically command the motor to go forward, reverse, brake, and coast.
- H-bridges are available as integrated circuits, or can be built from discrete components.
- A "double pole double throw" relay can generally achieve the same electrical functionality as an H-bridge, but an H-bridge would be preferable where a smaller physical size is needed, high speed switching, low driving voltage, or where the wearing out of mechanical parts is undesirable.
- The term "H-bridge" is derived from the typical graphical representation of such a circuit, which is built with four switches, either solid-state (eg, L293/ L298) or mechanical (eg, relays).



S1	S2	S3	S4	Result
1	0	0	1	Motor rotates in one direction
0	1	1	0	Motor rotates in opposite direction
0	0	0	0	Motor free runs (coasts)
0	1	0	1	Motor brakes
1	0	1	0	Motor brakes

Structure of an H-bridge (highlighted in red)

- To power the motor, you turn on two switches that are diagonally opposed.



The two basic states of an H-bridge.

14.2 Motor Driver ICs: L293/L293D and L298

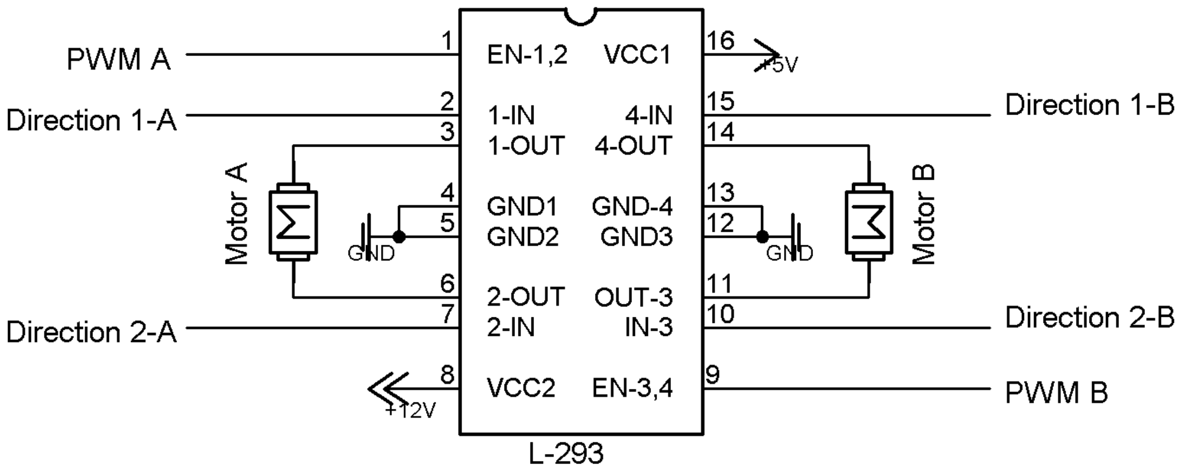
Figure 25: L293D

Figure 24: L298

- The current provided by the MCU is of the order of 5mA and that required by a motor is ~500mA. Hence, motor can't be controlled directly by MCU and we need an interface between the MCU and the motor.

- A Motor Driver IC like L293D or L298 is used for this purpose which has two H-bridge drivers. Hence, each IC can drive two motors.

- Note that a motor driver does not amplify the current; it only acts as a switch (An H bridge is nothing but 4 switches).



- Drivers are enabled in pairs, with drivers 1 and 2 being enabled by the Enable pin. When an enable input is high (logic 1 or +5V), the associated drivers are enabled and their outputs are active and in phase with their inputs.

- When the enable pin is low, the output is neither high nor low (disconnected), irrespective of the input.

- Direction of the motor is controlled by asserting one of the inputs to motor to be high (logic 1) and the other to be low (logic 0).

- To move the motor in opposite direction just interchange the logic applied to the two inputs of the motors.

- Asserting both inputs to logic high or logic low will stop the motor.
- Resistance of our motors is about 26 ohms i.e. its short circuit current will be around. 0.46Amp which is below the maximum current limit.
- It is always better to use high capacitance (~1000µF) in the output line of a motor driver which acts as a small battery at times of current surges and hence improves battery life.
- **Difference between L293 and L293D:** Output current per channel = 1A for L293 and 600mA for L293D.

14.2.1 Difference between L293 and L298:

- L293 is quadruple half-H driver while L298 is dual full-H driver, i.e, in L293 all four input-output lines are independent while in L298, a half H driver cannot be used independently, only full H driver has to be used.
- Output current per channel = 1A for L293 and 2A for L298. Hence, heat sink is provided in L298.
- Protective Diodes against back EMF are provided internally in L293D but must be provided externally in L298.

14.2.2 Speed Control:

- To control motor speed we can use pulse width modulation (PWM), applied to the enable pins of L293 driver.
- PWM is the scheme in which the duty cycle of a square wave output from the microcontroller is varied to provide a varying average DC output.
- What actually happens by applying a PWM pulse is that the motor is switched ON and OFF at a given frequency. In this way, the motor reacts to the time average of the power supply.

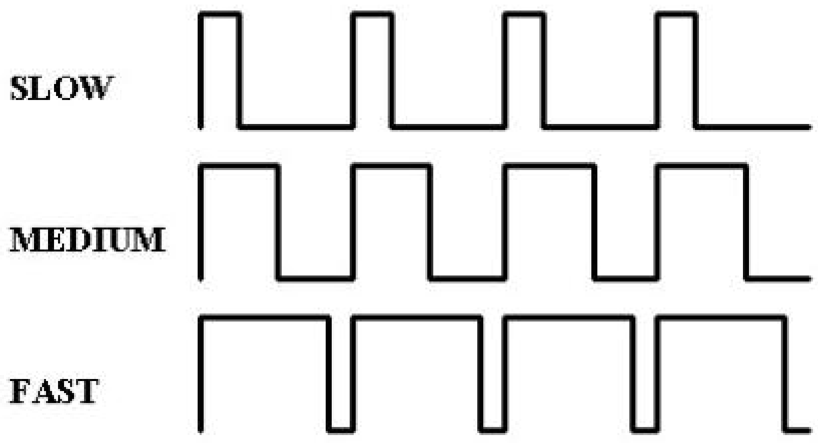
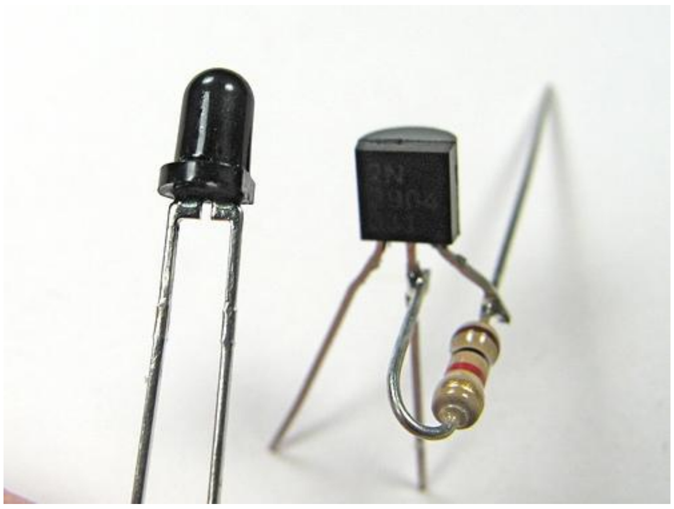
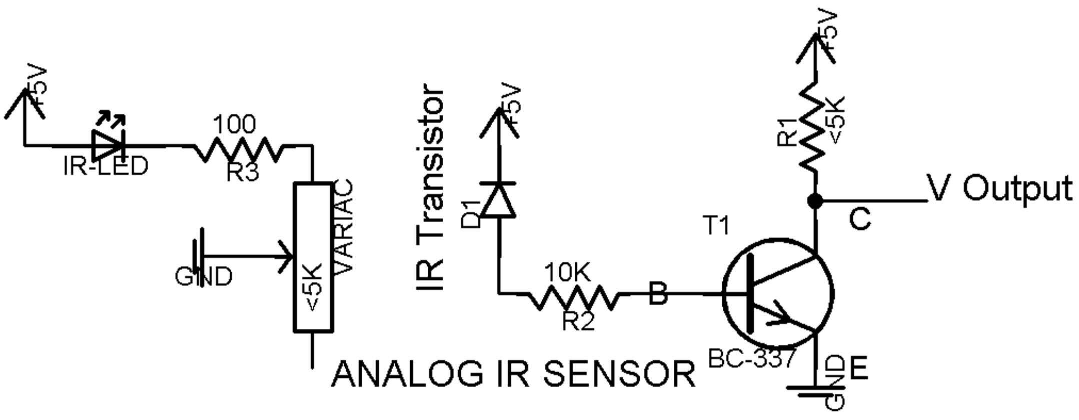


Figure 26: Velocity control of motor using PWM

Chapter 15 Sensors

15.1 Analog Sensor





The IR analog sensor consists of:

Transmitter: An Infra Red emitting diode

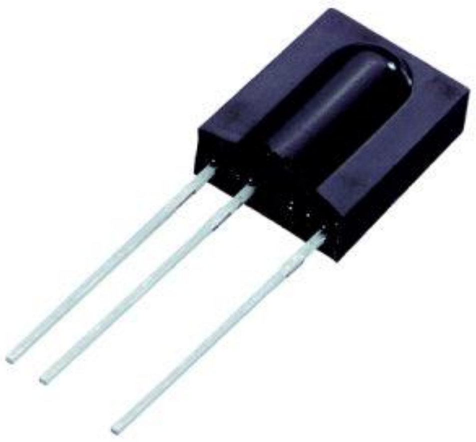
Receiver: A Phototransistor (also referred as photodiode)

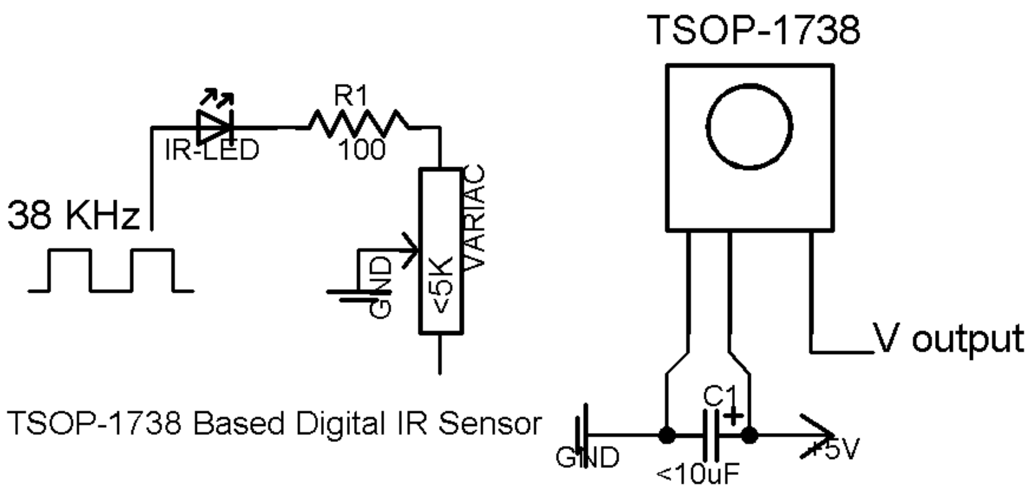
It is better to keep R2 as a variac to vary the sensitivity.

The output varies from 0V to 5V depending upon the amount of IR it receives, hence the name analog.

The output can be taken to a microcontroller either to its ADC (Analog to Digital Converter) or LM 339 can be used as a comparator.

15.2 Digital IR Sensor - TSOP Sensor





- TSOP 1738 Sensor is a digital IR Sensor; It is logic 1 (+5V) when IR below a threshold is falling on it and logic 0 (0V) when it receives IR above threshold.

- It does not respond to any stray IR, it only responds to IR falling on it at a pulse rate of 38 KHz. Hence we have a major advantage of high immunity against ambient light.

- No comparator is required and the range of the sensor can be varied by varying the intensity of the IR emitting diode (the variac in figure).

Chapter 16 References

[1] Atmega 16 Datasheet

[2] www.wikipedia.org

[3] http://www.cmosexod.com/micro_uart.htm

[4] http://www.best-microcontroller-projects.com/hardware-interrupt.html

[5] http://www.avrtutor.com/tutorial/interrupt/interrupts.php