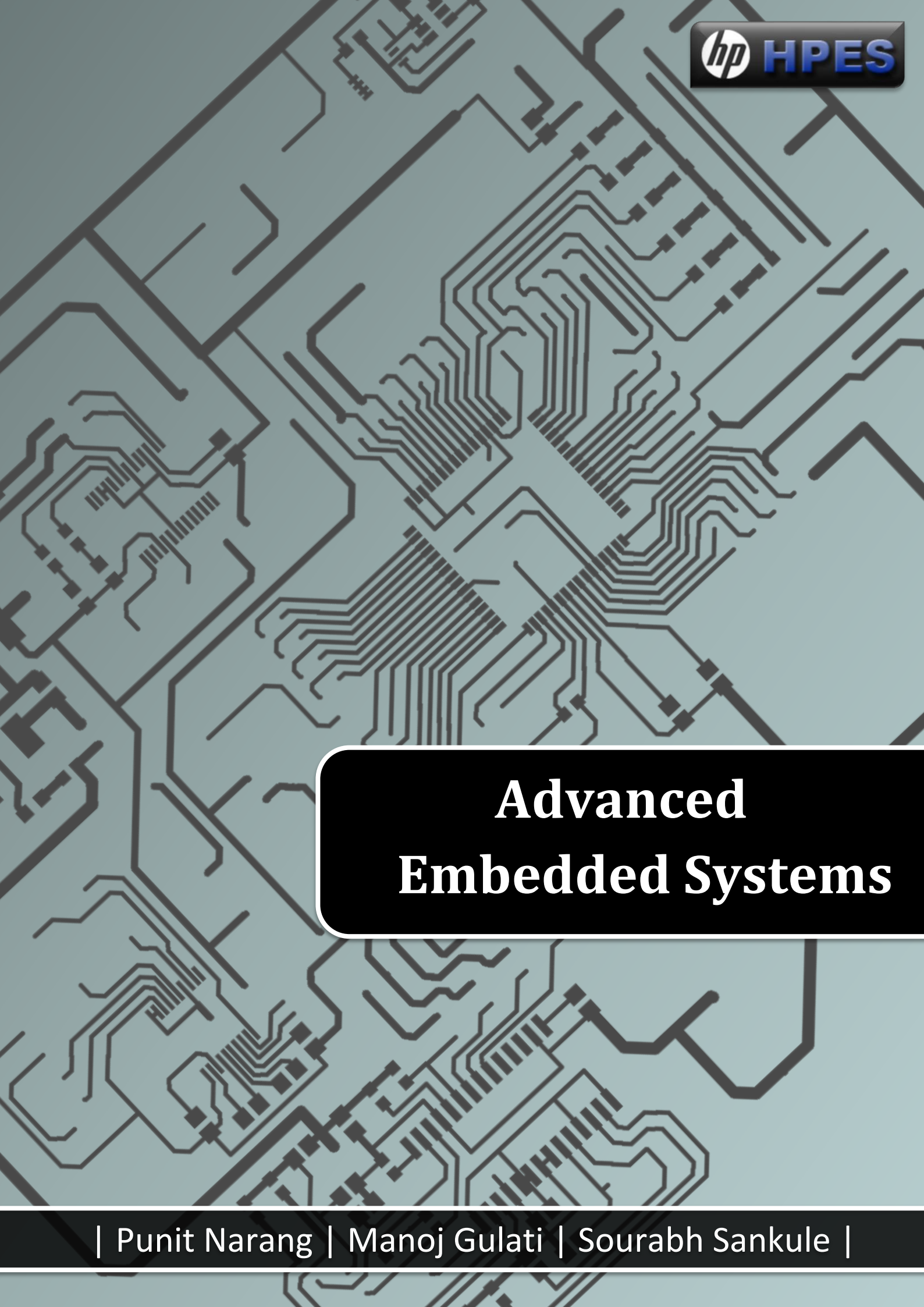



# Advanced Embedded Systems

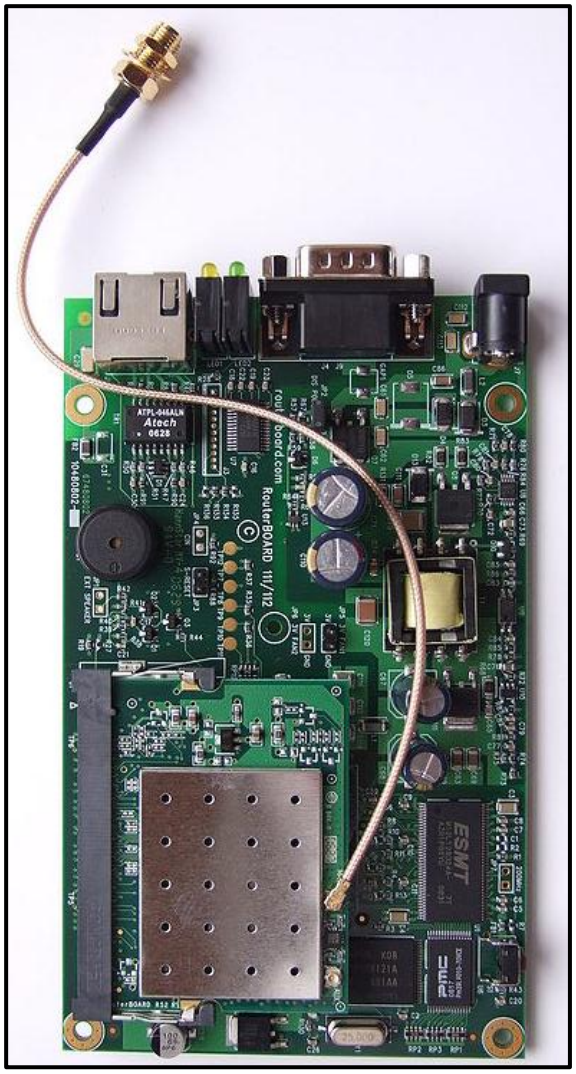
| Punit Narang | Manoj Gulati | Sourabh Sankule |

# Contents

# Chapter 1 - Introduction to Embedded Electronics

## 1.1 Embedded System

An embedded system is a computer system designed to do one or a few dedicated and/or specific functions often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today.



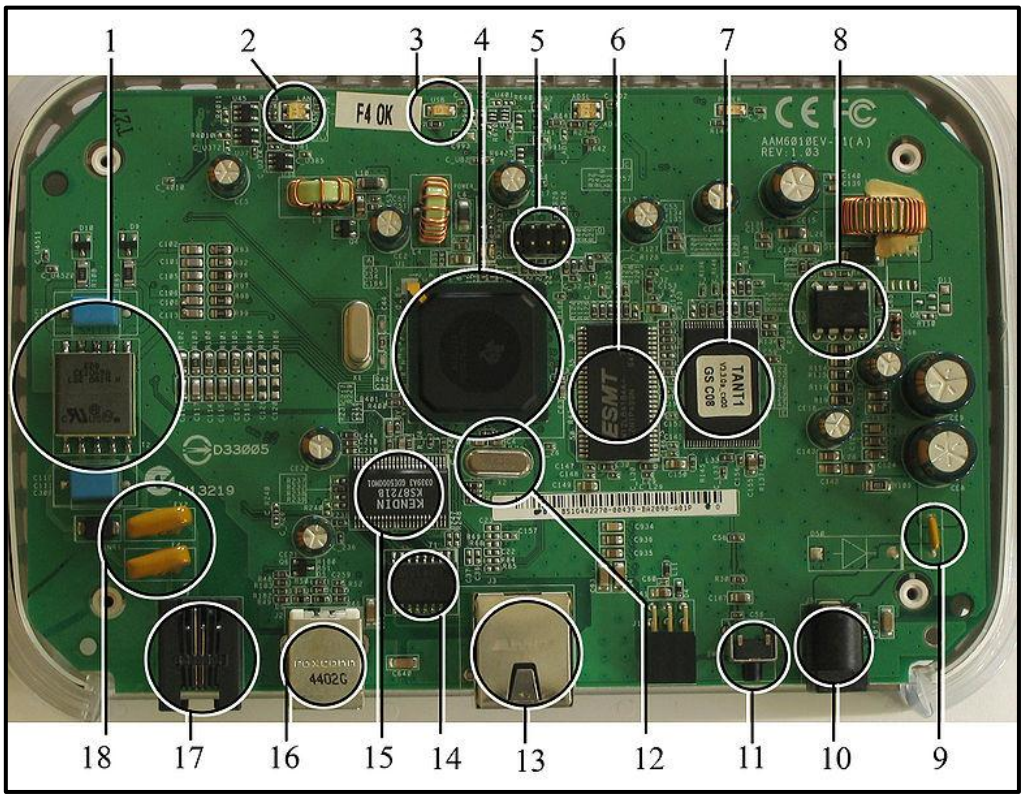

Embedded systems are controlled by one or more main processing cores that are typically either microcontrollers or digital signal processors (DSP). The key characteristic, however, is being dedicated to handle a particular task. They may require very powerful processors and extensive communication, for example air traffic control systems may usefully be viewed as embedded, even though they involve mainframe computers and dedicated regional and national networks between airports and radar sites (each radar probably includes one or more embedded systems of its own).

Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance. Some embedded systems are mass-produced, benefiting from economies of scale.

Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, or the systems controlling nuclear power plants. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

In general, "embedded system" is not a strictly definable term, as most systems have some element of extensibility or programmability. For example, handheld computers share some elements with embedded systems such as the operating systems and microprocessors that power them, but they allow different applications to be loaded and peripherals to be connected. Moreover, even systems that do not expose programmability as a primary feature generally need to support software updates. On a continuum from "general purpose" to "embedded", large application systems will have subcomponents at most points even if the system as a whole is "designed to perform one or a few dedicated functions", and is thus appropriate to call "embedded".

### 1.1.1 Example of Embedded System




Picture of the internals of an ADSL modem/router. A modern example of an embedded system. This image shows the parts found inside a Netgear DG632 ADSL Modem/router. It acts as a router between an ethernet port and an ADSL broadband internet connection, and provides typical home router features, such as DHCP.

The labeled parts are as follows:

1. Telephone decoupling electronics (for ADSL).
2. Multicolour LED (displaying network status).
3. Single colour LED (displaying USB status).
4. Main processor, a TNETD7300GDU, a member of Texas Instruments' AR7 product line.
5. JTAG (Joint Test Action Group) test and programming port.
6. RAM, a single ESMT M12L64164A 8 MB chip.
7. Flash memory, obscured by sticker.
8. Power supply regulator.
9. Main power supply fuse.
10. Power connector.
11. Reset button.
12. Quartz crystal.
13. Ethernet port.
14. Ethernet transformer, Delta LF8505.
15. KS8721B Ethernet PHY transmitter receiver.
16. USB port.
17. Telephone (RJ11) port.
18. Telephone connector fuses.

### 1.1.2 Characteristics

1. Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks. Some also have real-time performance constraints that must be met, for reasons such as safety and usability; others may have low or no performance requirements, allowing the system hardware to be simplified to reduce costs.

2. Embedded systems are not always standalone devices. Many embedded systems consist of small, computerized parts within a larger device that serves a more general purpose. For example, the Gibson Robot Guitar features an embedded system for tuning the strings, but the overall purpose of the Robot Guitar is, of course, to play music. Similarly, an embedded system in an automobile provides a specific function as a subsystem of the car itself.

3. The program instructions written for embedded systems are referred to as firmware, and are stored in read-only memory or Flash memory chips. They run with limited computer hardware resources: little memory, small or non-existent keyboard and/or screen.

### 1.1.3 User interface

Embedded systems range from no user interface at all — dedicated only to one task — to complex graphical user interfaces that resemble modern computer desktop operating systems. Simple embedded devices use buttons, LEDs, graphic or character LCDs (for example popular HD44780 LCD) with a simple menu system.

More sophisticated devices use graphical screen with touch sensing or screen-edge buttons provide flexibility while minimizing space used: the meaning of the buttons can change with the screen, and selection involves the natural behavior of pointing at what's desired. Handheld systems often have a screen with a "joystick button" for a pointing device.

Some systems provide user interface remotely with the help of a serial (e.g. RS-232, USB, $I^2C$, etc.) or network (e.g. Ethernet) connection. In spite of the potentially necessary proprietary client software and/or specialist cables that are needed, this approach usually gives a lot of advantages: extends the capabilities of embedded system, avoids the cost of a display, simplifies BSP, allows to build rich user interface on the PC. A good example of this is the combination of an embedded web server running on an embedded device (such as an IP camera) or a network routers. The user interface is displayed in a web browser on a PC connected to the device, therefore needing no bespoke software to be installed.

### 1.1.4 Processors in embedded systems

Secondly, Embedded processors can be broken into two broad categories: ordinary microprocessors (µP) and microcontrollers (µC), which have many more peripherals on chip, reducing cost and size. Contrasting to the personal computer and server markets, a fairly large number of basic CPU architectures are used; there are Von Neumann as well as various degrees of Harvard architectures, RISC as well as non-RISC and VLIW; word lengths vary from 4-bit to 64-bits and beyond (mainly in DSP processors) although the most typical remain 8/16-bit. Most architectures come in a large number of different variants and shapes, many of which are also manufactured by several different companies.

A long but still not exhaustive list of common architectures are: 65816, 65C02, 68HC08, 68HC11, 68k, 78K0R/78K0, 8051, ARM, AVR, AVR32, Blackfin, C167, Coldfire, COP8, Cortus APS3, eZ8, eZ80, FR-V, H8, HT48, M16C, M32C, MIPS, MSP430, PIC, PowerPC, R8C, RL78, SHARC, SPARC, ST6, SuperH, TLCS-47, TLCS-870, TLCS-900, TriCore, V850, x86, XE8000, Z80, AsAP etc.

### 1.1.5 Peripherals

Embedded Systems talk with the outside world via peripherals, such as:

- Serial Communication Interfaces (SCI): RS-232, RS-422, RS-485 etc.
- Synchronous Serial Communication Interface: I2C, SPI, SSC and ESSI (Enhanced Synchronous Serial Interface)
- Universal Serial Bus (USB)
- Multi Media Cards (SD Cards, Compact Flash etc.)
- Networks: Ethernet, LonWorks, etc.
- Fieldbuses: CAN-Bus, LIN-Bus, PROFIBUS, etc.
- Timers: PLL(s), Capture/Compare and Time Processing Units
- Discrete IO: aka General Purpose Input/Output (GPIO)
- Analog to Digital/Digital to Analog (ADC/DAC)
- Debugging: JTAG, ISP, ICSP, BDM Port, BITP, and DP9 ports.

## 1.2 Microcontrollers

### 1.2.1 What is a Microcontroller?

A Microcontroller is a programmable digital processor with necessary peripherals. Both microcontrollers and microprocessors are complex sequential digital circuits meant to carry out job according to the program / instructions. Sometimes analog input/output interface makes a part of microcontroller circuit of mixed mode(both analog and digital nature).

### 1.2.2 Microcontrollers vs. Microprocessors

- A microprocessor requires an external memory for program/data storage. Instruction execution requires movement of data from the external memory to the microprocessor or vice versa. Usually, microprocessors have good computing power and they have higher clock speed to facilitate faster computation.
- A microcontroller has required on-chip memory with associated peripherals. A microcontroller can be thought of a microprocessor with inbuilt peripherals.
- A microcontroller does not require much additional interfacing ICs for operation and it functions as a stand-alone system. The operation of a microcontroller is multipurpose, just like a Swiss knife.
- Microcontrollers are also called embedded controllers. A microcontroller clock speed is limited only to a few tens of MHz. Microcontrollers are numerous and many of them are application specific.

### 1.2.3 Development/Classification of microcontrollers (Invisible)

Microcontrollers have gone through a silent evolution (invisible). The evolution can be rightly termed as silent as the impact or application of a microcontroller is not well known to a common user, although microcontroller technology has undergone significant change since early 1970's. Development of some popular microcontrollers is given as follows.

| | | |
|---|---|---|
| Intel 4004 | 4 bit (2300 PMOS trans, 108 kHz) | 1971 |
| Intel 8048 | 8 bit | 1976 |
| Intel 8031 | 8 bit (ROM-less) | . |
| Intel 8051 | 8 bit (Mask ROM) | 1980 |
| Microchip PIC16C64 | 8 bit | 1985 |
| Motorola 68HC11 | 8 bit (on chip ADC) | . |
| Intel 80C196 | 16 bit | 1982 |
| Atmel AT89C51 | 8 bit (Flash memory) | . |
| Microchip PIC 16F877 | 8 bit (Flash memory + ADC) | . |

### 1.2.4 Development of microprocessors (Visible)

Microprocessors have undergone significant evolution over the past four decades. This development is clearly perceptible to a common user, especially, in terms of phenomenal growth in capabilities of personal computers. Development of some of the microprocessors can be given as follows.

| | | |
|---|---|---|
| Intel 4004 | 4 bit (2300 PMOS transistors) | 1971 |
| Intel 8080<br>8085 | 8 bit (NMOS)<br>8 bit | 1974 |
| Intel 8088<br>8086 | 16 bit<br>16 bit | 1978 |
| Intel 80186<br>80286 | 16 bit<br>16 bit | 1982 |
| Intel 80386 | 32 bit (275000 transistors) | 1985 |
| Intel 80486 SX<br>DX | 32 bit<br>32 bit (built in floating point unit) | 1989 |
| Intel 80586 I<br>MMX<br>Celeron II<br>III<br>IV | 64 bit | 1993<br>1997<br>1999<br>2000 |
| Z-80 (Zilog) | 8 bit | 1976 |
| Motorola Power PC 601<br>602<br>603 | 32-bit | 1993<br>1995 |

We use more number of microcontrollers compared to microprocessors. Microprocessors are primarily used for computational purpose, whereas microcontrollers find wide application in devices needing real time processing / control.
Applications of microcontrollers are numerous. Starting from domestic applications such as in washing machines, TVs, air-conditioners, microcontrollers are used in automobiles, process control industries, cell phones, electrical drives, robotics and in space applications.

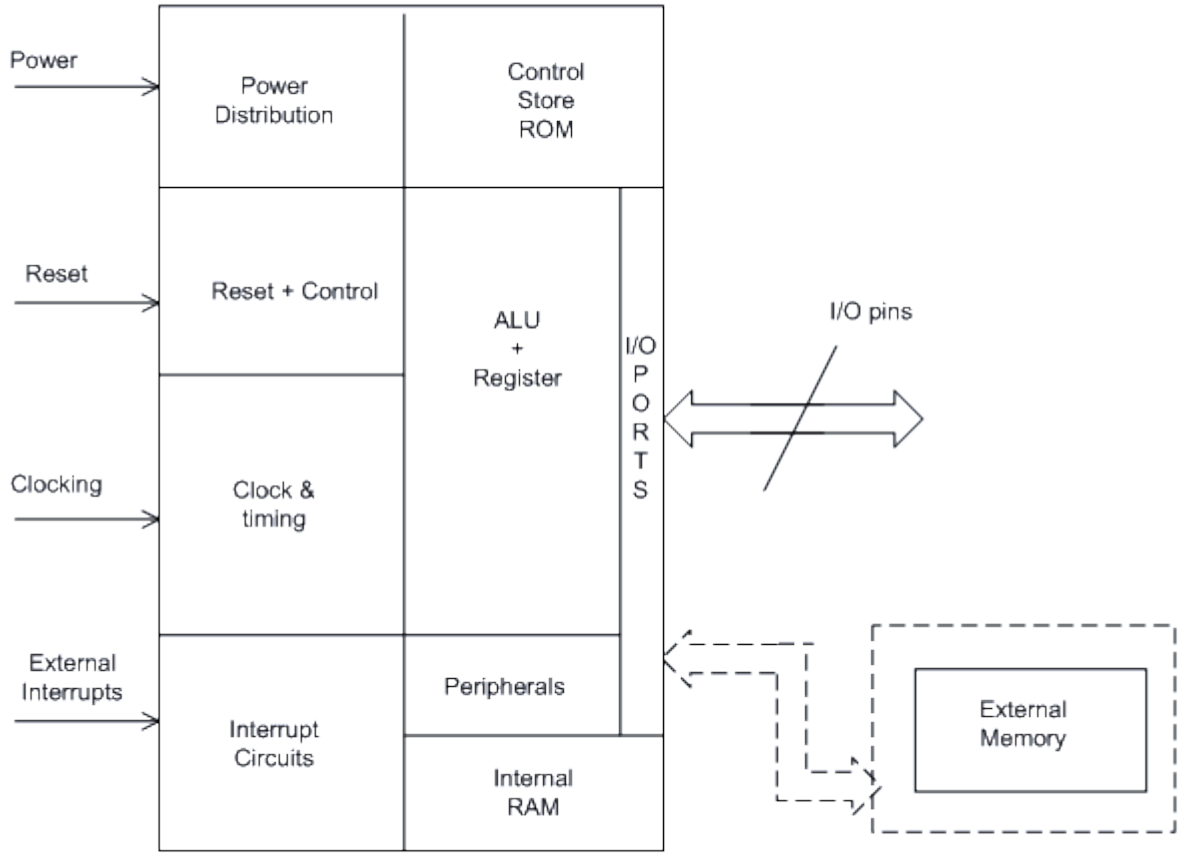
**Broad Classification of different microcontroller chips could be as follows:**

- Embedded (Self -Contained) 8 - bit Microcontroller
- 16 to 32 Microcontrollers
- Digital Signal Processors

**Features of Modern Microcontrollers**

- Built-in Monitor Program
- Built-in Program Memory
- Interrupts
- Analog I/O
- Serial I/O
- Facility to Interface External Memory
- Timers

### 1.2.5 Internal Structure of a Microcontroller



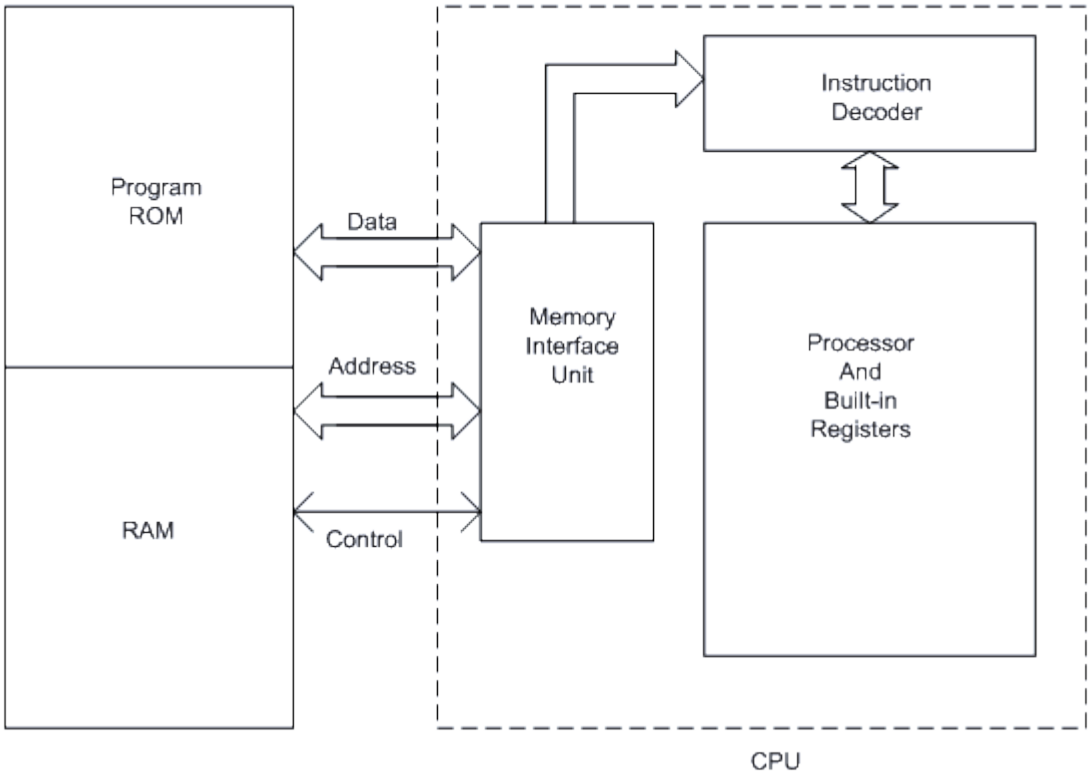

At times, a microcontroller can have external memory also (if there is no internal memory or extra memory interface is required). Early microcontrollers were manufactured using bipolar or NMOS technologies. Most modern microcontrollers are manufactured with CMOS technology, which leads to reduction in size and power loss. Current drawn by the IC is also reduced considerably from 10mA to a few micro Amperes in sleep mode (for a microcontroller running typically at a clock speed of 20MHz).

### 1.2.6 Harvard vs. Princeton Architecture

Many years ago, in the late 1940's, the US Government asked Harvard and Princeton universities to come up with a computer architecture to be used in computing distances of Naval artillery shell for defence applications. Princeton suggested computer architecture with a single memory interface. It is **also known as Von Neumann architecture** after the name of the chief scientist of the project in Princeton University John Von Neumann (1903 - 1957 Born in Budapest, Hungary).

Harvard suggested a computer with two different memory interfaces, one for the data / variables and the other for program / instructions. Although Princeton architecture was accepted for simplicity and ease of implementation, Harvard architecture became popular later, due to the parallelism of instruction execution.

### 1.2.7 Princeton Architecture (Single memory interface)

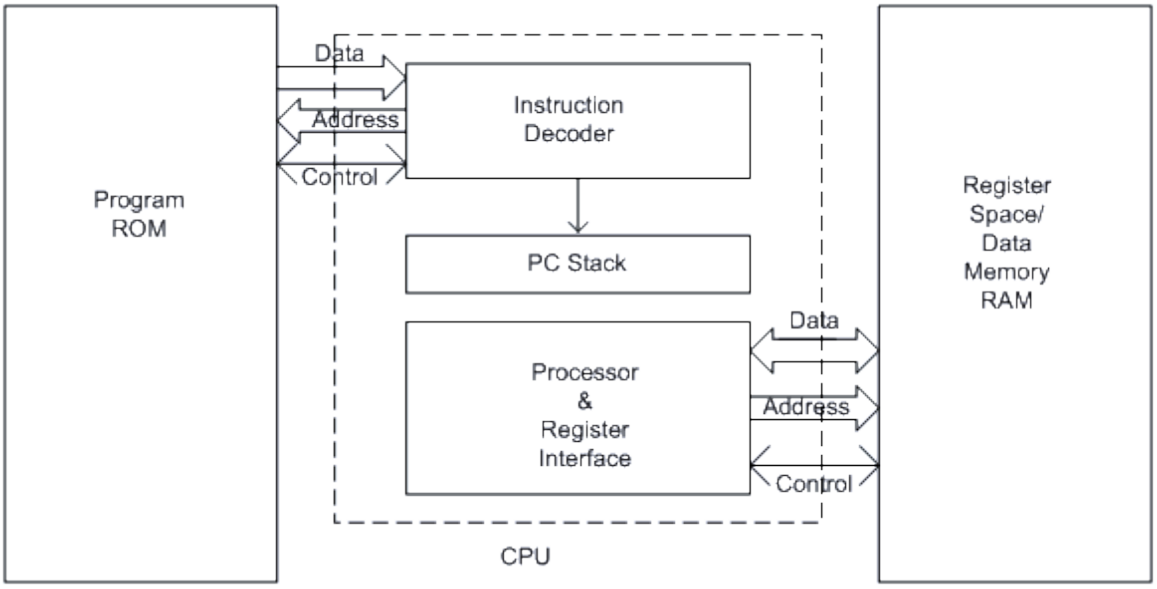



**Example:**

An instruction "Read a data byte from memory and store it in the accumulator" is executed as follows: -

Cycle 1 - Read Instruction
Cycle 2 - Read Data out of RAM and put into Accumulator

### 1.2.8 Harvard Architecture (Separate Program and Data Memory interfaces)




The same instruction (as shown under Princeton Architecture) would be executed as follows:

Cycle 1
- Complete previous instruction
- Read the "Move Data to Accumulator" instruction

Cycle 2
- Execute "Move Data to Accumulator" instruction
- Read next instruction

Hence each instruction is effectively executed in one instruction cycle, except for the ones that modify the content of the program counter. For example, the "jump" (or call) instructions takes 2 cycles. Thus, due to parallelism, Harvard architecture executes more instructions in a given time compared to Princeton Architecture.

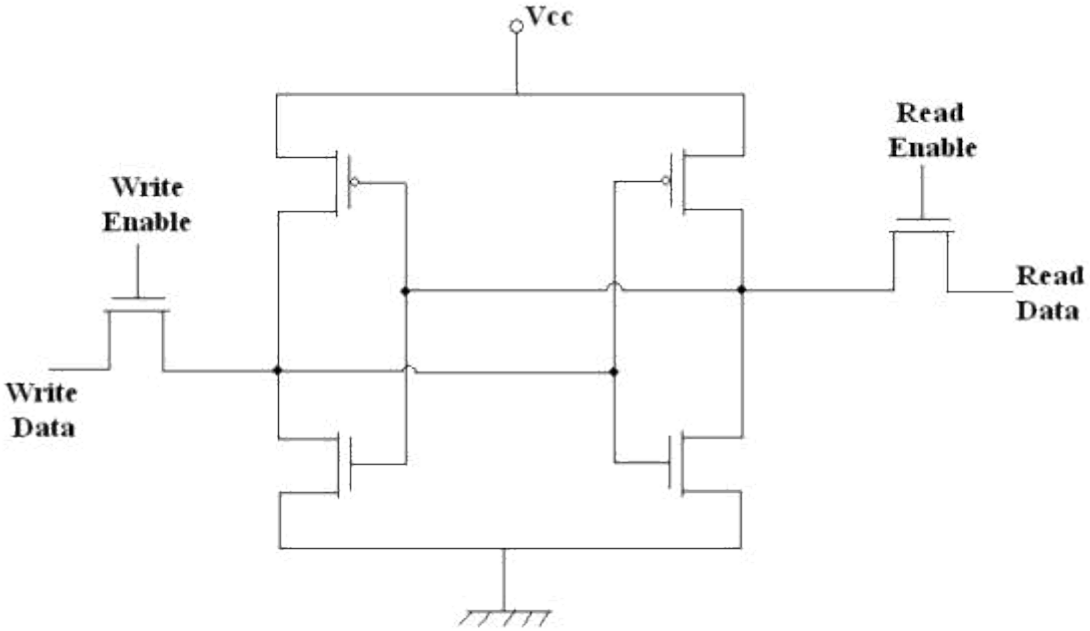
## 1.3 Data Memory Organization

Data memory can be classified into the following categories

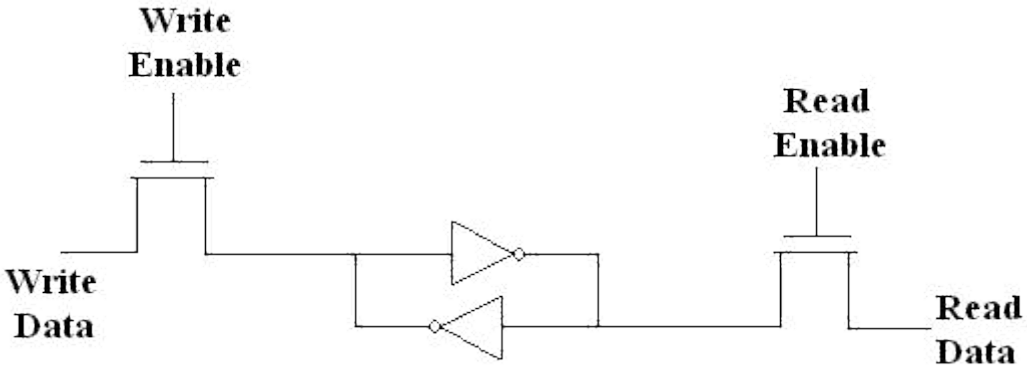
- Bits
- Registers
- Variable RAM
- Program counter stack

Microcontroller can have ability to perform manipulation of individual bits in certain registers (bit manipulation). This is a unique feature of a microcontroller, not available in a microprocessor.

Eight bits make a byte. Memory bytes are known as file registers. Registers are some special RAM locations that can be accessed by the processor very easily.




Static RAM (SRAM) memory cell




SRAM memory cell equivalent

Processor stacks store/save the data in a simple way during program execution. Processor stack is a part of RAM area where the data is saved in a Last In First Out (LIFO) fashion just like a stack of paper on a table. Data is stored by executing a 'push' instruction and data is read out using a 'pop' instruction.

**I/O Registers:** In addition to the Data memory, some special purpose registers are required that are used in input/output and control operations. These registers are called I/O registers. These are important for microcontroller peripheral interface and control applications.

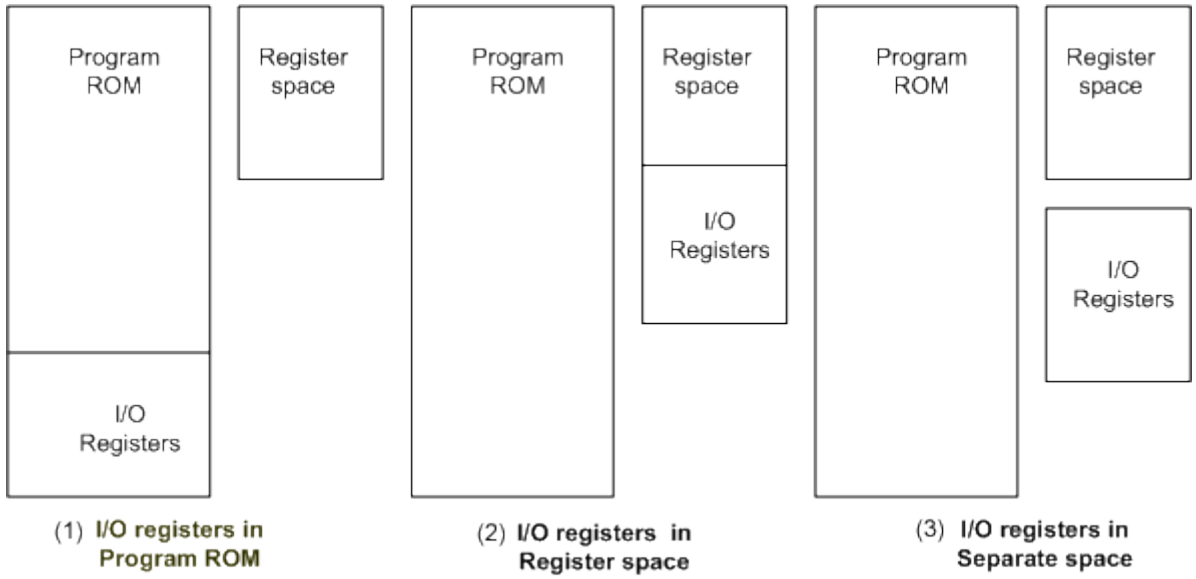
**Hardware interface registers (I/O Space)**

As we already know a microcontroller has some embedded peripherals and I/O devices. The data transfer to these devices takes place through I/O registers. In a microprocessor, input /output (I/O) devices are externally interfaced and are mapped either to memory address (memory mapped I/O) or a separate I/O address space (I/O mapped I/O).

### 1.3.1 I/O Registers space in Harvard Architecture

These are the following options available for I/O register space in Harvard Architecture.

- I/O registers in program ROM.
- I/O registers in register space (Data Memory area).
- I/O registers in separate space.




Organisation of I/O registers in Harward Architecture

The first option is somewhat difficult to implement as there is no means to write to program ROM area. It is also complicated to have a separate I/O space as shown in (3). Hence the second option where I/O registers are placed in the register space is widely used.

## 1.4 CISC (Complex Instruction Set Computer) Processor Architecture

A 1960s architect who took the CISC (Complex Instruction Set Computer) approach was an architect who could build a computing system that would utilize the smallest amount of assembly code possible. Armed with fresh innovation in a world of growing technological advancement at his disposal, and the Modus operandi of reducing the level of code, a CISC architect would strive to build as much of the "coding" as possible into the computer's hardware itself. An "Intelligent" and "logical" processor hardware system which could "understand" high-level instructions would have huge cost-cutting implications by enabling the use of minimal lines of code to achieve maximum computer functionality and complete any computing task.

A CISC system would contain a Microprocessor instruction set so that each single instruction can execute several low-level operations such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction. For a specific task, a CISC processor would come prepared with a specific instruction, e.g. "ADD".

Take a look at a working example to see how powerful and economical this system was:

**ADD 2:3**

- First of all, two values are loaded into separate register
- Then, the operands are added in the exception unit
- Finally, the sum is stored in the associated register

At the very heart of CISC is a set "complex" instruction like ADD. The computer's memory banks are directly operated on, thus making the loading or storing functions redundant. ADD is similar to what a programmer in C++ or any other high-level language would code. Takes control over thousands or millions of transistors, CPU etc. One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Micro program instruction sets can be written to match high-level languages and the compiler does not have to be as complicated. Minimal lines of coding must also have reduced the probability of errors in the code, thus reducing cost and debugging-time. In addition, small code sizes could be stored easily to enable a frugal use of RAM.

CISC seemed like such a natural and intuitive system at the time and it didn't even have a name. The term was retroactively coined in contrast to reduced instruction set computer computers of the 1970s (although we also have examples of pre-RISC systems of the 1960s) which eventually brought CISC (Complex Instruction Set Computer) to its knees and forced CISC-philes to defend and debate with their adversaries.

## 1.5 RISC (Reduced Instruction Set Computer) Architecture Design

The RISC-brigade strive for an instruction set reduced both in size and complexity of addressing modes which they argue enables easier implementation, greater instruction level parallelism, and more efficient compilers. A RISC architect tries to keep the instruction set as simple as possible so that a job can be executed within one clock cycle. Because all of the instructions execute

in a uniform amount of time (i.e. one clock), pipelining is possible. Rather than using an "intelligent" hard-coded CPU hardware system, there is more emphasis on the software. Every slight small hardware job can be managed and customized by the software used.

So, to add two integers,

- LOAD A, 2:3
- LOAD B, 5:2
- ADD A, B
- STORE 2:3, A

"LOAD" moves data from the memory bank to a register
"ADD" finds the sum of two operands located within the registers
"STORE" moves data from a register to the memory banks

They support only register-to-register operations and a few simple addressing modes. In using three instructions codes to achieve only one task, larger amounts of RAM would be needed to store the assembly code, and the compiler would need to do more work to convert that code to a lower-level form. However, this is balanced by the more economical use of registers by means of a set of "reduced instructions" which incorporate a reduced amount of transistors; and as already mentioned, by maximizing the number of instructions per program, the number of clock cycles per second is reduced. So, by using a number of small instructions rather than one "large" instruction, the amount of actual work done by the machine is reduced.

If there were any advantages in this system, the RISC people certainly had their work cut-out for them at first. RISC chips weren't in wide-circulation until the 1970s. Much of the software available at the time was designed for CISC machines. It would have been a commercial risk to actually start mass-producing this technology.

## 1.6 RISC & CISC Architecture in Today's Computer Systems

In the early years, there appeared more of a distinction between the two designs than appears today. Systems designed on the RISC philosophy included IBM's System/360 (1964) which was the first commercially available micro programmed computer architecture latter to become known as CISC architecture. The success of System/360 resulted in CISC architectures dominating computer, and later microprocessor, design for two decades. Other CISC computers included VAX (mid-1970s); PDP-11 (1970s); and the Motorola 68000 series (1970s). CISC was also an influence on the Windows 3.1 (1992) and Windows 95 were designed with CISC processors in mind. If pure CISC design is not commonly used in today’s computing systems, it may have to do with the following:

- Increasing instruction set & chip hardware complexity leads backward-compatibility issues
- Variable length instructions slow down the overall performance of the machine
- Many specialized instructions are15 not used frequently
- CISC instructions typically also set the condition codes as a side effect

Whatever the disadvantages in CISC usage, it took at least 10 years for RISC to take commercial hold as Intel was one roadblock to its widespread implementation as Intel (a CISC user) had vast resources to continue implementing the CISC model.

Although RISC is often thought of as a more recent development, in fact the first system which could now be considered as RISC-based was the CDC 6600 supercomputer, designed in 1964. But the first major RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. More modern systems were Apple's Power Macintosh line (1994) Windows NT (1993) was RISC compatible.

Nowadays, CISC and RISC conflict has ended with some factions claiming that RISC has won, and have more or less converge. Examples of convergences include:

- With an increase in processor speeds, CISC chips are now able to execute more than one instruction within a single clock which enables RISC-like pipelining.
- We can fit many more transistors on a single chip thus providing more space to execute CISC-like commands.

Although today, it is proposed that Intel x86 is the only chip which retains pure CISC architecture. It is however at least as fast the fastest true RISC single-chip solutions available. How the two are compared depends on whether a qualitative comparison or a quantitative comparison is made.

Well known RISC families include DEC Alpha, AMD 29k, ARC, ARM, Atmel AVR, MIPS, PA-RISC, Power (including PowerPC), SuperH, and SPARC.

## 1.7 Atmel AVR

The AVR is a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time.


The AVR is a modified Harvard architecture machine with program and data stored in separate physical memory systems that appear in different address spaces, but having the ability to read data items from program memory using special instructions.

### 1.7.1 Device architecture

Flash, EEPROM, and SRAM are all integrated onto a single chip, removing the need for external memory in most applications. Some devices have a parallel external bus option to allow adding additional data memory or memory-mapped devices. Almost all devices have serial interfaces, which can be used to connect larger serial EEPROMs or flash chips.

### 1.7.2 Program memory

Program instructions are stored in non-volatile flash memory. Although they are 8-bit MCUs, each instruction takes one or two 16-bit words.

The size of the program memory is usually indicated in the naming of the device itself (e.g., the ATmega64x line has 64 kB of flash while the ATmega32x line has 32 kB).

### 1.7.3 Internal data memory

The data address space consists of the register file, I/O registers, and SRAM.

### 1.7.4 Internal registers

The AVRs have 32 single-byte registers and are classified as 8-bit RISC devices.

In most variants of the AVR architecture, the working registers are mapped in as the first 32 memory addresses (000016-001F16) followed by the 64 I/O registers (002016-005F16).

Actual SRAM starts after these register sections (address 006016). (Note that the I/O register space may be larger on some more extensive devices, in which case the memory mapped I/O registers will occupy a portion of the SRAM address space.)

Even though there are separate addressing schemes and optimized opcodes for register file and I/O register access, all can still be addressed and manipulated as if they were in SRAM.

#### *1.7.4.1 EEPROM*

Almost all AVR microcontrollers have internal EEPROM for semi-permanent data storage. Like flash memory, EEPROM can maintain its contents when electrical power is removed.

In most variants of the AVR architecture, this internal EEPROM memory is not mapped into the MCU's addressable memory space. It can only be accessed the same way an external peripheral device is, using special pointer registers and read/write instructions which makes EEPROM access much slower than other internal RAM.

### 1.7.5 Program execution

Atmel's AVRs have a two stage, single level pipeline design. This means the next machine instruction is fetched as the current one is executing. Most instructions take just one or two clock cycles, making AVRs relatively fast among the eight-bit microcontrollers.

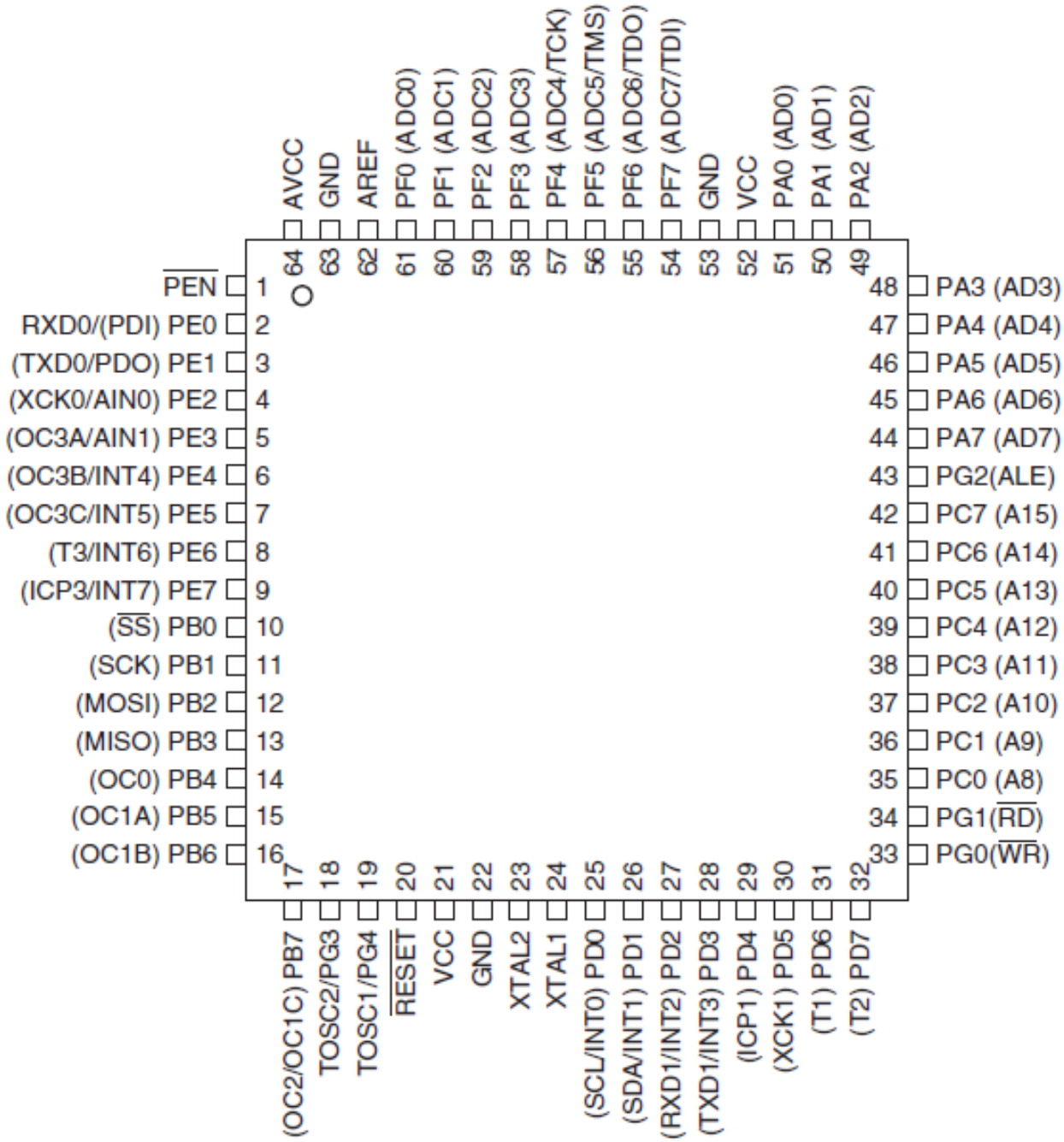
The AVR family of processors were designed with the efficient execution of compiled C code in mind and has several built-in pointers for the task.

# Chapter 2 - Introduction to Atmega128A Microcontroller

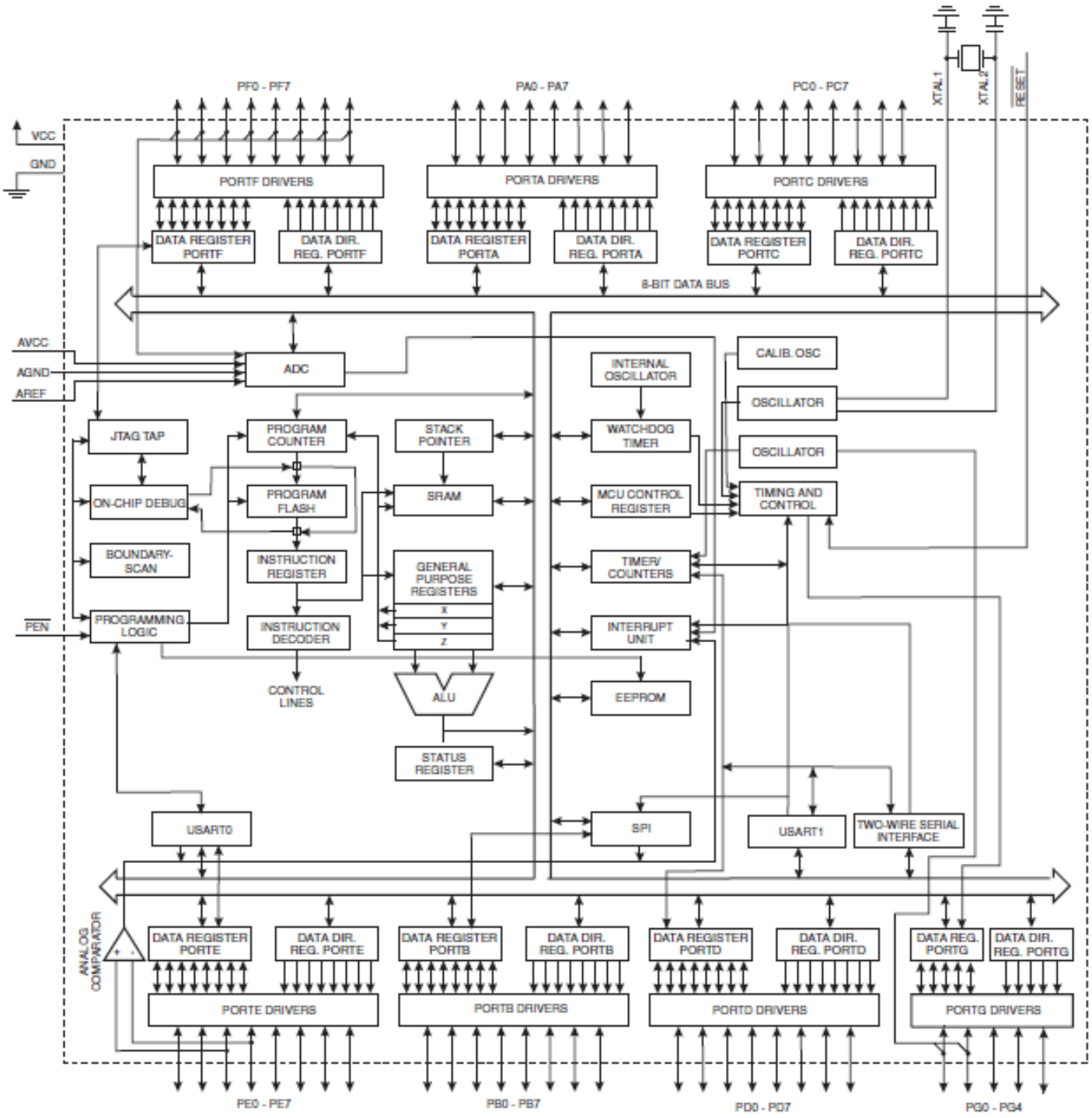
## 2.1 Features

• High-performance, Low-power AVR® 8-bit Microcontroller. Advanced RISC Architecture.
• Nonvolatile Program and Data Memories
– 128K Bytes of In-System Reprogrammable Flash. Endurance: 10,000 Write/Erase Cycles
– Optional Boot Code Section with Independent Lock Bits
In-System Programming by On-chip Boot Program
True Read-While-Write Operation
– 4K Bytes EEPROM
Endurance: 100,000 Write/Erase Cycles
– 4K Bytes Internal SRAM
– Up to 64K Bytes Optional External Memory Space
– Programming Lock for Software Security
– SPI Interface for In-System Programming
• JTAG (IEEE std. 1149.1 Compliant) Interface
– Boundary-scan Capabilities According to the JTAG Standard
– Extensive On-chip Debug Support
– Programming of Flash, EEPROM, Fuses and Lock Bits through the JTAG Interface
• Peripheral Features
– 53 Programmable I/O Lines
– Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
– Two Expanded 16-bit Timer/Counters with Separate Prescaler, Compare Mode and Capture Mode
– Real Time Counter with Separate Oscillator
– Two 8-bit PWM Channels
– 6 PWM Channels with Programmable Resolution from 2 to 16 Bits
– Output Compare Modulator
– 8-channel, 10-bit ADC
8 Single-ended Channels
7 Differential Channels
2 Differential Channels with Programmable Gain at 1x, 10x, or 200x
– Byte-oriented Two-wire Serial Interface
– Dual Programmable Serial USARTs
– Master/Slave SPI Serial Interface
– Programmable Watchdog Timer with On-chip Oscillator
– On-chip Analog Comparator
• Special Microcontroller Features
– Power-on Reset and Programmable Brown-out Detection
– Internal Calibrated RC Oscillator
– External and Internal Interrupt Sources
– Software Selectable Clock Frequency
– Global Pull-up Disable

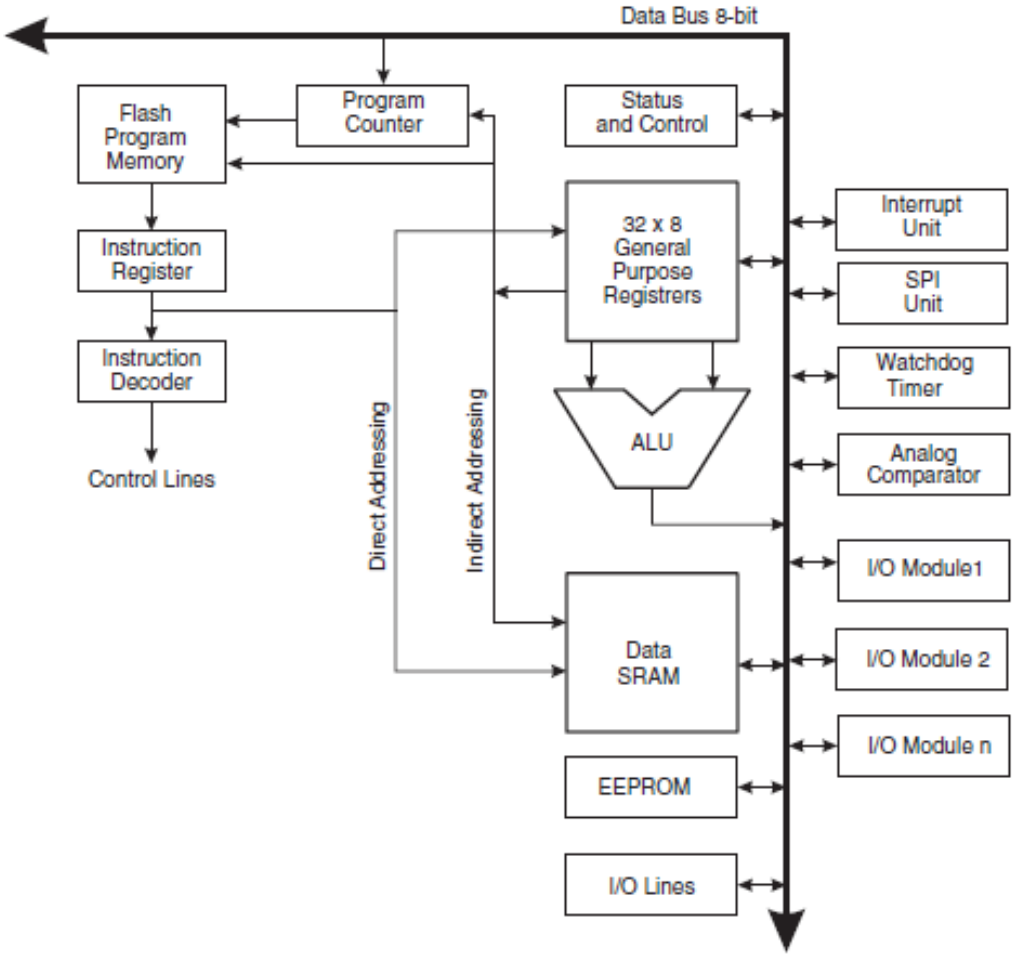
## 2.2 Pin Configuration



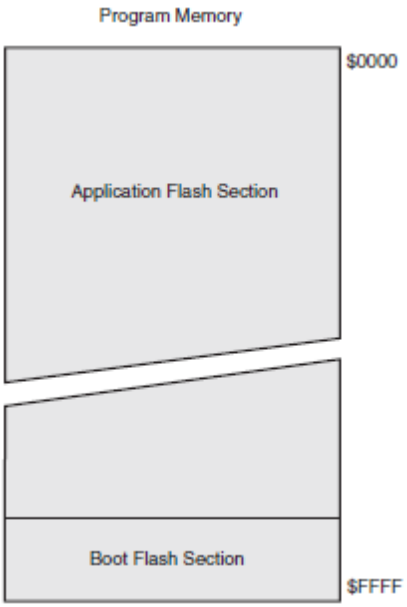
## 2.3 Block Diagram



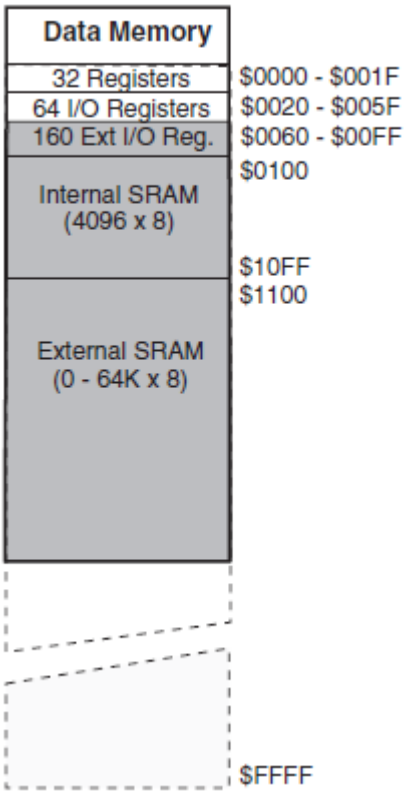
## 2.4 Block Diagram of the AVR Architecture



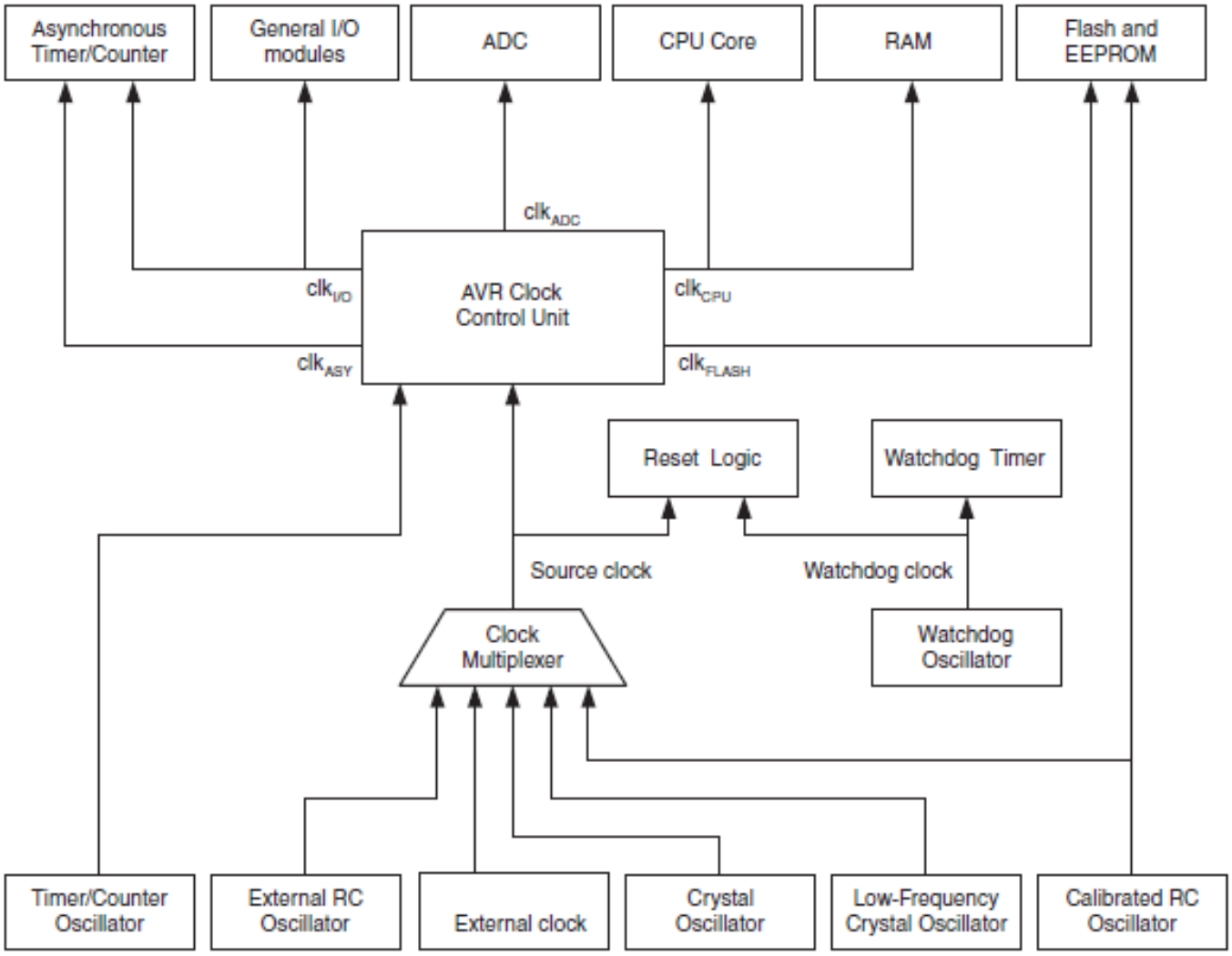
## 2.5 Program Memory Map




## 2.6 Data Memory Map



## 2.7 Clock System




### 2.7.1 Clock Options

- External Crystal/Ceramic Resonator
- External Low-frequency Crystal
- External RC Oscillator
- Calibrated Internal RC Oscillator
- External Clock

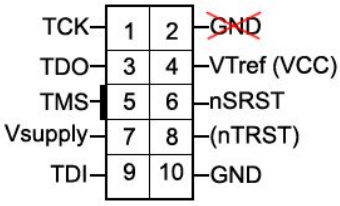
Clock options can be configured from Fuse Bits.

### 2.7.2 Default Clock Source

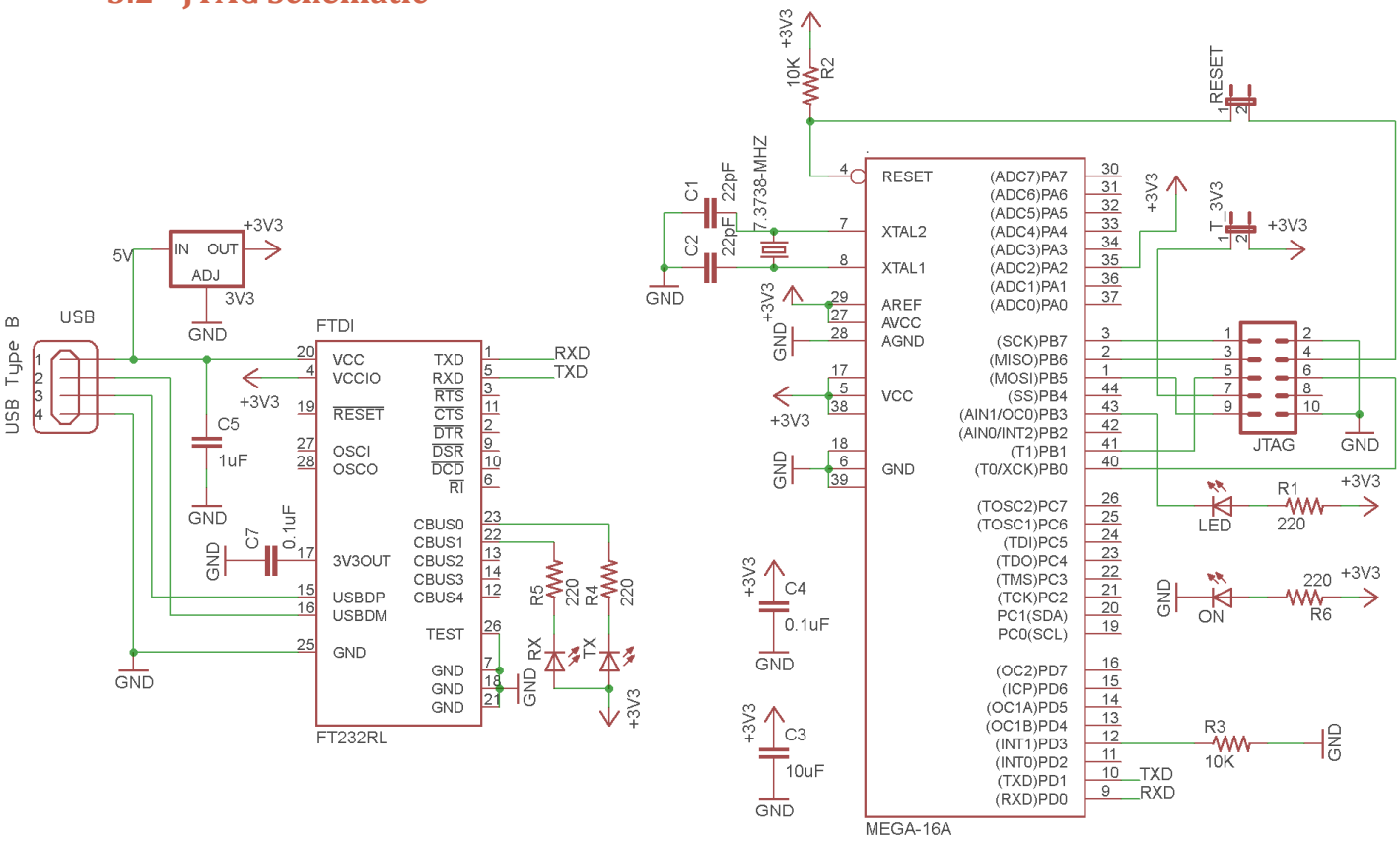
The default clock source setting is the 1 MHz Internal RC Oscillator with longest startup time. This default setting ensures that all users can make their desired clock source setting using an In-System or Parallel Programmer.

# Chapter 3 - JTAG Description and Installing FTDI Driver

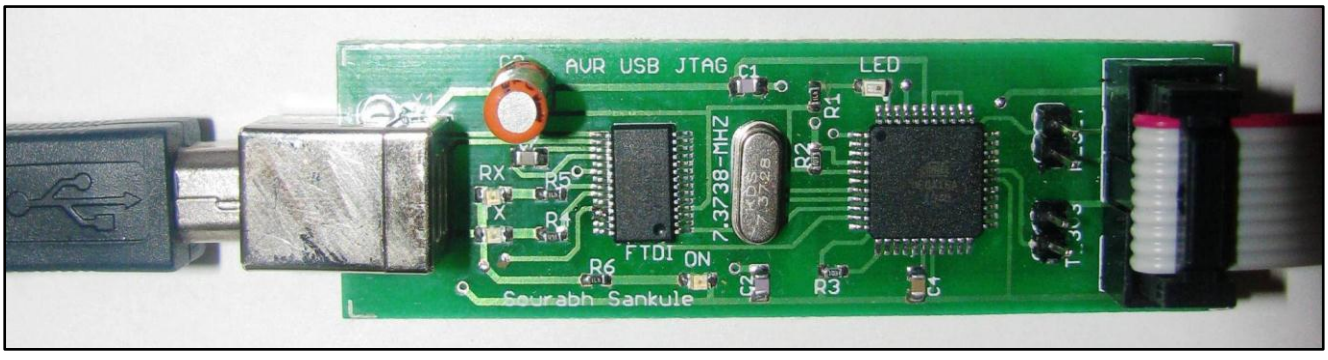
## 3.1 JTAG Pin out




## 3.2 JTAG Schematic




## 3.3 JTAG Image



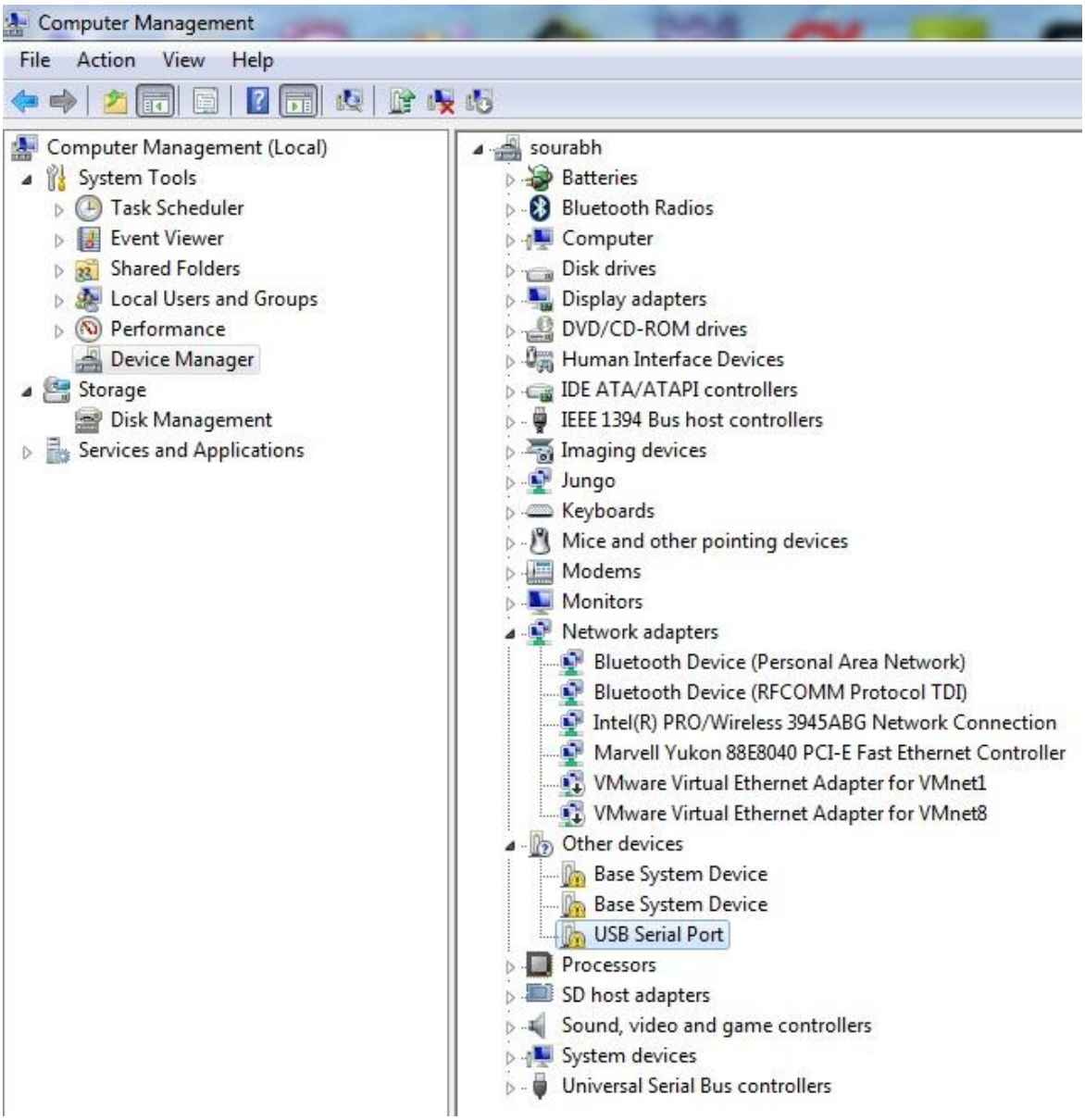
## 3.4 How to install FTDI drivers (USB-UART chip)

1. Download FTDI drivers from,

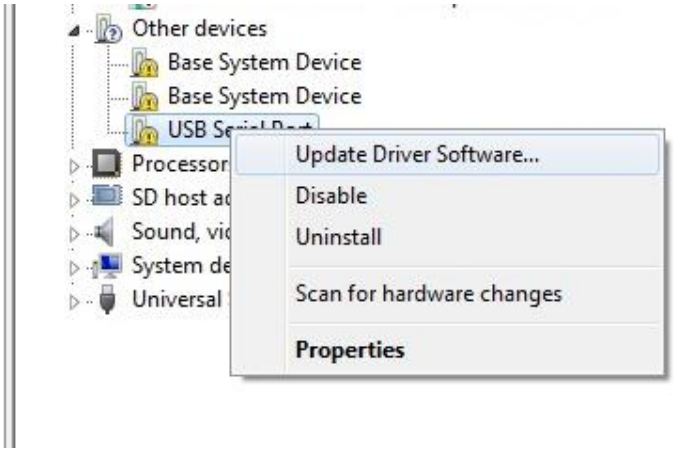
http://www.ftdichip.com/Drivers/VCP.htm

Save it on the drive and unzip the folder.

2. Plug the USB cable (JTAG / USB on the board). Open **Device Manager** where you can see following,

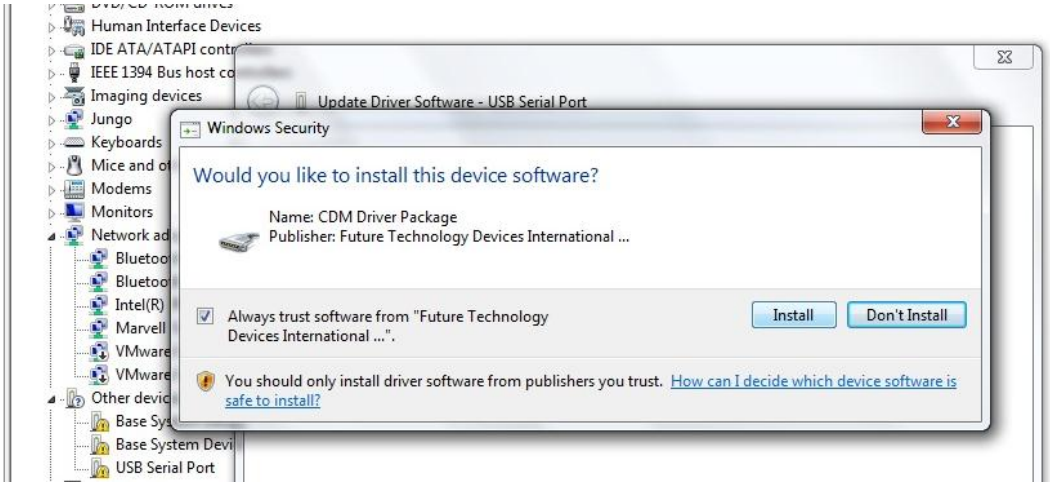


3. Right click on **USB Serial Port** and click on **Update Driver Software**



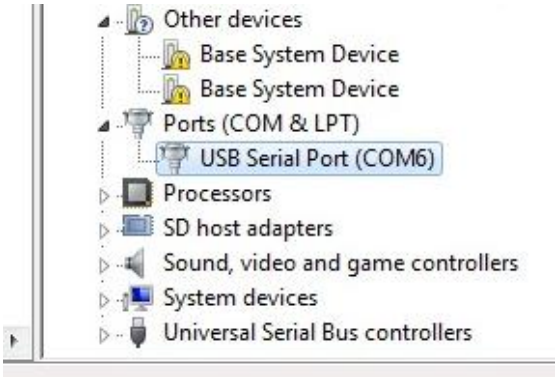

4. Browse to the unzipped driver folder.


5. It will ask for confirmation, click on **Install**

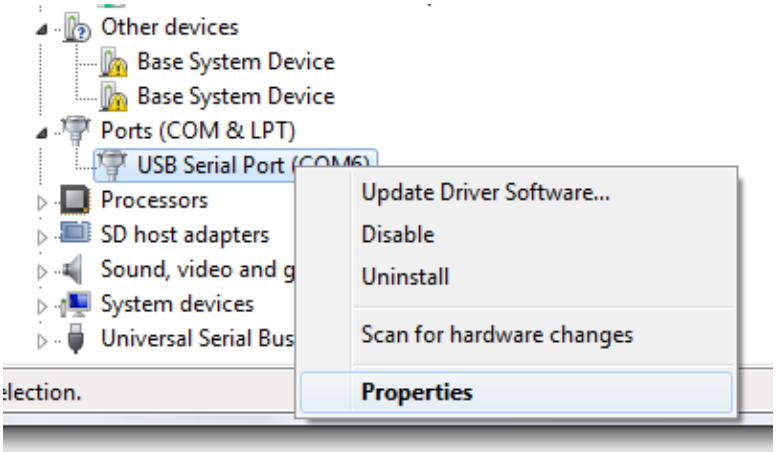


6. Then you can see the successful driver install message.

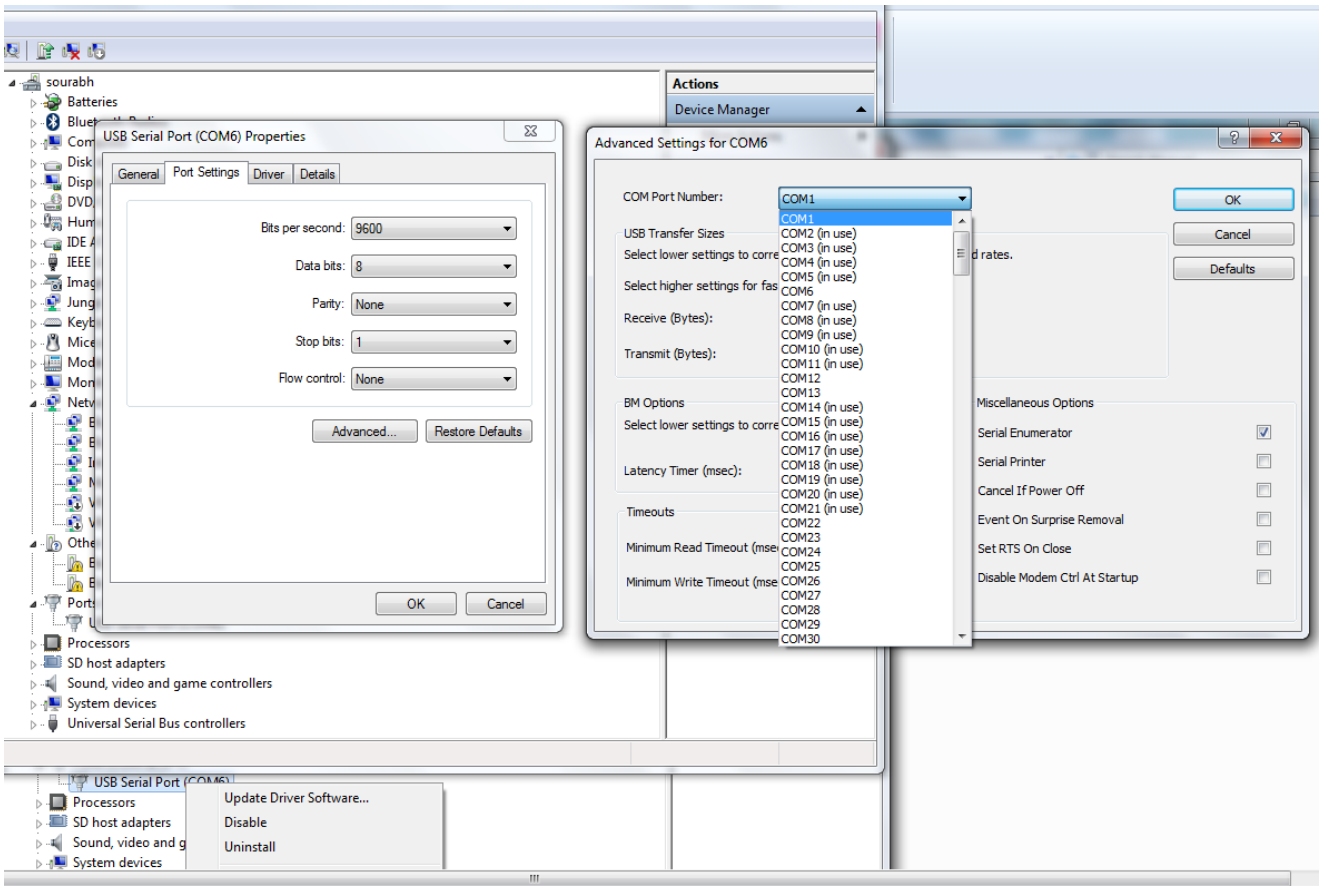

7. You can check the assigned COM PORT number,



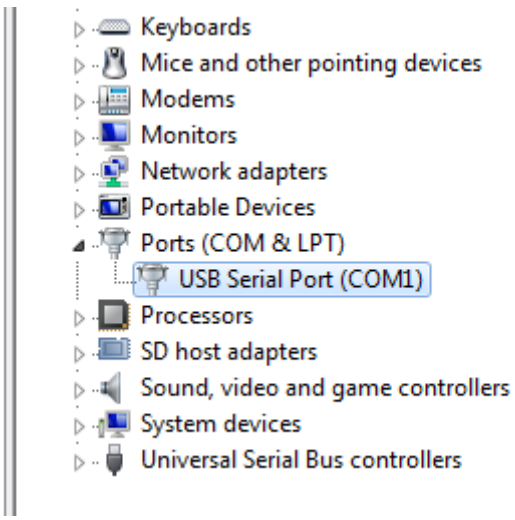

8. If you wish to change COM PORT number (AVR Studio, JTAG ICE detects COM Port from 1 to 4), then right click and click on **Properties**



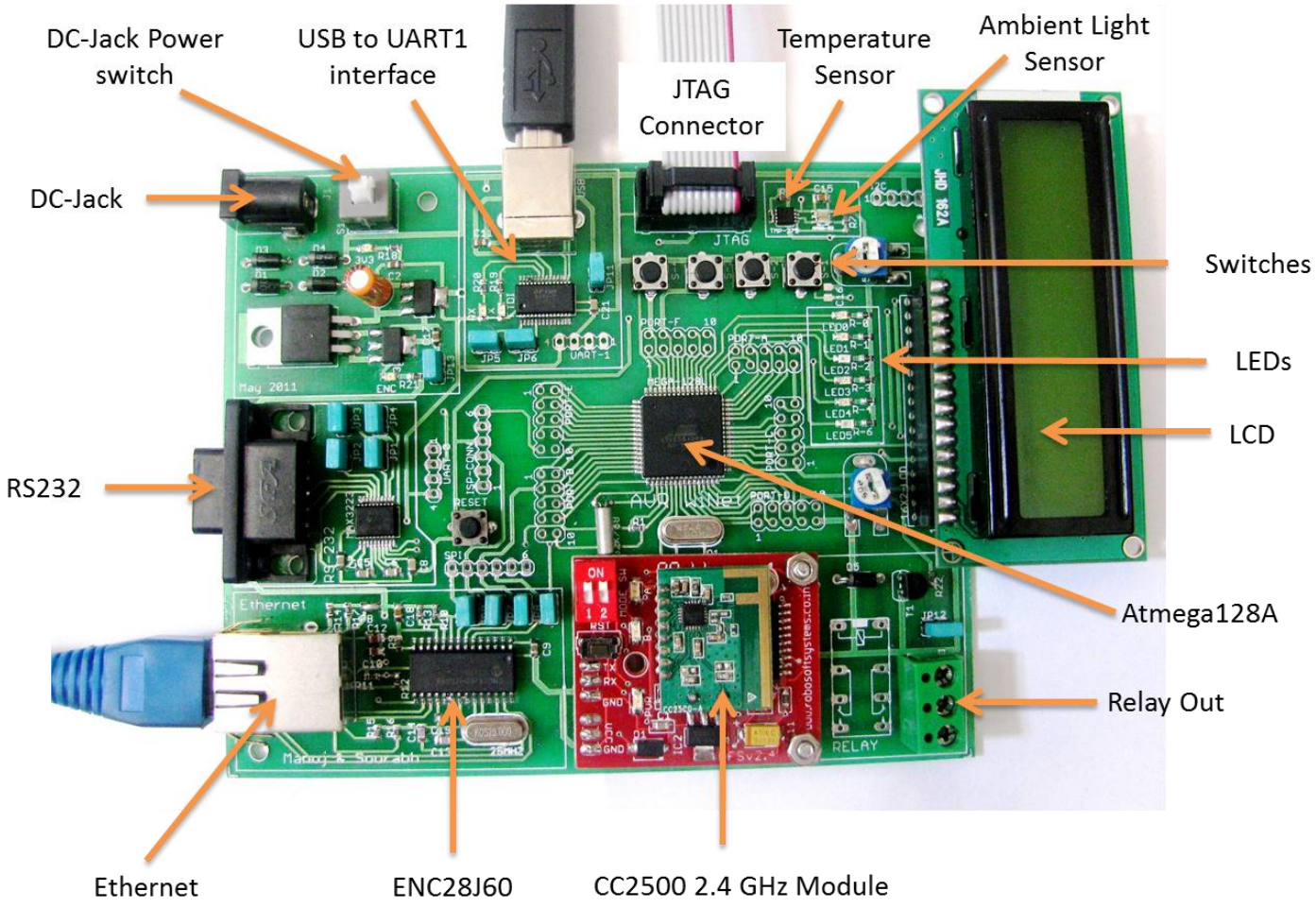
9. Now open **Port Settings** Tab and click on **Advanced** and choose desired COM Port and click on **OK**




10. Check the new assigned COM Port number.

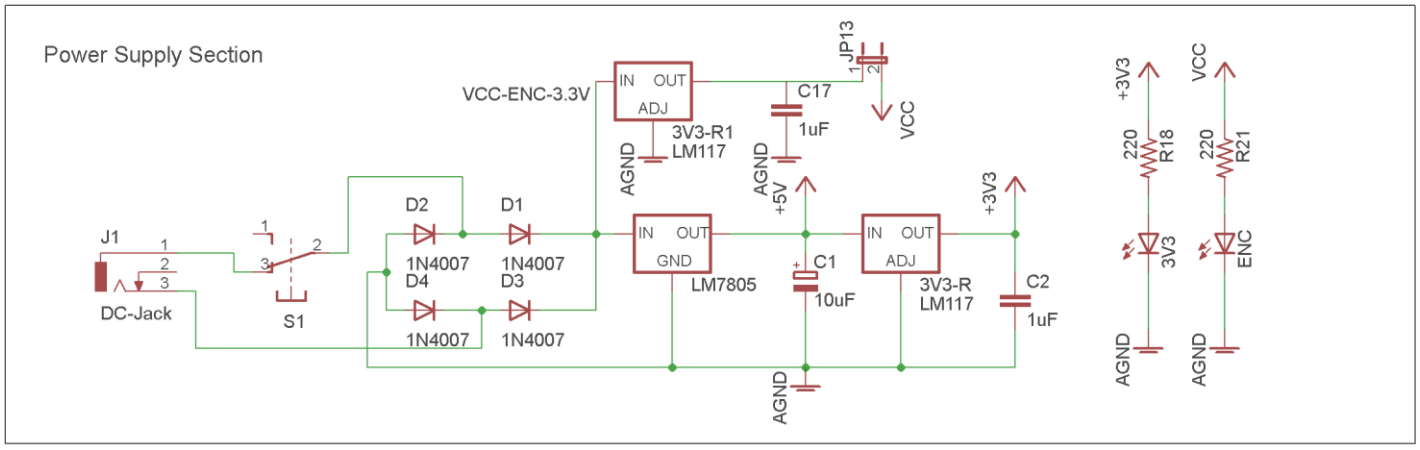


# Chapter 4 - WiNet (Wireless-Networking) Board Description

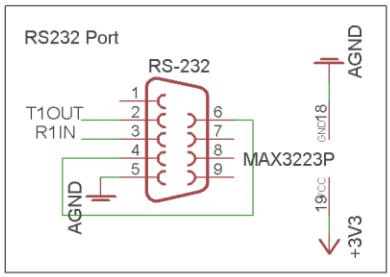


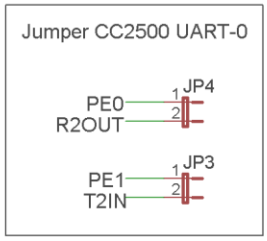

## 4.1 Peripherals

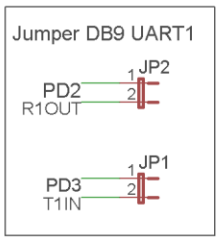
- 6 LEDs connected in Sink Mode.
- 4 Switches connected to GND, internal pull up required from microcontroller.
- TMP-275 Temperature Sensor, with I2C interface.
- APDS-9300 Ambient Light Sensor, with I2C interface.
- 16x2 / 40x4 LCD.
- Relay for controlling external load.
- USB to UART interface using FTDI-FT232 chip.
- Ethernet interface using ENC28J60 Mac-Phy chip.
- RS-232 Port using MAX3232.
- CC2500 2.4 GHz Module for Wireless Communication.
- GPIO output from all Ports.
- UART0 UART1 I2C and SPI Output.

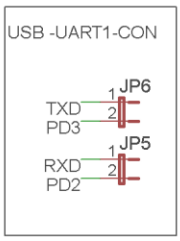


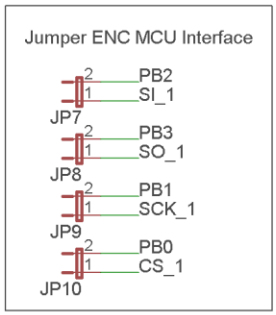

# WiNet Board Schematic

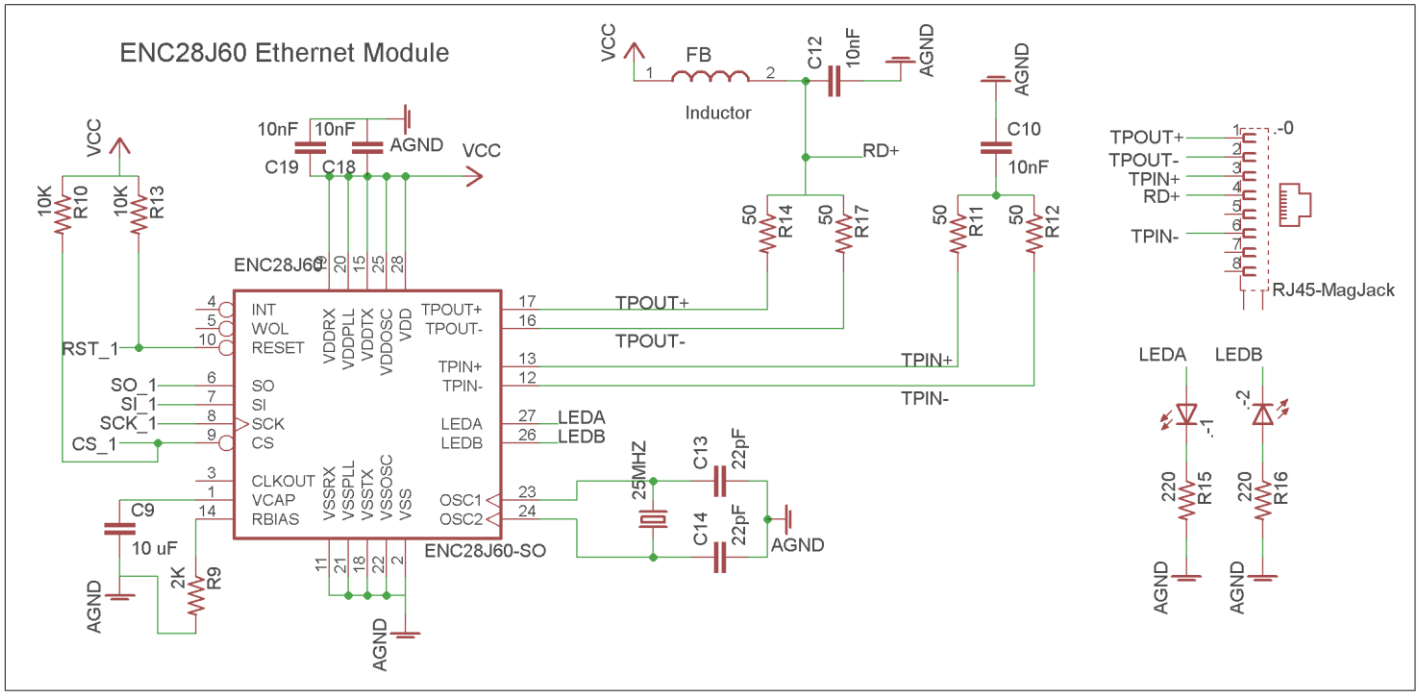


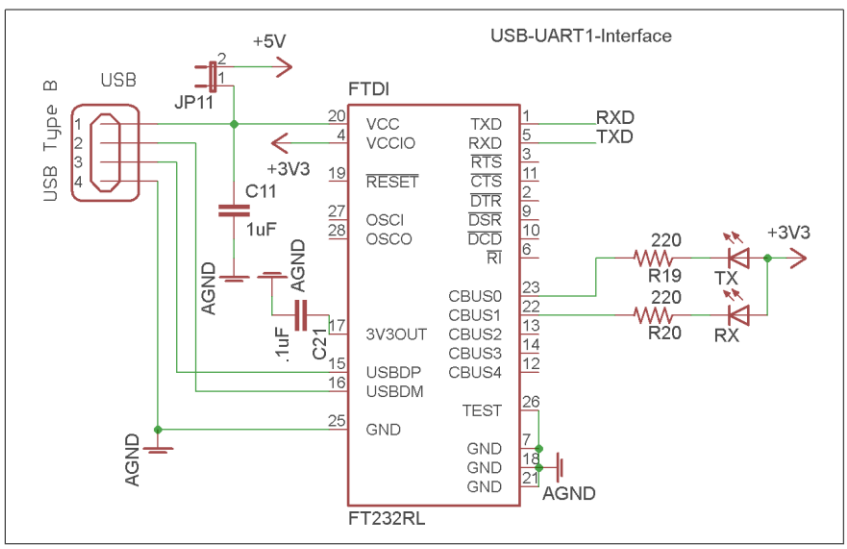



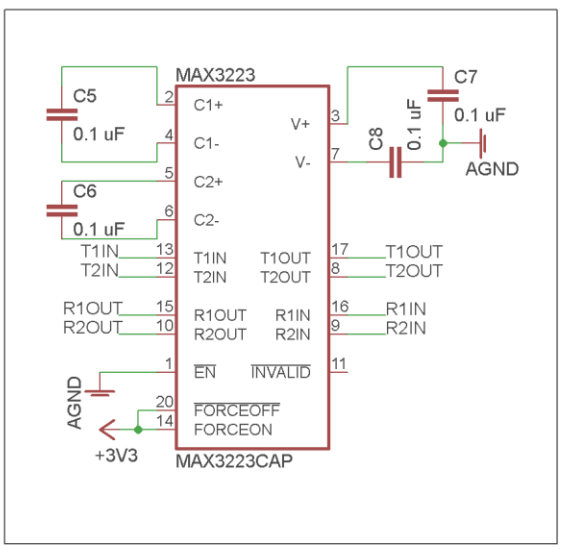



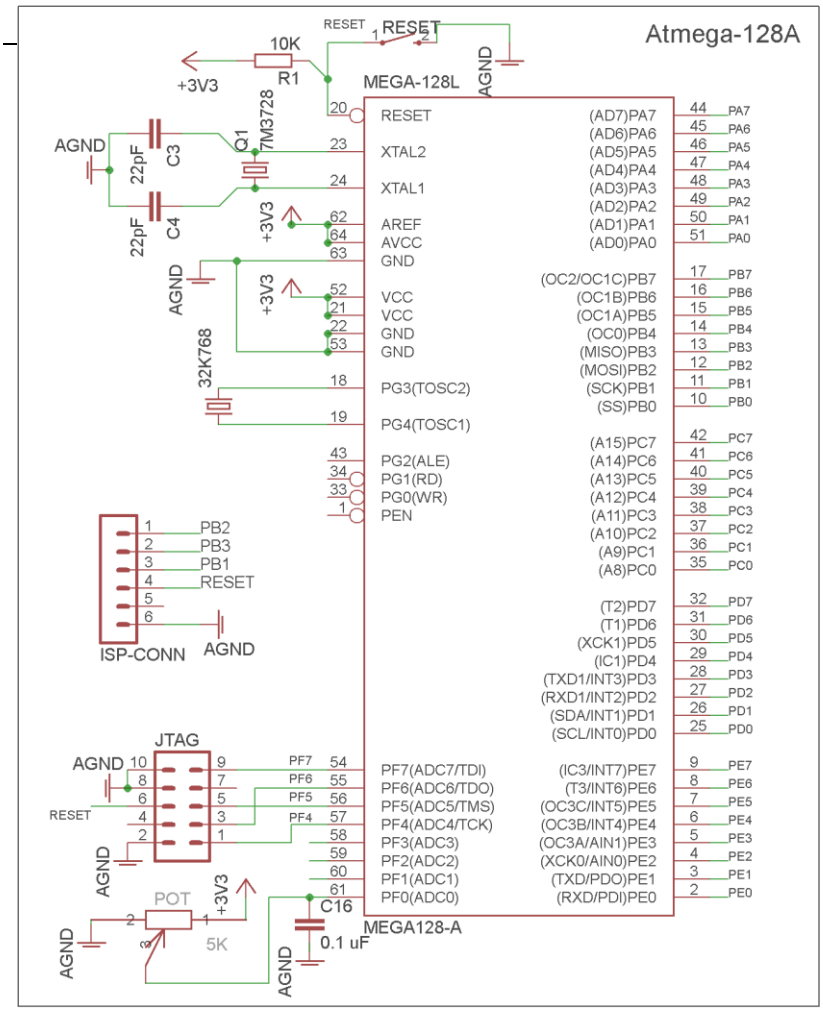



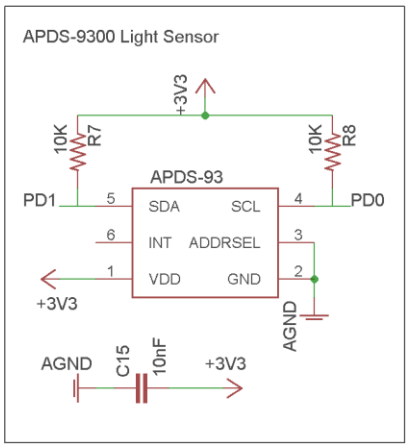



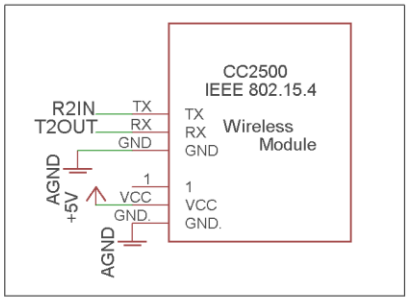



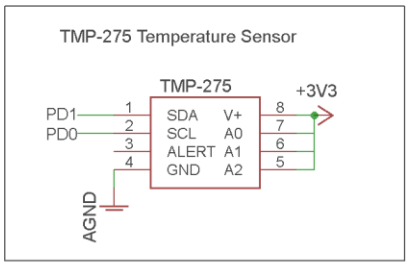



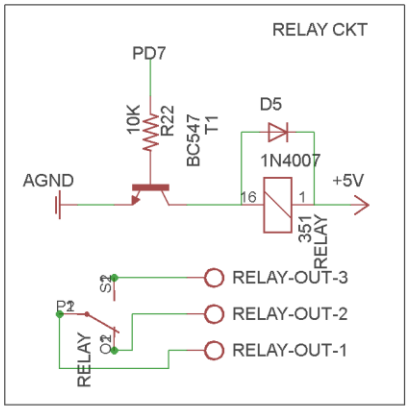



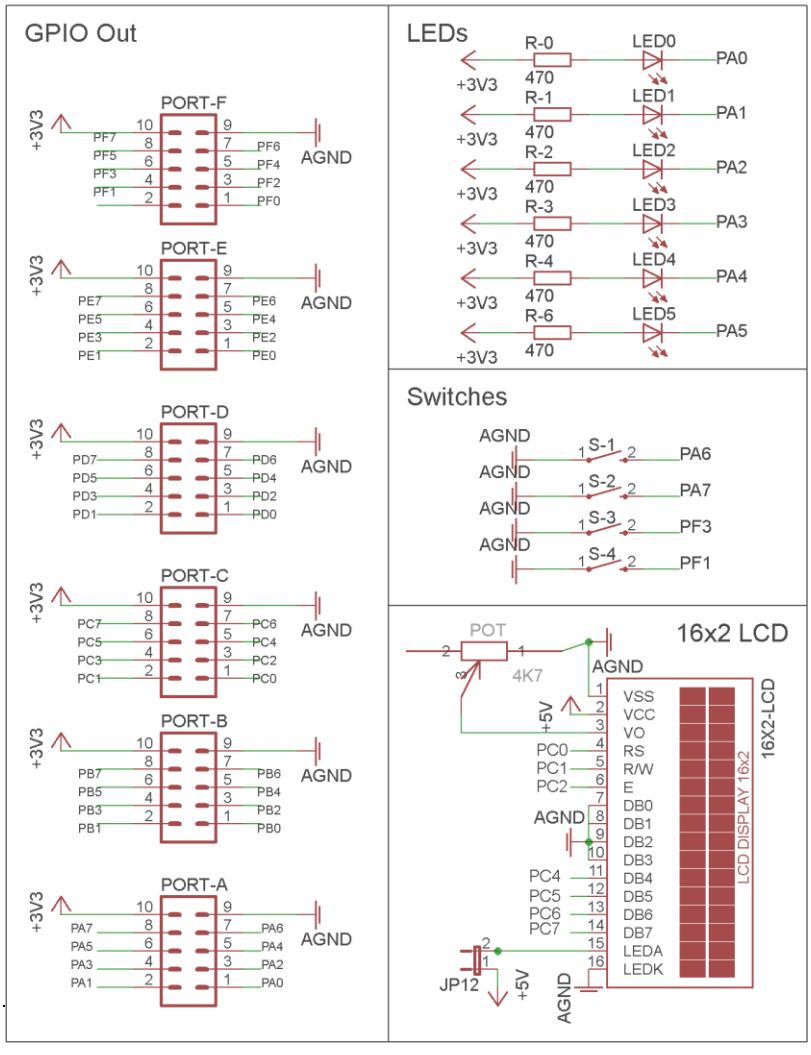

Atmega-128A
MEGA-128L
MEGA128-A
RESET
10K
R1
+3V3
AGND
7M3728
Q1
22pF
C3
C4
32K768
ISP-CONN
JTAG
POT
5K
C16
0.1 uF
PB2
PB3
PB1
RESET
PF7(ADC7/TDI)
PF6(ADC6/TDO)
PF5(ADC5/TMS)
PF4(ADC4/TCK)
PF3(ADC3)
PF2(ADC2)
PF1(ADC1)
PF0(ADC0)
(AD7)PA7
(AD6)PA6
(AD5)PA5
(AD4)PA4
(AD3)PA3
(AD2)PA2
(AD1)PA1
(AD0)PA0
(OC2/OC1C)PB7
(OC1B)PB6
(OC1A)PB5
(OC0)PB4
(MISO)PB3
(MOSI)PB2
(SCK)PB1
(SS)PB0
(A15)PC7
(A14)PC6
(A13)PC5
(A12)PC4
(A11)PC3
(A10)PC2
(A9)PC1
(A8)PC0
(T2)PD7
(T1)PD6
(XCK1)PD5
(IC1)PD4
(TXD1/INT3)PD3
(RXD1/INT2)PD2
(SDA/INT1)PD1
(SCL/INT0)PD0
(IC3/INT7)PE7
(T3/INT6)PE6
(OC3C/INT5)PE5
(OC3B/INT4)PE4
(OC3A/AIN1)PE3
(XCK0/AIN0)PE2
(TXD/PDO)PE1
(RXD/PDI)PE0

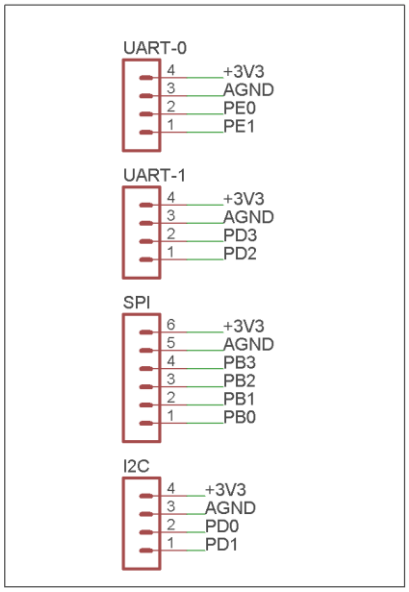

APDS-9300 Light Sensor
APDS-93
SDA
SCL
INT
ADDRSEL
VDD
GND
PD1
PD0
10K
R7
10K
R8
C15
10nF


CC2500
IEEE 802.15.4
Wireless
Module
R2IN
T2OUT
TX
RX
GND
VCC


TMP-275 Temperature Sensor
TMP-275
SDA
SCL
ALERT
GND
V+
A0
A1
A2
PD1
PD0
+3V3


RELAY CKT
PD7
10K
R22
BC547
T1
D5
1N4007
AGND
+5V
RELAY
RELAY-OUT-3
RELAY-OUT-2
RELAY-OUT-1


GPIO Out
PORT-F
PORT-E
PORT-D
PORT-C
PORT-B
PORT-A
LEDs
R-0
R-1
R-2
R-3
R-4
R-6
LED0
LED1
LED2
LED3
LED4
LED5
470
Switches
S-1
S-2
S-3
S-4
16x2 LCD
POT
4K7
VSS
VCC
VO
RS
R/W
E
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
LEDA
LEDK
16X2-LCD
LCD DISPLAY 16x2
JP12

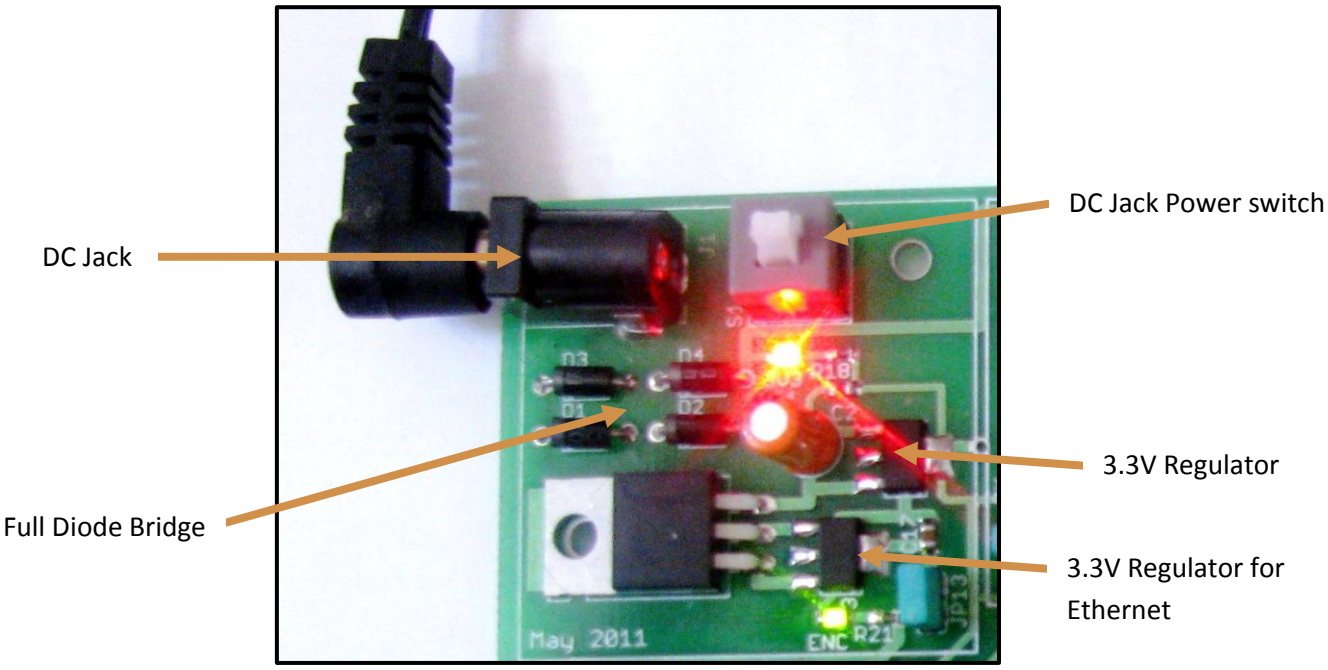

UART-0
UART-1
SPI
I2C
+3V3
AGND
PE0
PE1
PD3
PD2
PB3
PB2
PB1
PB0
PD0
PD1

## 4.2 Powering the Board

### 4.2.1 Using DC Adapter

Board contains DC regulator of 3.3V and 5V which can be fed from any DC source < 12V. Full Diode Bridge is provided at the input, securing wrong polarity errors.

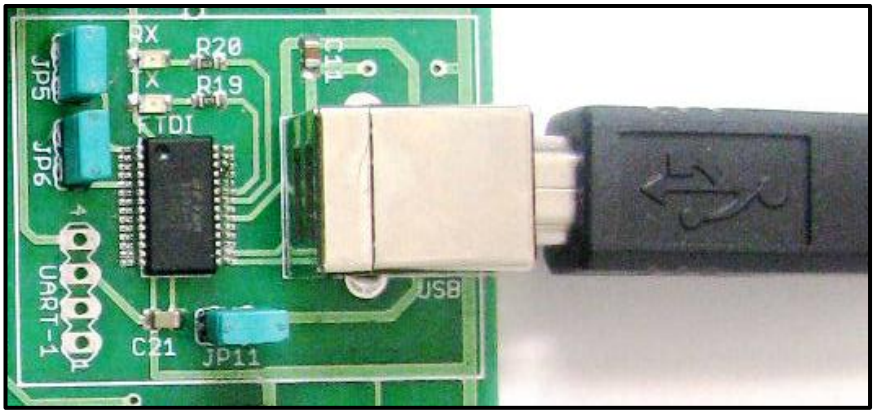
***=> No polarity considerations while giving DC jack input.***




Connect the 9V DC adapter provided, and press the DC-Jack Switch, **RED power LED** should turn ON, indicating successful powering of the board.

### 4.2.2 Using USB

Connect the USB Cable and set the Jumper JP11.



### 4.2.3 Powering Ethernet Section

Set the Jumper JP13 to power the Ethernet Section.

**Note:** Ethernet section cannot be powered from USB.

## 4.3 Microcontroller

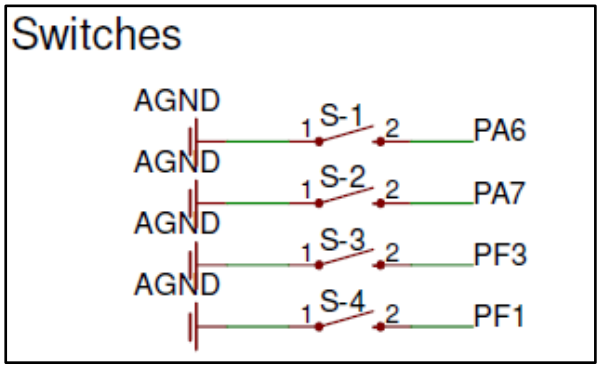
WiNet Board has **Atmega128A** as the processing unit. AtMega128A is a 3.3V device, it has 128KB flash, 4KB RAM, I2C, SPI, JTAG, ISP and UART interfaces. Complete details can found in the datasheet.

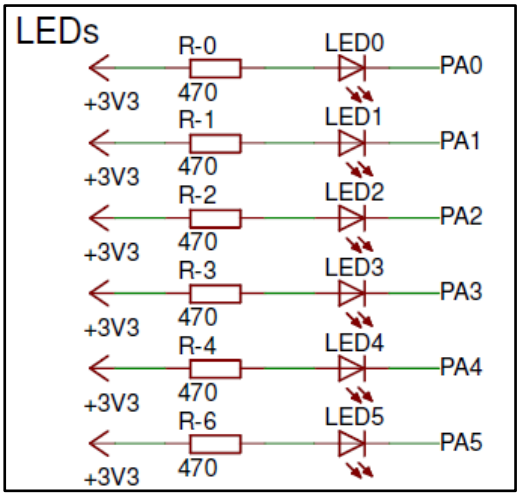
## 4.4 LEDs

6 LEDs are provided on WiNet for general purpose debugging. They are connected from PORTA-0 to PORTA-5

## 4.5 Switches

4 Switches are provided for general inputs. On pressing it connects to GND. User has to pull the line up from the microcontroller itself for no press condition.

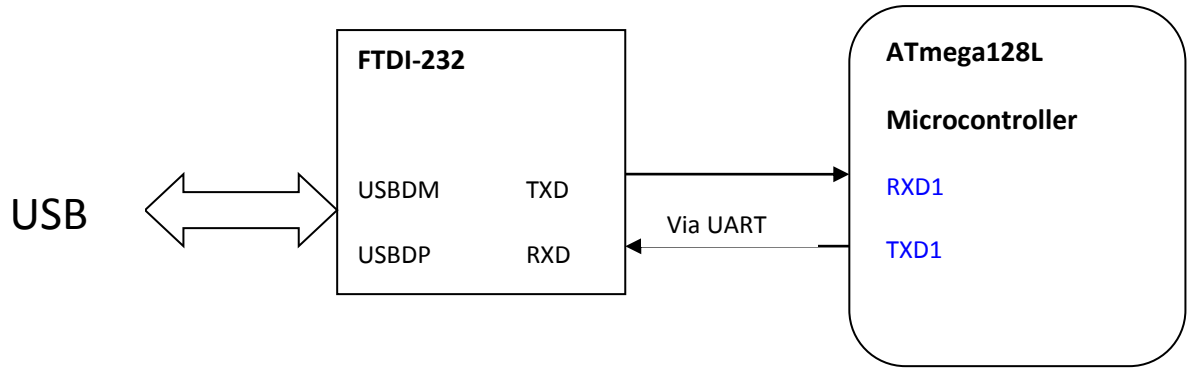


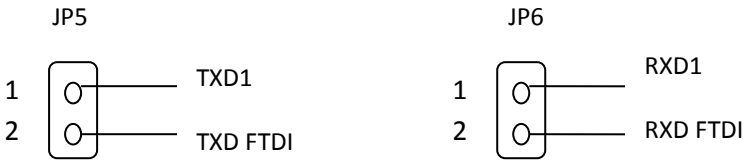



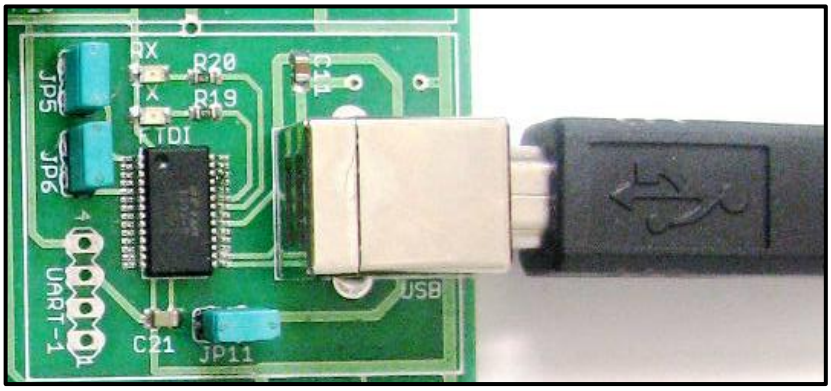

## 4.6 USB to UART interface

WiNet provides a USB interface for serial communication test applications. This USB interface is actually an UART- USB interface based on FTDI232 chip for PC to WiNet communication. The interface is as shown in figure below.

For installing drivers, please read the chapter ***'How to install FTDI Drivers'***



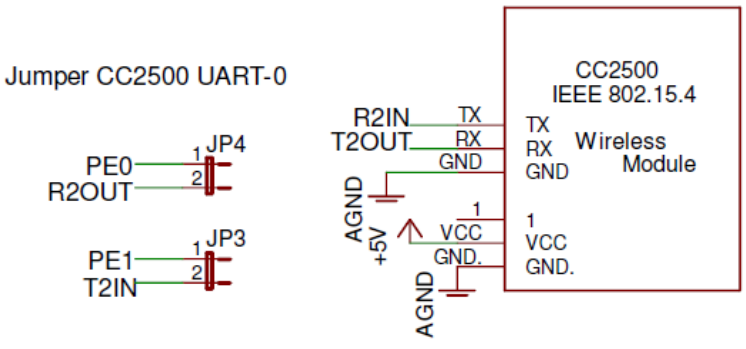



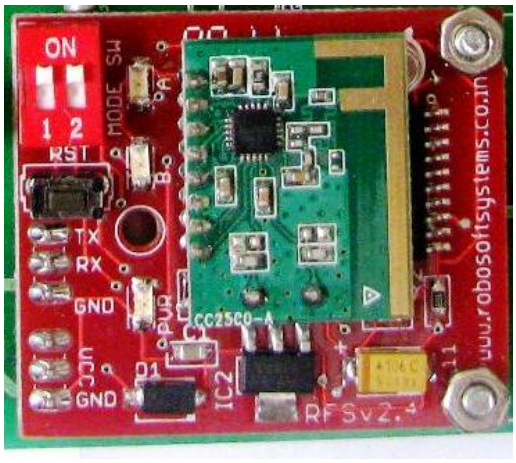




## 4.7 CC2500 Module

CC2500 is a 2.4 GHz compliant radio. It has a data rate of 250Kbps and range of 30m (indoor) and 100m (outdoor). It is interfaced with the microcontroller via **UART0**.

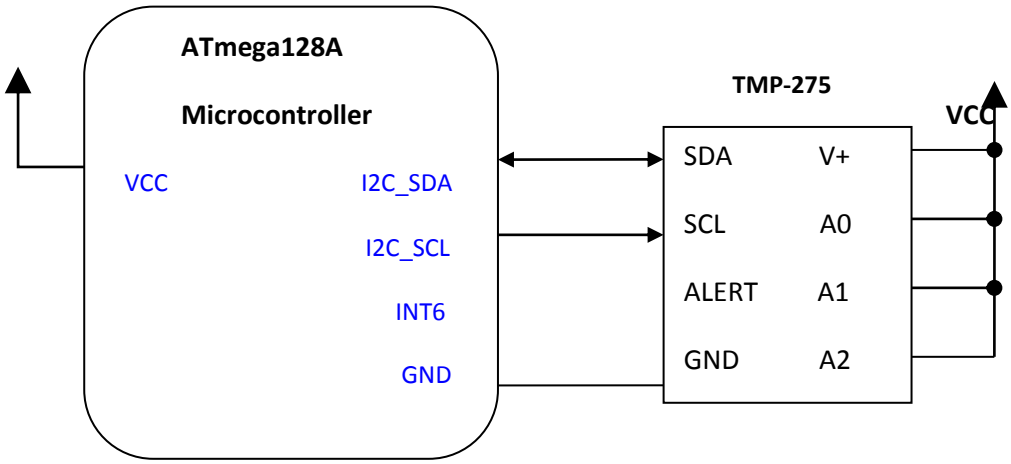
Figure below shows the interface in WiNet.



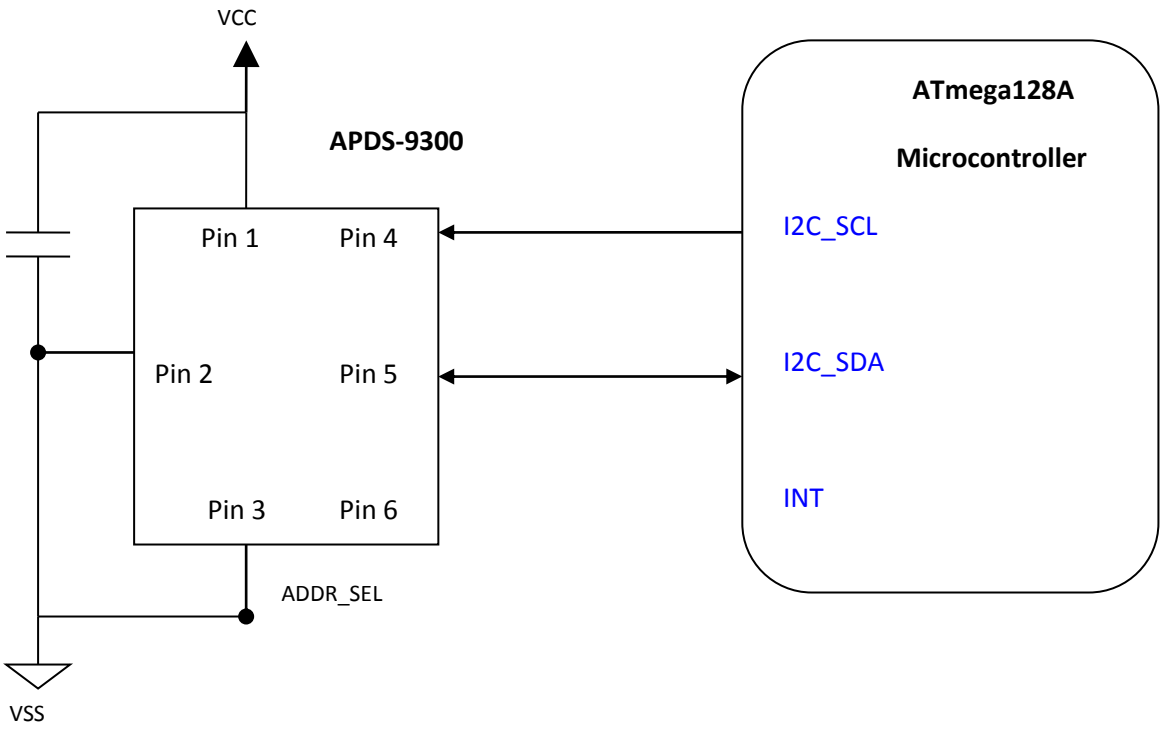
## 4.8 Temperature Sensor

TMP275 is a temperature sensor from TI. It has a resolution of 0.5 degree centigrade. It is a digital output sensor interfaced with microcontroller via I2C. Details of TMP275 can be found in the datasheet.




## 4.9 Light Sensor

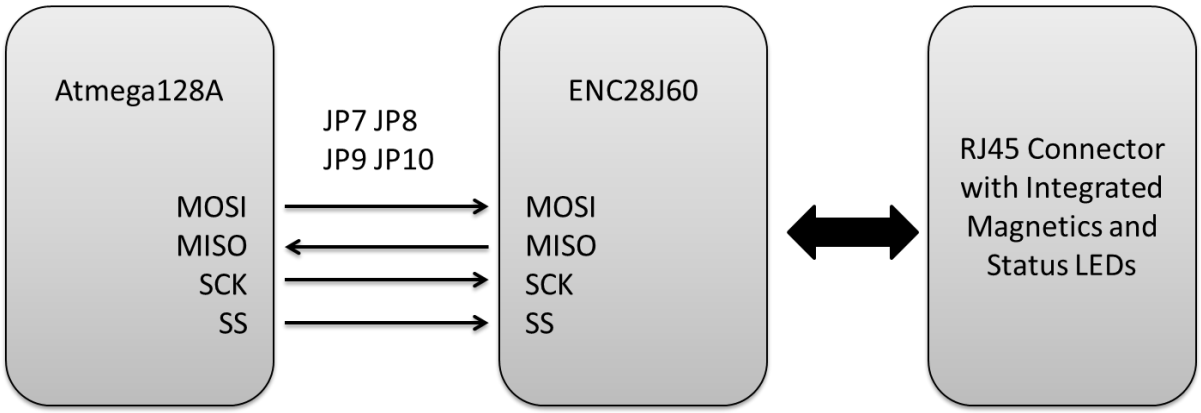
Light sensor is **APDS-9300** from **Avago** Technologies. The APDS-9300 is a low-voltage Digital Ambient Light Photo Sensor that converts light intensity to digital signal output capable of direct I2C interface. Each device consists of one broadband photodiode (visible plus infrared) and one infrared photodiode. Details can be found in the Datasheet.

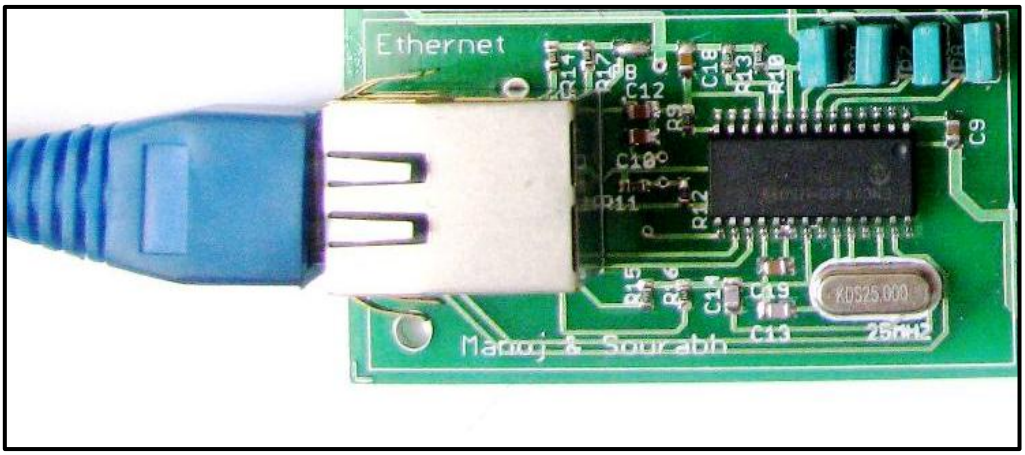


## 4.10 Ethernet Interface

WiNet board is connected to Ethernet via ENC28J60 chip from Microchip. It is an IEEE 802.3 compatible Ethernet controller with Integrated MAC and 10BASE-T PHY. TCP/IP stack reside on the Atmega128A microcontroller and drives the ENC chip via SPI interface.

Details about ENC chip can be found in the datasheet.



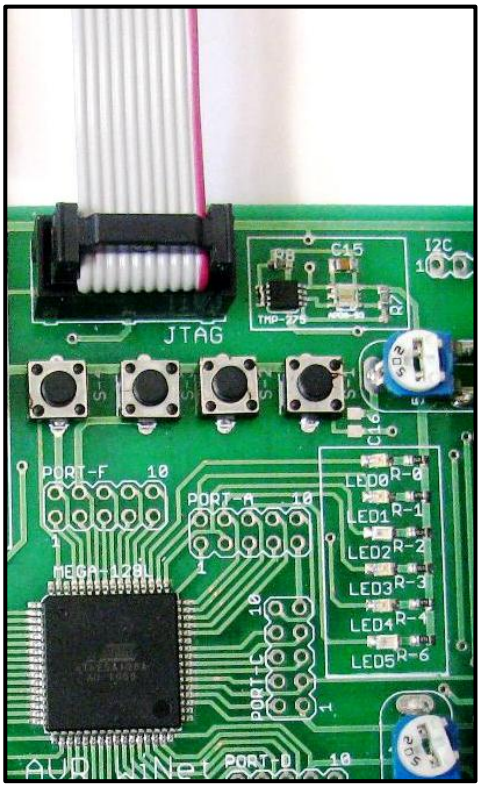

## 4.11 Programming WiNet Board

Atmel provides a free IDE for its micro-controllers “AVR Studio”.

Here are the steps to install the set up

- Download AVR Studio from http://www.atmel.com/dyn/products/tools_card_v2.asp?tool_id=2725
- Install AVR Studio.
- Install FTDI Drivers, Please read **‘How to Install FTDI Drivers’** chapter.
- Interface AVR JTAG ICE to WiNet board for programming or debugging

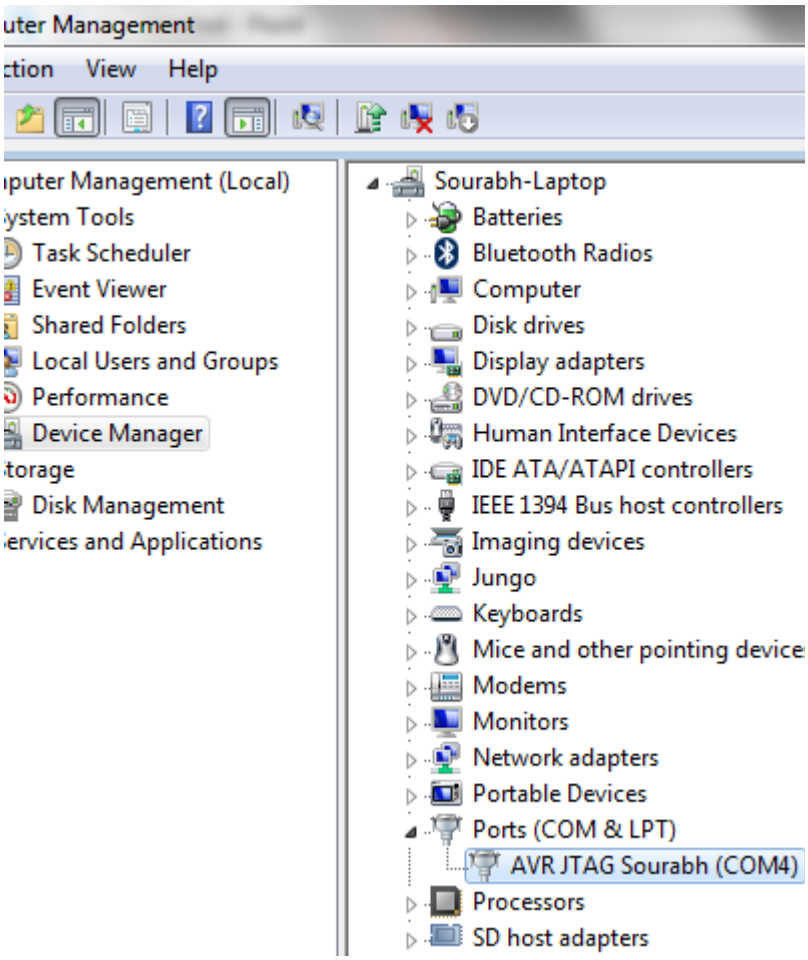
## 4.12 Connecting AVR JTAG ICE to WiNet Board




1. Connect the USB cable of the JTAG ICE to the computer. Install the FTDI driver if it was not previously installed. See the FTDI driver doc.

2. Connect the JTAG cable to the JTAG connector on the WiNet Board. See the image for correct orientation.

3. Check the COM PORT number is < 5. If not change it, see the FTDI driver chapter for the same.




After this please follow steps in **Section 5.8**

# Chapter 5 - Embedded Development Tools

## 5.1 Cross Compiler

A compiler which is used to compile code for platform other than the one you are using for compilation. As we are compiling code for AVR microcontrollers on x86/x64 based PCs, we are using cross-compilers. A cross-compiler is used when the host machine is not capable to compile the code itself. Generally we use cross-compilers for Microcontroller code compilation.

Here, we will be using GNU based GCC compiler for AVR. For windows, the compiler is available as WinAVR. WinAVR is windows version of AVR-GCC. The steps used for compilation with WinAVR are exactly similar with avr-gcc.

In this chapter, we will be providing an overview of all the utilities required in the building process. For further information on any command, please refer to **man** pages through linux. Also, you can read the documentation online at: http://linux.die.net/man/.

### 5.1.1 AVR Toolchain

The series of commands used for compilation is called Toolchain. The toolchain comprises of the compiler and other supporting utilities. A brief description of AVR-GCC toolchain commands is as:

1. **avr-gcc =>** The compiler for AVR microcontrollers, which gives object code as output.
2. **avr-as =>** AVR assembler, used for converting assembly code to object code.
3. **avr-ld =>** AVR-GCC linker, which will combine all object files into executable file, ELF.
4. **avr-objcopy =>** Used to convert executable into desired output format. We will use it to create hex file.
5. **avr-gdb =>** AVR microcontroller based debugger. A very powerful debugger but needs a lot of time to grasp.
6. **avrdude =>** Used to program the microcontroller with generated hex file.

A bunch of other utilities exist in addition to these. Refer to documentation for further guidance.

## 5.2 Installing Toolchain

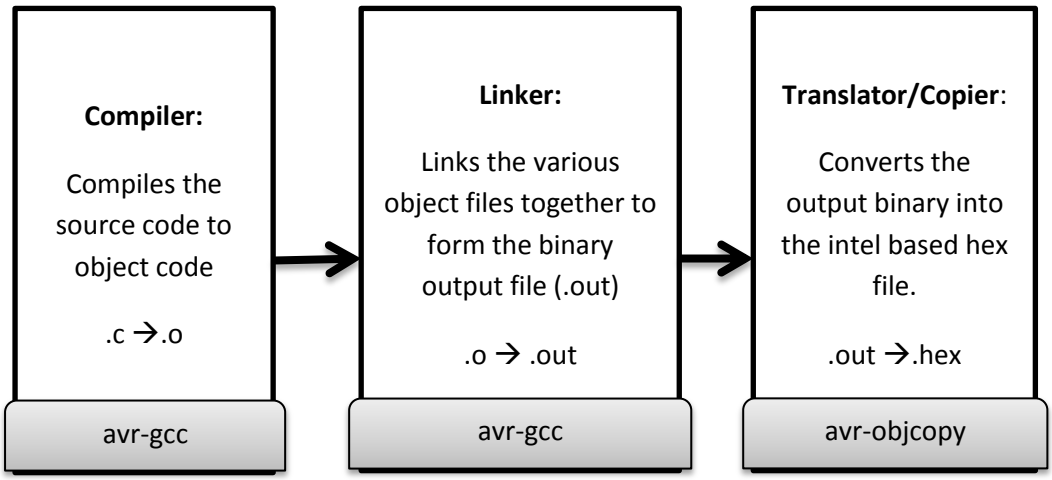
The installation procedure for avr-gcc is quite simple for both Linux as well as Windows.

| **For Linux(Ubuntu)** |
| --- |
| sudo apt-get install gcc-avr avrdude gdb-avr avr-libc binutils-avr avarice |

| **For Windows(Any Version)** |
| --- |
| 1. Download WinAVR from http://sourceforge.net/projects/winavr/files/WinAVR/20100110/.<br>2. Install it to any location. Prefer default location.<br>3. Installation takes just a few minutes and is as simple as Clicking **"Next"** a few times. |

## 5.3 Building the executable

As we know, we have a series of steps for building the executable code. The steps needed for converting the **source code** to the **intel based hex** file, are as:




### 5.3.1 Compiling the first program

Once we know the steps, we can easily build our program executable. For a source file;

**main.c**

```
#include<avr/io.h>
int main() {
        DDRA = 0xff;
        PORTA = 0x00;
        return 0;
}
```

**Step 1: Compilation (For AtMega128, as an example)**

```
avr-gcc main.c  -mmcu=atmega128  -c
```

Here, we get main.o as output. By default, avr-gcc command performs Linking operation automatically. We are using **–c** option to not to run linker but only perform compile action. This is just to demonstrate the Linking Process separately.

**Step 2: Linking**

```
avr-gcc -mmcu=atmega128 main.o -o main.out
```

The resulting file after linking is main.out. The **–o** option is used to specify the output file. Please note that using avr-gcc for linking is preferable.

**Step 3: Creating hex file**

```
avr-objcopy  -O ihex main.out main.hex
```

We have final output in our hands as main.hex. We can program the MCU with it. We can also use this program under any AVR Simulator. Programming the microcontroller is discussed later in this chapter.

## 5.4 Programming the Micro-controller

**avrdude** utility, part of WinAVR can be used to program the microcontroller.

Following is the command to use AVRDude program the Atmega-128 microcontroller connected via JTAG programmer on COM Port 4.

**Step 4: Burning the hex file**

```
avrdude -p atmega128 -P com4 -c jtagmkI  -U flash:w:main.hex
```

## 5.5 The Make Utility

As the size of project grows, it becomes difficult to maintain and recompile every source file manually. Here comes the life saver, the **make** utility. Make is a utility which automates the build process of a project. It works by finding a special file called **makefile**, which has instructions to automate the build process. This makefile has to be created by us manually. Once we have written the makefile, it helps us save a lot of time and effort.

### 5.5.1 Building makefile

A makefile consists of steps of instructions given to compiler tool-chain, set of Macros which can be used while compilation.

#### *5.5.1.1 Defining Makefile Macros*

Macros play very important role in makefile definition. We can define Macros at the beginning of the file and use them throughout.

```
CC=avr-gcc
```

This Macro is telling that we are using avr-gcc for compilation. Macros in makefile are quite similar to Macros in C.

Once defined, we can use the Macro by appending a **"$"** sign with the Macro name enclosed in brackets.

```
$(CC) -mmcu=atmega128 main.c
```

One major advantage of using Macros is that, if we need to change some setting, we can simply change the Macro value. Rest of the program remains the same.

#### *5.5.1.2 Defining Compilation Procedure (Make targets)*

A makefile is just a sequence of commands similar to a batch file. We can define multiple sets of instructions each doing some specific job. For instance, we can define a label **hex**, which will generate hex file as output. We can have a label **main**, which will generate object file of main.c.

As we manually compiled the main.c earlier, now we can combine those steps under a label.

```
main:
        avr-gcc main.c  -mmcu=atmega128  -c
        avr-gcc -mmcu=atmega128 main.o
        avr-objcopy  -O ihex a.out main.hex
```

**Don't forget to add a Tab like this before every command.**

#### 5.5.1.3 Makefile Comments

We can also add comments in makefile, so that we can explain purpose of a command used in compilation. Any line starting with **"#"** is called a comment.

```
#My first makefile
```

#### 5.5.1.4 Final Makefile

The final makefile which will compile main.c is shown as:

**makefile**

```
#Makefile to compile main.c
CC=avr-gcc
LD=avr-gcc
MCU=atmega128
CFLAGS=-Wall -c
SRC=main.c
OUT=main.hex
PORT=com4

main:
	$(CC) -mmcu=$(MCU) $(CFLAGS) $(SRC)
	$(LD) -mmcu=$(MCU) *.o
	avr-objcopy -O ihex a.out $(OUT)
program:
	avrdude -p $(MCU) -P $(PORT) -c jtagmkI -U flash:w:$(OUT)
clean:
	rm *.o *.hex *.out
```

The only difference here is that we are using Macros as explained earlier. Also, we have defined multiple targets as main, program, clean.

### 5.5.2 Compiling programs using make

Once we are done with writing makefile, we can simply invoke the **main** label.

```
make main
```

Make, by default is set to execute the very first label automatically. That is, the above command could also have been written like:

```
make
```

If you wish to clean up the previously compiled files, we can issue a command:

```
make clean
```

This will result into removing all the object files and hex file.

This was just a brief introduction to make utility and makefiles. Please read the make documentation for further help on make.

Makefile can be generated automatically using mFile utility. This utility is available within WinAVR.

## 5.6 IDE – Integrated Development Environment

An IDE is a software which encapsulates all the required utilities for creating a project. The basic features provided by IDE are:

1. **Editor =>** Used for writing the source code. Some good editors also provide Intellisense. Intellisense is a feature which autocompletes the code as you type.
2. **Toolchain =>** Toolchain is complete set of utilities required for compiling, linking, creating hex file etc.
3. **Debugger =>** Debugger is also built-in with the IDE.
4. **Other Utilities =>** Some other supporting utilities are also available which help in speeding up the development process.

In this section, we will be working on the standard IDE provided by Atmel, AVR Studio. We will learn how we can create a Project, how to build, program and debug the microcontroller.

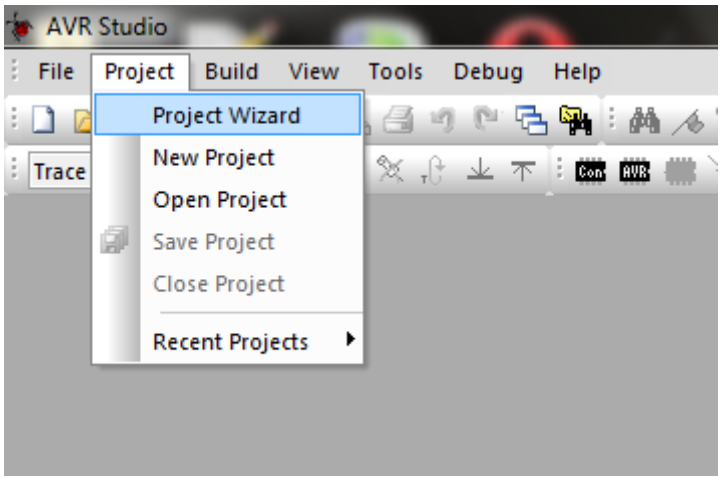
| ? | **Did you Know** |
|---|---|
| Various other IDEs are also available which can be used for AVR programming. Example: Eclipse, CodeVisionAVR. | |

## 5.7 Introduction to AVR Studio

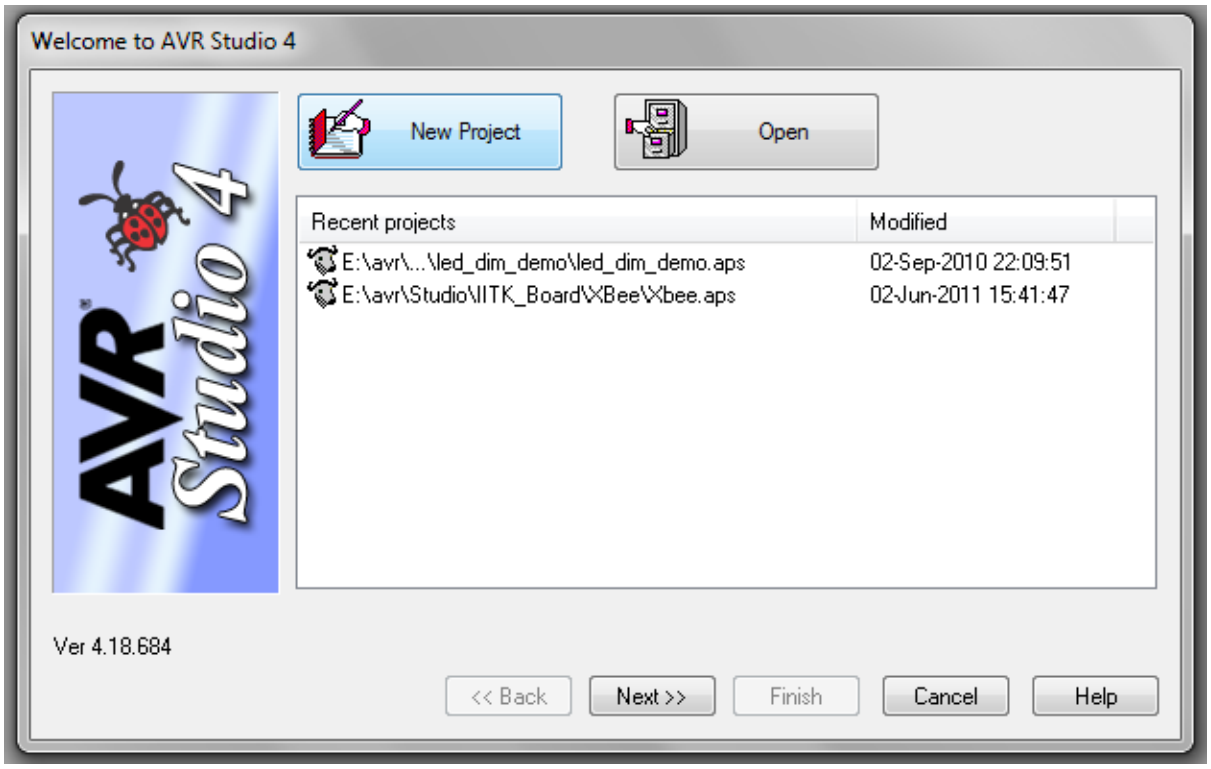
AVR Studio is used by embedded programmers for programming and debugging for many of the Atmel microprocessors such as the Atmega8 or even the Atmega128. While it has support for assembly programming for those who prefer to use higher languages, it uses the coff format for debugging. Beginning with version 4 AVR Studio has now moved to dwarf2, and can be more readily used in conjunction with the open source gcc based compiler WinAVR.

## 5.8 Creating new Project

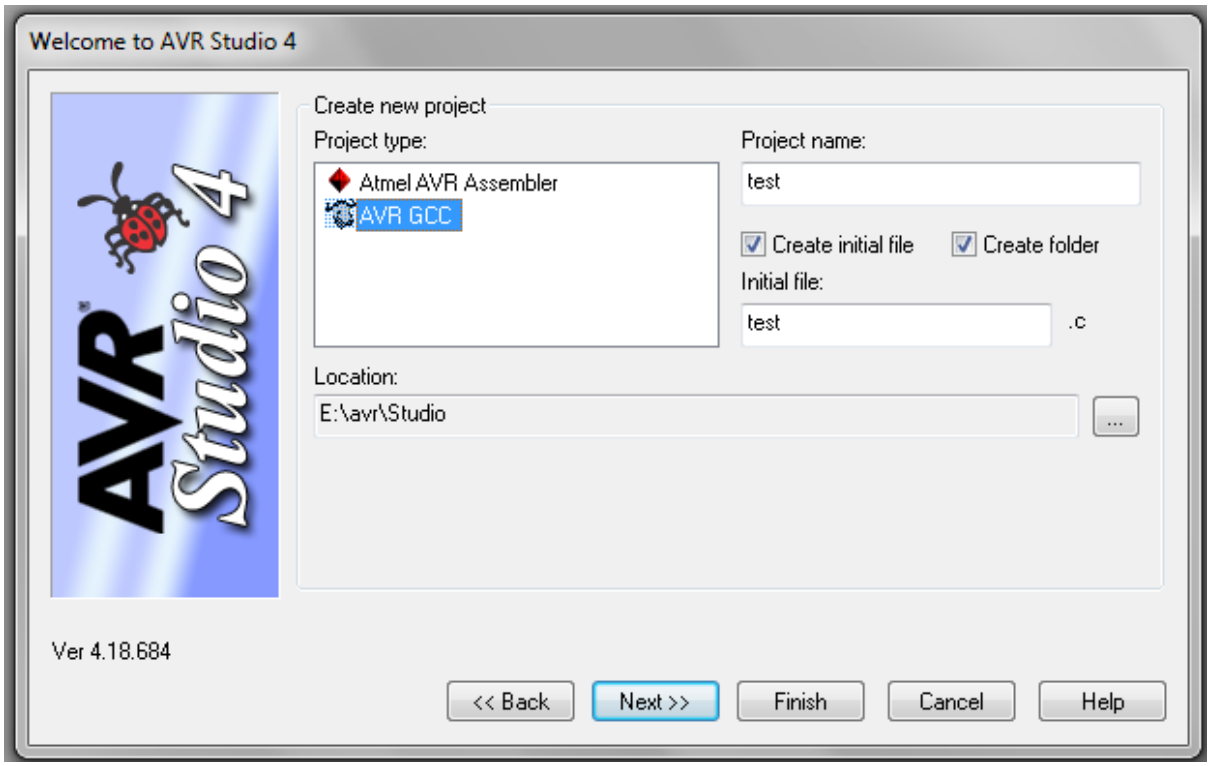
1. Open AVR Studio and go to Project → Project Wizard.



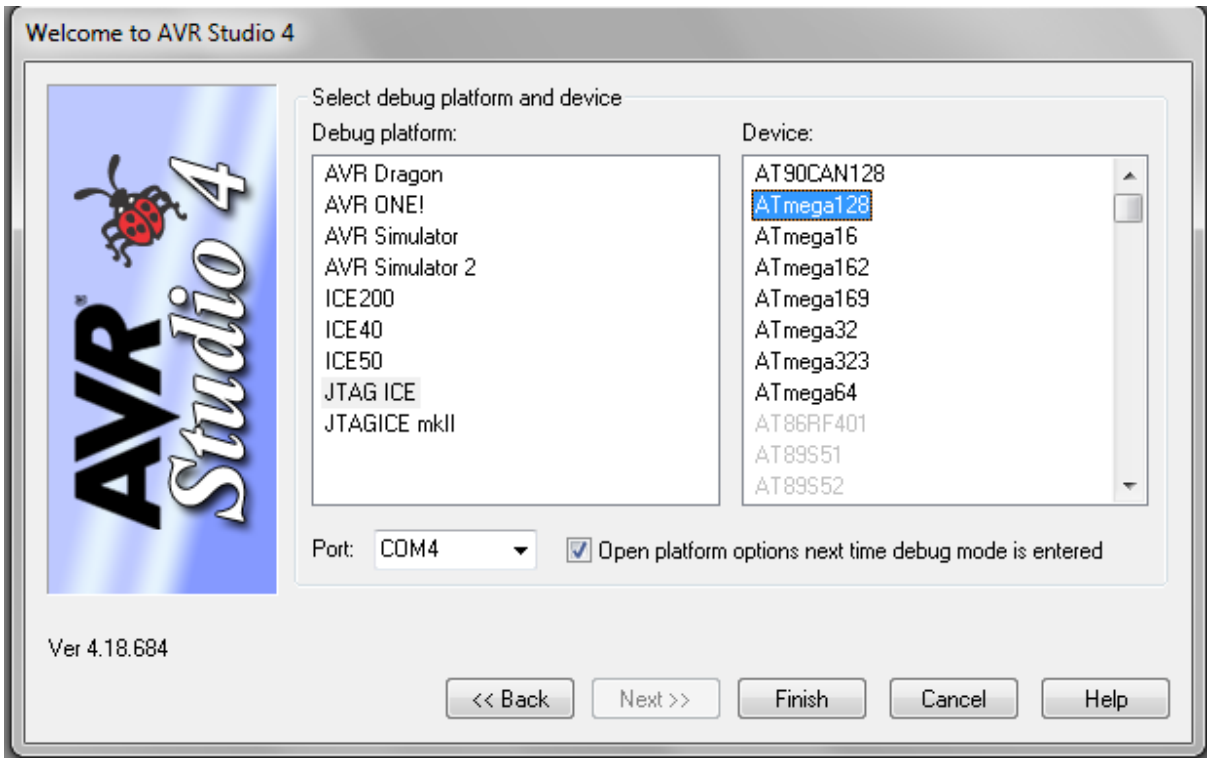

2. Click on New Project.



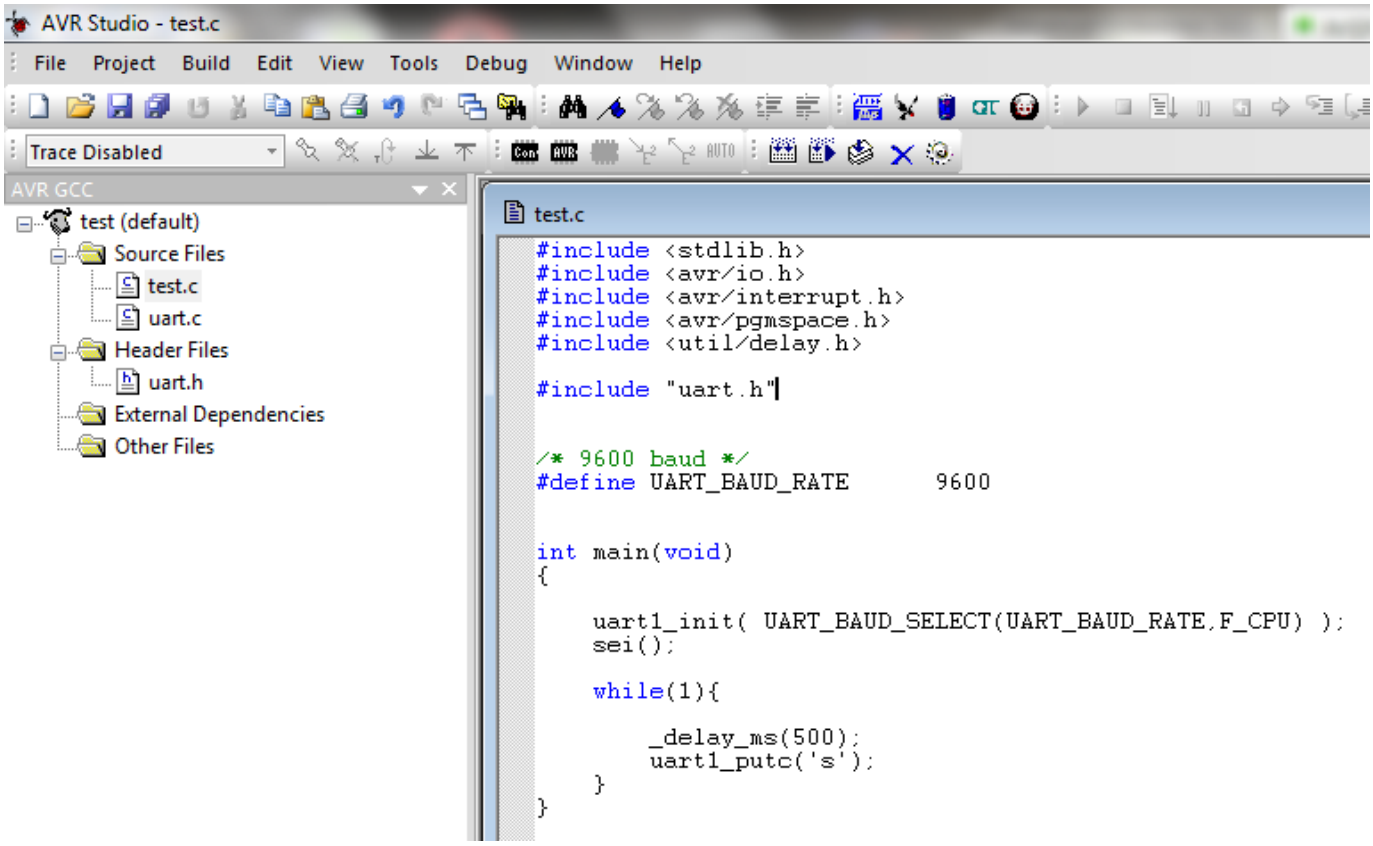
3. Choose AVR-GCC and give path for project and type Project name. Click on Next.



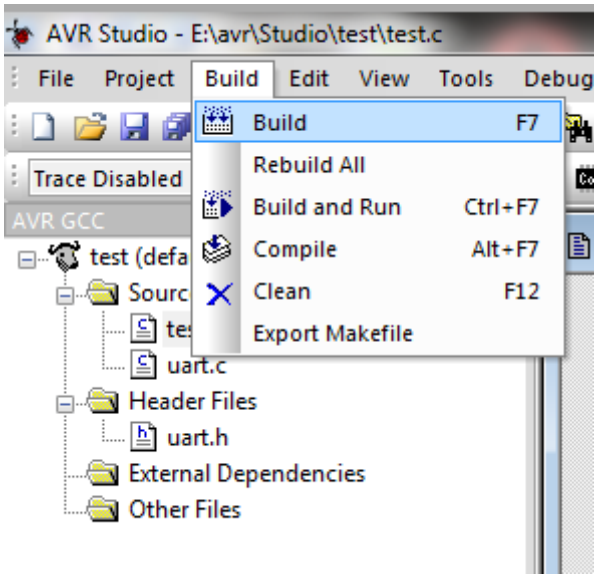

4. Choose Debug Platform: JTAG ICE, Device: Atmega128, Port as per your system. You can check platform options. Click on Finish.



5. Write the code in test.c
6. Copy additional libraries to project folder and include .c files to Source Files and .h files to Header Files.

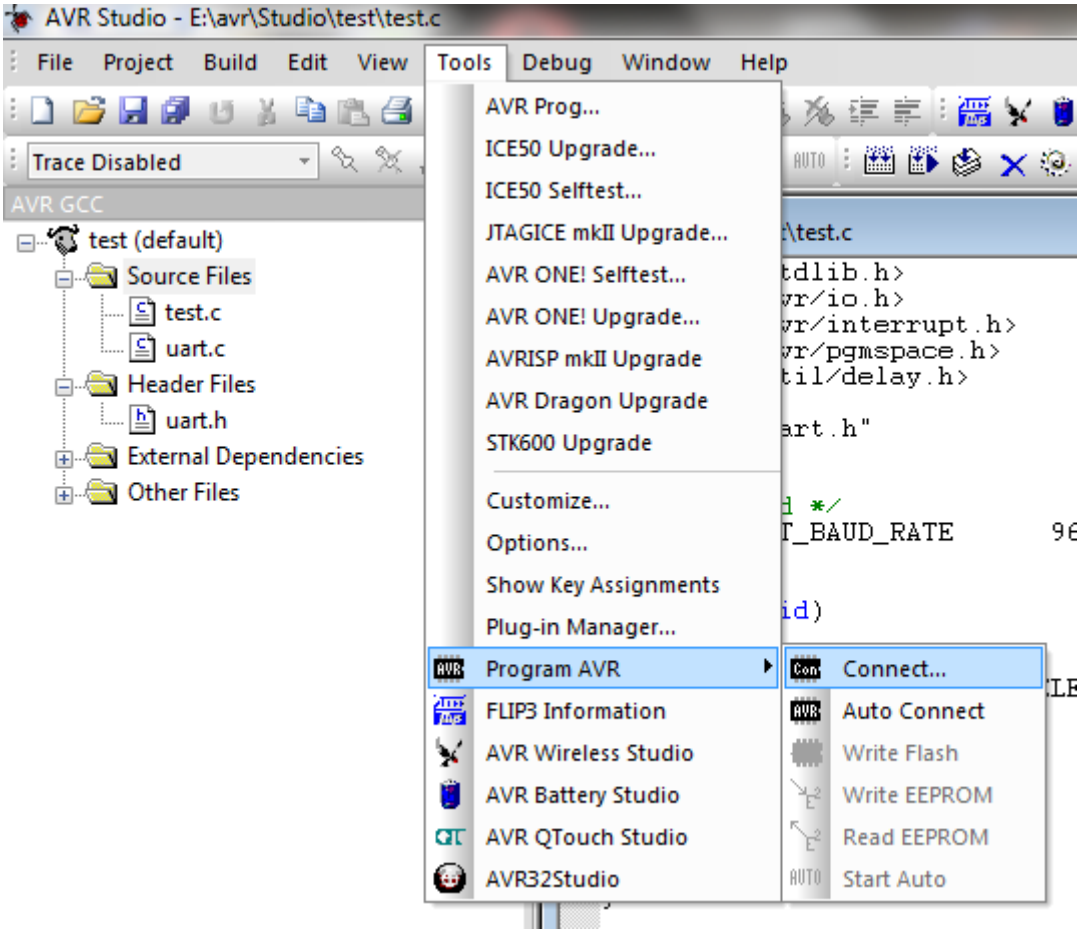



7. Go to Build → Build/Rebuild All to compile-link-object_copy and create hex file.

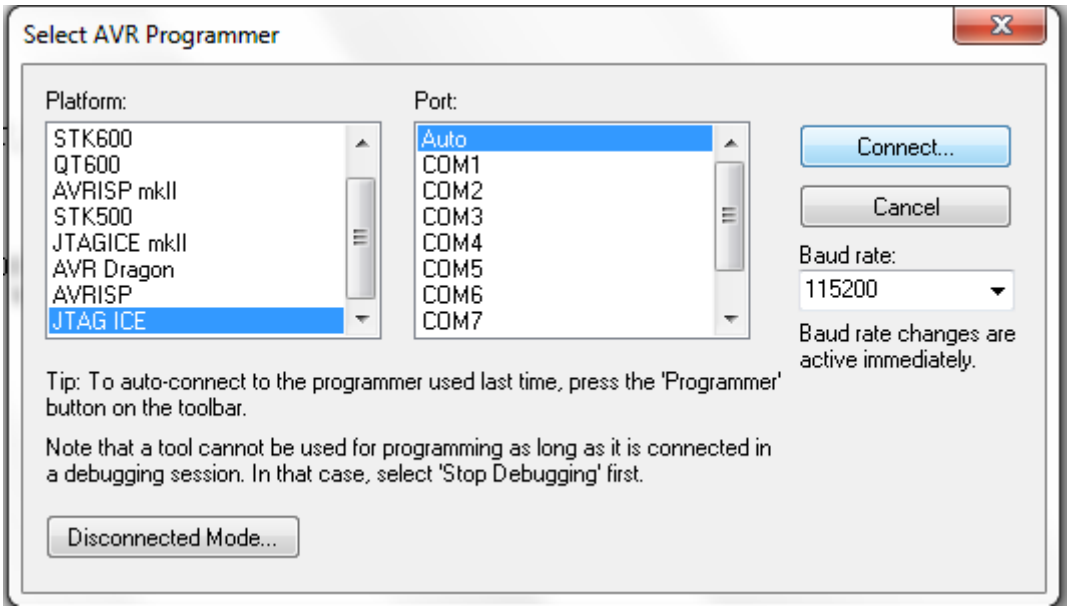


## 5.9 Burning hex file using AVR Studio.

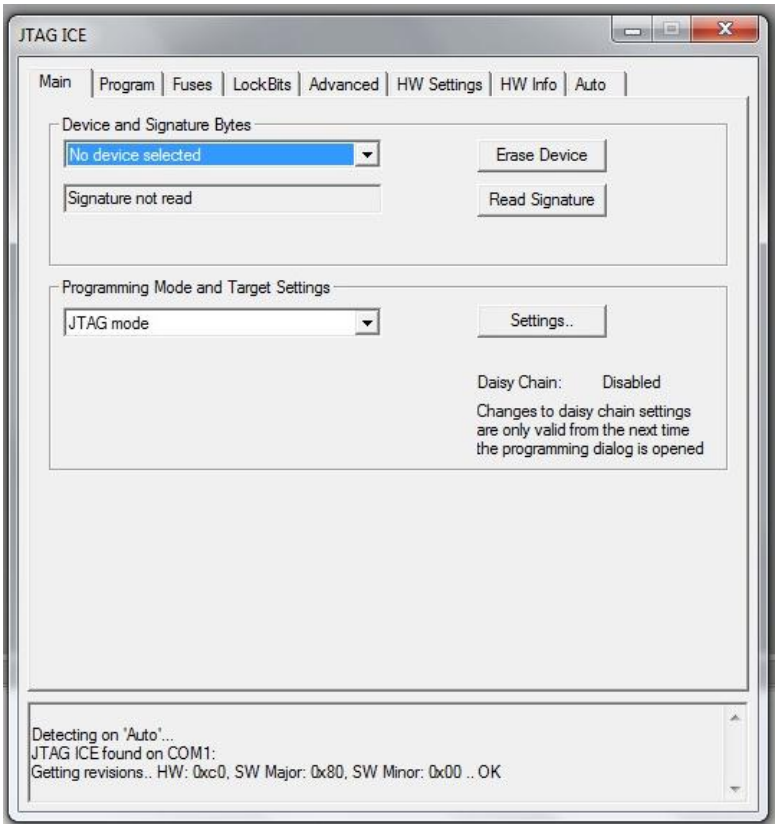
1. Go to Tools → Program AVR → Connect



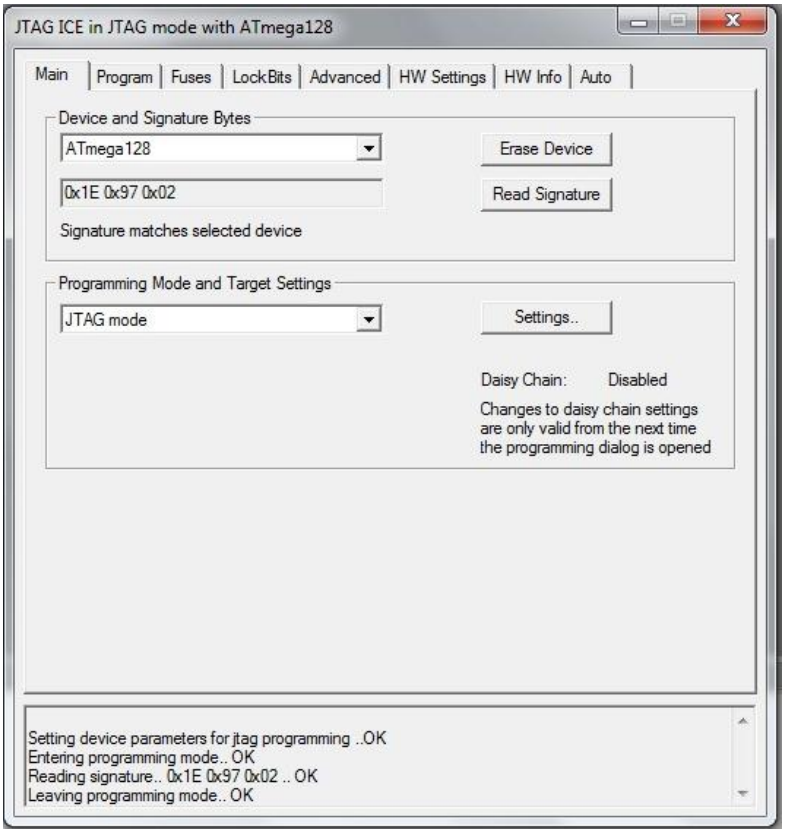

2. Select Platform as: **JTAG ICE**, select correct **COM Port** (you can also choose **Auto**) and click on **Connect.** (Make sure Board's power is ON and JTAG cable is connected in the correct orientation)



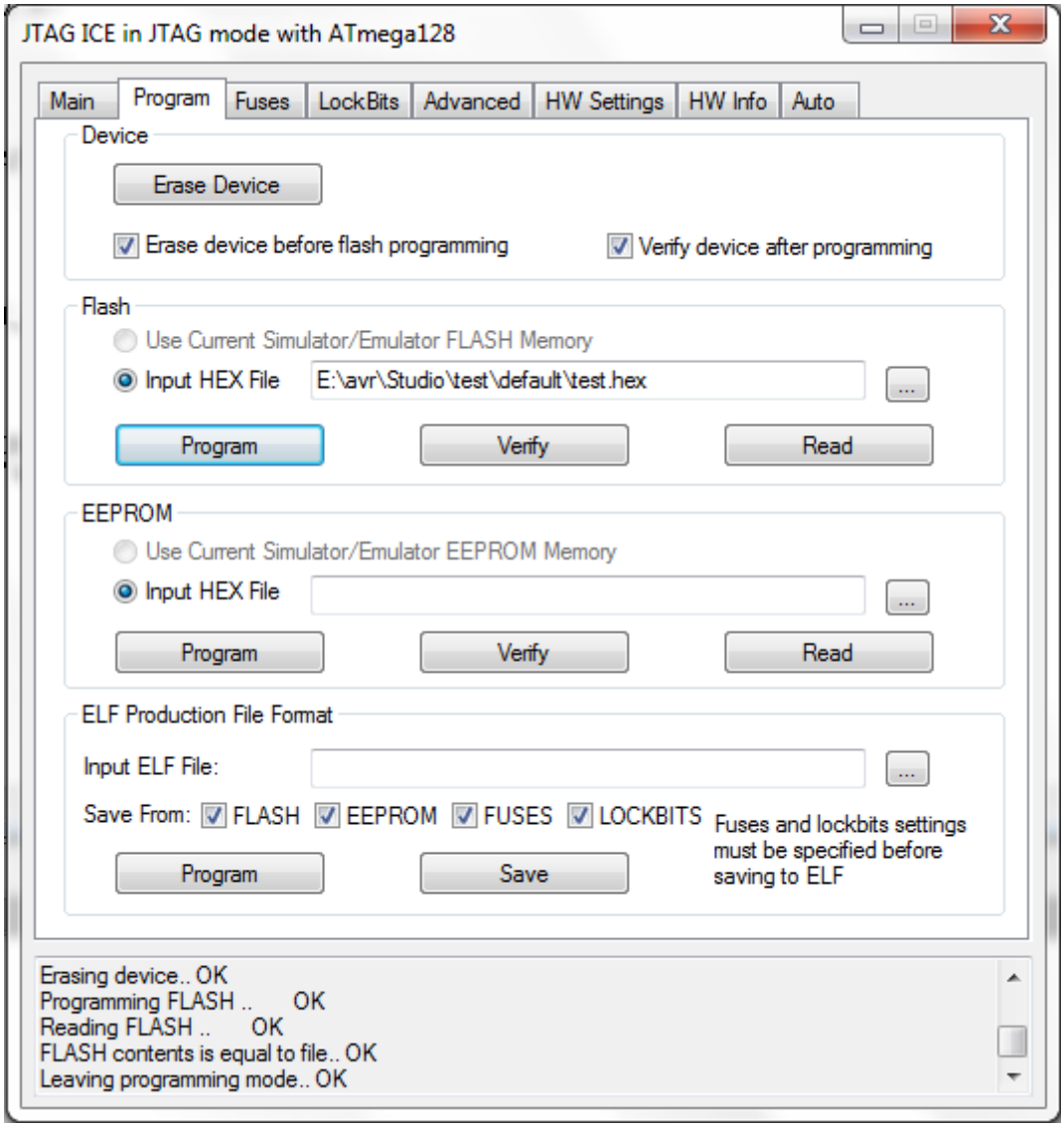
3. Following window will open,




5. Chose Device as: **ATmega 128** and click on **Read Signature** to verify the chip signature. It should return **OK** and show the message**: Signature matches selected device.**



6. To burn a **.hex** file, go to Program, browse the file and click on Program.




10. To know more details and about JTAG debugging steps, please follow Atmel's JTAG ICE manual,

    www.atmel.com/atmel/acrobat/doc2475.pdf

# Chapter 6 - Embedded C Review

## 6.1 Introduction

In this section, we will be discussing a limited set of features of C language. The features we will be studying are:

1. MACRO definitions
2. Typedef Keyword
3. Bit-Wise operators
4. Pointers

## 6.2 Macro definitions

MACROs are constants which can be used to name an expression/statement. Whenever a MACRO is called, it gets expanded to its original definition. MACRO processing is done by a Program called Preprocessor. The process of preprocessing is done before compilation.

**MACRO demo**

```
#define PI 3.14
#define ADD(X,Y) (X)+(Y);

int main() {
        printf("%f",PI);            // Using PI macro
        int result = ADD(5,6);      // Using ADD macro
}
```

Here, we have created 2 MACROs. The first macro is just a constant value, which gets replaced before compilation. The second definition works like addition function to add X and Y. However, please note that MACROs are completely different from functions.

MACROs are called before Compilation process, while functions are used at runtime.

## 6.3 Typedef Keyword

Typedef keyword is used to create new data types from existing data types. As an example, we can use it to create a new data type **uint_demo**, which will work as unsigned int.

**Use of typedef Keyword**

```
typedef unsigned int uint_demo;
uint_demo x;
```

Here, we have used typedef to create a new data-type as **uint_demo**. The uint_demo data type mimics unsigned int data-type.

AVR Library contains a lot of typedef definitions. Checkout the header files of AVR to know more about it.

## 6.4 Bit-Wise Operators

Bitwise operators can be used to directly manipulate the bits of int/char data-types. C provides us a variety of such operators which are shown as:

| Operation | Operator Symbol in C | Operation Mode |
|---|---|---|
| **AND** | & | Binary |
| **OR** | \| | Binary |
| **XOR** | ^ | Binary |
| **1's Complement** | ~ | Unary |
| **Left Shift** | << | Binary |
| **Right Shift** | >> | Binary |

Binary operators require 2 operands to work, while unary operator requires only 1 operand to operate.

### 6.4.1 AND, OR, XOR operators

The AND, OR, XOR operations work according to following Truth tables.

| Input A | Input B | AND | OR | XOR |
|---|---|---|---|---|
| **0** | **0** | 0 | 0 | 0 |
| **0** | **1** | 0 | 1 | 1 |
| **1** | **0** | 0 | 1 | 1 |
| **1** | **1** | 1 | 1 | 0 |

Here we see that AND gives 1 only if both inputs are 1, OR gives 1 if any of input is 1. While, XOR gives 1 **if and only if** one of the input is 1.

**Example #1 : Bitwise AND, OR, XOR**

```
int a = 46, b = 23;
int x =a & b;
int y = a | b;
int z = a ^ b;
```

The values of x,y,z variables can be shown in the form of table as:

| Variable | Binary Form | Integer Value | Operation Applied |
|---|---|---|---|
| **a** | **0000000000101110** | **46** | **None** |
| **b** | **0000000000010111** | **23** | **None** |
| x | 0000000000000110 | 6 | AND |
| y | 0000000000111111 | 63 | OR |
| z | 0000000000111001 | 57 | XOR |

### 6.4.2 Complement Operator(~)

The purpose of this operator is to invert all the bits of the operand. Inversion basically means that all the 0's will now become 1's. All the 1's will now become 0's.

**Example #2: Complement Operator**

```
int a = 6;
int x =~a;
```

| Variable | Binary Form | Value |
|---|---|---|
| **a** | **0000000000000110** | **6** |
| x | 1111111111111001 | -7 |

### 6.4.3 Shifting Operators

The shift operators are used for shifting the bits of a variable by some value x.

**Example #3 : Shifting Operators**

```
int a = 108, b = 48;
int x =a << 2;          // Left Shift Operation
int y = b >> 1;         //Right Shift Operation
```

Here, the bits of variable **a** are shifted leftwards 2 times. The result can be illustrated as:

| Left Shift Operation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Bit N** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Value(2^N)** | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **a** | 0 | **1** | **1** | **0** | **1** | **1** | **0** | **0** |
| **x** | **1** | **0** | **1** | **1** | **0** | **0** | **0** | **0** |

Result=> 176

The Bit number 6 now becomes at Bit number 8 and hence, is removed due to overflow. Similarly every other bit is shifted leftwards two times.

| Right Shift Operation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Bit N** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Value(2^N)** | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **b** | 0 | 0 | **1** | **1** | **0** | **0** | **0** | **0** |
| **y** | 0 | 0 | 0 | **1** | **1** | **0** | **0** | **0** |

Result=> 24

Here, the bit number 6 now becomes the $5^{th}$ and similarly all other bits are shifted in right direction. Hence, the right shift operation results in 24 as output.

**? Did you Know**

The left shift operator can be used when multiplying a value by 2 or a multiple of 2. Similarly, Right Shift operator can be used to divide by 2 or multiple of 2.

## 6.5 Bit Operations on Registers

### 6.5.1 Setting a bit

Use the bitwise OR operator (|) to set a bit.

```
REGA |= 1 << x;
```

That will set bit x of register REGA.

### 6.5.2 Clearing a bit

Use the bitwise AND operator (&) to clear a bit.

```
REGA &= ~(1 << x);
```

That will clear bit x of `REGA`. You must invert the bit string with the bitwise NOT operator (~), then AND it.

### 6.5.3 Toggling a bit

The XOR operator (^) can be used to toggle a bit.

```
REGA ^= 1 << x;
```

That will toggle bit x of `REGA`.

### 6.5.4 Checking/Reading a bit

To check a bit, AND it with the bit you want to check:

```
bit = REGA & (1 << x);
```

That will put the value of bit x of `REGA` into the variable bit.

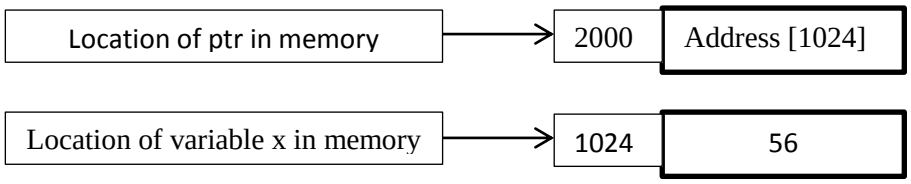
## 6.6 Pointers

Pointers are special variables which store address of another memory location instead of storing some value. Two important operators used in pointers are value-at (*), address-of (&).

The size of a pointer does not depend on the type of data-type on which it is pointing. Its size is always going to be equal to size of int.

| **Pointer Demo** | |
|---|---|
| char x = 56; | |
| char *ptr; | // An char pointer |
| *ptr = &x; | // The operator address-of(&) is used to retrieve address of variable x |
| char out = *ptr; | // retrieves the value pointed by ptr |
| printf("%d",ptr); | // Prints the address of pointer ptr |

In this example, we create a pointer to a character variable x. Graphically, this can illustrated as:




Here, the pointer **ptr** is stored at location 2000 in memory. The content of location 2000 is the address of variable **x**, that is 1024. Hence, the pointer points to memory location of **x** variable. When the value pointed by a ptr has to be used, we use ***ptr** to use its value.

### 6.6.1 Pointer Operations

We can only use addition and subtraction operations on pointers. The use of Multiplication/division or any other operation on pointer will throw an error.

| Operation | Operators allowed | Effect of operation |
|---|---|---|
| **Addition** | +, ++ | Address of pointer is incremented |
| **Subtraction** | -, -- | Address of pointer is decremented |

As an example, if the initial address of a pointer, *ptr is 2030. Then, *ptr++ will cause the ptr to point to 2032 memory address.

Pointers can be summarized as:

| Pointer Type | Size of Pointer | Data Type | Size of variable | Example |
|---|---|---|---|---|
| **char*** | 2 | Char | 1 | char *ptr; |
| **int*** | 2 | Int | 2 | int *ptr; |
| **long int*** | 2 | long int | 4 | long *ptr; |

We have covered the very basics of pointers. We can also have pointer to another pointer variable. Also, some other complex forms may be there. Covering each aspect of pointers in not possible here. For more information on pointers, check out http://oreilly.com/catalog/pcp3/chapter/ch13.html.

Some C tips and techniques are described in the following Manual provided by Atmel. Interested readers can have a look.

www.atmel.com/atmel/acrobat/doc1497.pdf

# Chapter 7 - GPIO (General Purpose Input Output)

So let's start with understanding the functioning of AVR. We will first discuss about I/O Ports. See the pin configuration of Atmega-16.

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)
- PE0 - PE7 (PORTE)
- PF0 - PF7 (PORTF)
- PG0 – PG4 (PORTG)

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 PORTF)
- UART0 (RxD, TxD on PORTE)
- TIMERS (OC0 - OC2)
- SPI (MISO, MOSI, SCK on PORTB)
- External Interrupts (INT0 - INT2)

## 7.1 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.

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.

The I/O Ports are defined in **<avr/io.h>.** You need to include this header file to work with I/O Ports.

## 7.2 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.

## 7.3 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.

### 7.3.1 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;**

### 7.3.2 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

Tri-stated 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.

## 7.4 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.

## 7.5 Bit Operations

Use Right and Left Shift operators with binary Bit operators for setting, clearing, toggling, and reading any particular bit of a register.

For Example, Clearing 5$^{th}$ bit of DDRC register can be done in following way,

```
DDRD &= ~ (1 << 5);
```

# Chapter 8 - LCD Interfacing

## 8.1 Introduction

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.

## 8.2 Overview of LCD Display

LCD displays are widely used in many applications like mobile phones, robotics, DVD players, Measurement instruments etc. Intelligent LCD displays are very capable because they can display complete ASCII character set and even graphics. These displays are easily connected with micro controller and microprocessors. LCD displays are complete embedded system in them, because it include microcontroller, RAM and ROM.


**16X2 LCD DISPLAY**

LCD Modules can present textual information to user. It's like a cheap "monitor" that you can hook in all of your gadgets.

They come in various types. The most popular one is 16x2 LCD module. It has 2 rows & 16 columns.

The intelligent displays are two types:

**a)** Text Display

**b)** Graphics Display

Text display can display all character set and graphics display can show any graphics because they are interfaced pixel wise.

In recent year the LCD is finding widespread use replacing LEDs (seven segment LEDs or multi-segment LEDs).

This is due to the following reasons:

**a)** The declining prices of LCDs.

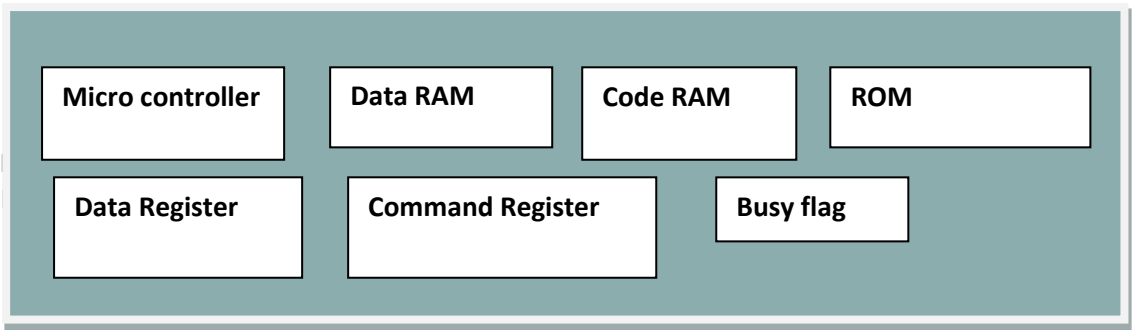
**b)** The ability to display the numbers, characters and graphics. This is not possible in LEDs, which can display the numbers and few characters.

**c)** Incorporation of a refreshing controller into the LCD, Thereby reliving the CPU of the task of refreshing the LCD. In contrast, the LED must be refreshed by the CPU (or in some other way) to keep displaying the data.

The interfacing of LCD is quite difficult. But we will try to make it simple and let us explain it for you.

We will learn how to interface the text intelligent LCD display. These displays are available in the market of 16 column and one Row and more than one row displays.

## 8.3 Block Diagram of LCD Display




**DATA RAM:**
This RAM is storing the ASCII values of corresponding characters which will be displayed on the LCD. For each column there is one location in the RAM. When we will store the ASCII value at that location than its corresponding character will be displayed on the screen.

**CODE RAM:**
This RAM stores the binary pattern according to the character.

**ROM:**
This ROM stores the binary pattern which is according to the Pixels of LCD and there are patterns of every character.

**COMMAND REGISTER:**
It stores various commands for proper functioning.

**DATA REGISTER:** This register work as buffer for data lines and the internal buses of LCD. The ASCII values of characters will be given to the data register.

**BF (BUSY FLAG):**
It indicates the internal working of the LCD. It show whether LCD is busy in any operation or not.
If BF=0 (LCD is idle we can proceed for next operation)
If BF=1 (LCD is busy we cannot proceed for next operation and we have to wait unless operation completes).

## 8.4 Description of Pins

**VCC, VSS and VO:**
While VCC and VSS provide +5 Volts and ground, respectively VEE is used for controlling LCD contrast

**RS (REGISTER SELECT):**
The RS pin is used to select Data Register or Command Register.
If RS=0, CR Register is selected, allowing the user to send a command such as clear display,curser at the home etc.
If RS=1, DR Register is selected, allowing the user to send data to be display on the LCD.

**R/W (READ/WRITE):**
When R/W=0, Write operation..
When R/W=1 Read Operation

**EN (ENABLE):**
The Enable pin is used by the LCD to latch binary bits available on its data pins. When data is supplied to data pins, a negative edge is applied to this pin So that the LCD latches in the data present at the data pins. This pulse must be a minimum of 450 ns wide.
There should be positive edge at EN pin when read operation is required.

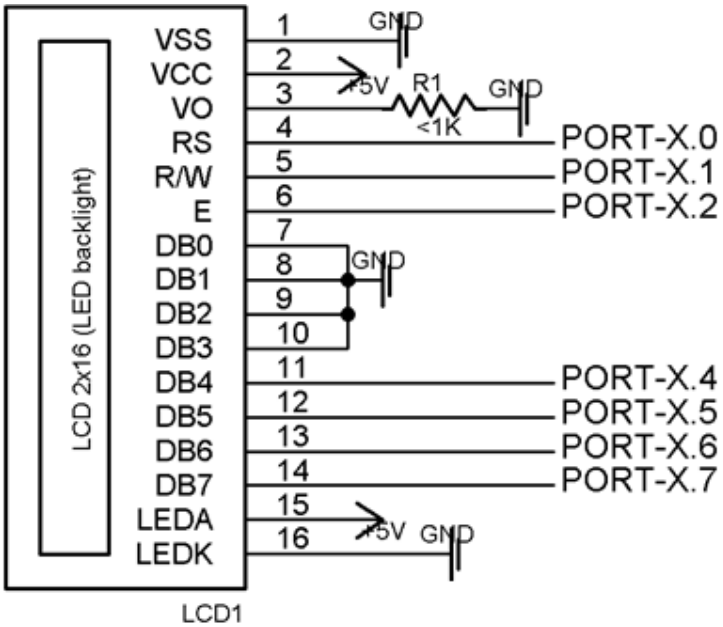
**D7-D0:**
This is 8-bit data pins. D7-D0 are used to send information to the LCD or read the contents of the LCD's internal registers.

**BK-LED (LEDA, LEDK):**
These pins are used to give the supply to the back light of the LCD display. So, that content of the LCD display can be viewed in the dark.

## 8.5 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:




**LCD Connections**

## 8.6 Data Communication Functions

We will be using Peter Fleury's libraries for the programming of LCD functions. The required files for UART are lcd.c, lcd.h. The functions available in these files are described as:

| | |
|---|---|
| **#1** | `void lcd_init(uint8_t dispAttr);` |
| **Info** | Initialize display and select type of cursor. |
| **#2** | `void lcd_clrscr(void);` |
| **Info** | Clear display and set cursor to home position. |
| **#3** | `void lcd_home(void);` |
| **Info** | Set cursor to home position. |
| **#4** | `void lcd_gotoxy(uint8_t x, uint8_t y);` |
| **Info** | Set cursor to specified position.<br>The position is given by x and y parameters. |
| **#5** | `void lcd_putc(char c);` |
| **Info** | Display character at current cursor position. |
| **#6** | `void lcd_puts(const char *s);` |
| **Info** | Display string without auto linefeed. |
| **#7** | `void lcd_puts_p(const char *progmem_s);` |
| **Info** | Display string from program memory without auto linefeed. |
| **#8** | `void lcd_command(uint8_t cmd);` |
| **Info** | Send LCD controller instruction command. |
| **#9** | `void lcd_data(uint8_t data);` |
| **Info** | Send data byte to LCD controller. |

## 8.7 Definitions for 4-bit IO mode

Change LCD_PORT if you want to use a different port for the LCD pins.

The four LCD data lines and the three control lines RS, RW, E can be on the same port or on different ports. Change LCD_RS_PORT, LCD_RW_PORT, LCD_E_PORT if you want the control lines on different ports.

Normally the four data lines should be mapped on one port, but it is possible to connect these data lines in different order or even on different ports by adapting the LCD_DATAx_PORT and LCD_DATAx_PIN definitions.

```
#define     LCD_PORT            PORTF
#define     LCD_DATA0_PORT      LCD_PORT
#define     LCD_DATA1_PORT      LCD_PORT
#define     LCD_DATA2_PORT      LCD_PORT
#define     LCD_DATA3_PORT       LCD_PORT
#define     LCD_DATA0_PIN       4
#define     LCD_DATA1_PIN       5
#define     LCD_DATA2_PIN       6
#define     LCD_DATA3_PIN       7
#define     LCD_RS_PORT         LCD_PORT
#define     LCD_RS_PIN           0
#define     LCD_RW_PORT         LCD_PORT
#define     LCD_RW_PIN          1
#define     LCD_E_PORT          LCD_PORT
#define     LCD_E_PIN           2
```

## 8.8 Definitions for Display Size

Change these definitions to adapt setting to your display,

```
#define     LCD_LINES           2
#define     LCD_DISP_LENGTH     16
#define     LCD_LINE_LENGTH     0x40
#define     LCD_START_LINE1     0x00
#define     LCD_START_LINE2      0x40
#define     LCD_START_LINE3     0x14
#define     LCD_START_LINE4     0x54
#define     LCD_WRAP_LINES      0
#define     LCD_IO_MODE         1
```

For Details Visit: http://homepage.hispeed.ch/peterfleury/group__pfleury__lcd.html

# Chapter 9 - Accessing internal EEPROM

Most of the AVRs in Atmel's product line contain at least *some* internal EEPROM memory. EEPROM, short for *Electronically Erasable Read-Only memory*, is a form of non-volatile memory with a reasonably long lifespan. Because it is non-volatile, it will retain its information during periods of no AVR power and thus is a great place for storing sparingly changing data such as device parameters.

The AVR internal EEPROM memory has a limited lifespan of 100,000 writes - reads are unlimited.

**How it is accessed?**

The AVR's internal EEPROM is accessed via special registers inside the AVR, which control the address to be written to (EEPROM uses byte addressing), the data to be written (or the data which has been read) as well as the flags to instruct the EEPROM controller to perform a write or a read.

The C language does not have any standards mandating how memory other than a single flat model (SRAM in AVRs) is accessed or addressed. Because of this, just like storing data into program memory via your program, every compiler has a unique implementation based on what the author believed was the most logical system.

## 9.1 Using the AVRLibC EEPROM library routines:

The AVRLibC, included with WinAVR, contains prebuilt library routines for EEPROM access and manipulation. Before we can make use of those routines, we need to include the eeprom library header:

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

At the moment, we now have access to the eeprom memory, via the routines now provided by eeprom.h. There are three main types of EEPROM access: byte, word and block. Each type has both a write and a read variant, for obvious reasons. The names of the routines exposed by our new headers are:

**uint8_t** eeprom_read_byte (const uint8_t *addr)

**void** eeprom_write_byte (uint8_t *addr, uint8_t value)

**uint16_t** eeprom_read_word (const uint16_t *addr)

**void** eeprom_write_word (uint16_t *addr, uint16_t value)

**void** eeprom_read_block (void *pointer_ram, const void *pointer_eeprom, size_t n)

**void** eeprom_write_block (void *pointer_eeprom, const void *pointer_ram, size_t n)

For more details see:
http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=38417

# Chapter 10 - UART Communication

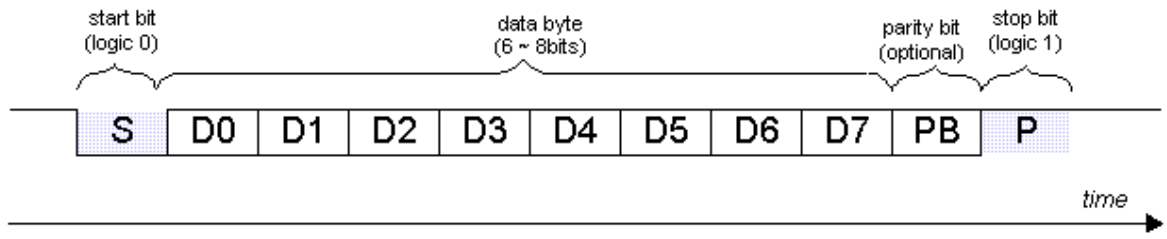
## 10.1 Introduction

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.

## 10.2 UART: Theory of Operation

Figure 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.




**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**.

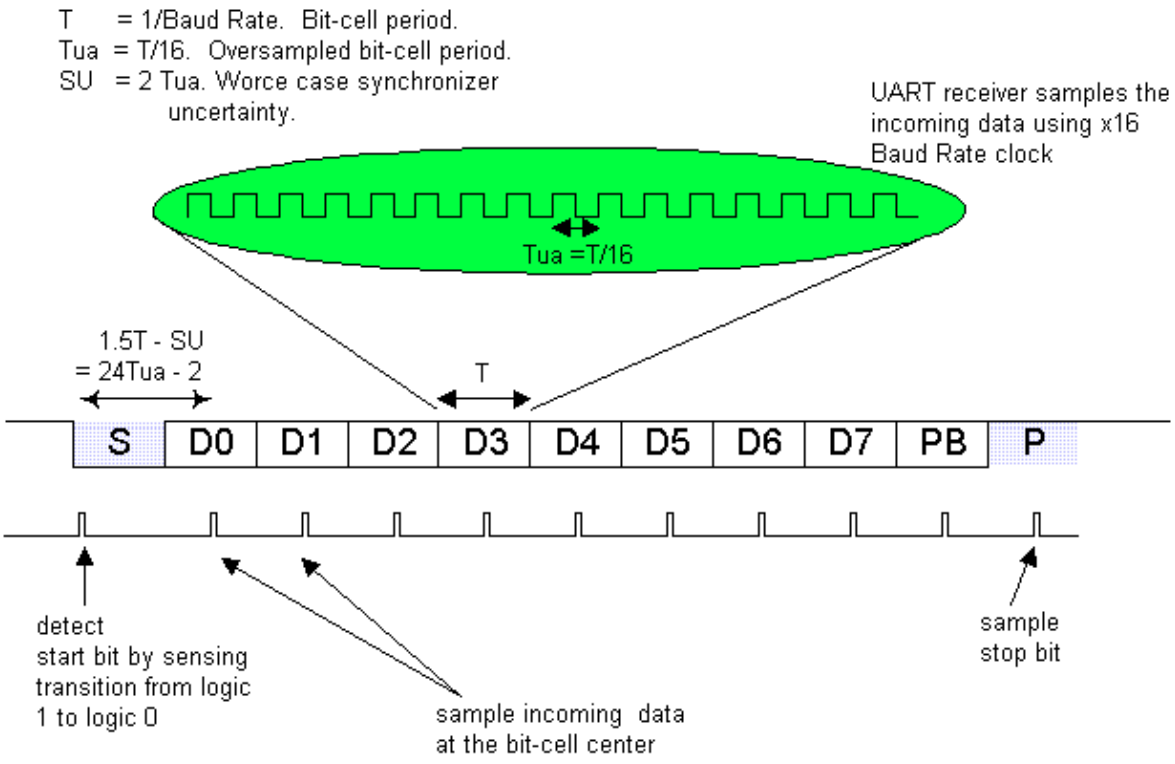
Since no common clock is shared, a known data transfer rate (baud rate) must be agreed upon prior to data transmission. That is, the receiving UART needs to know the transmitting UART's baud rate (and conversely the transmitter needs to know the receiver's baud rate, if any). In almost all cases the 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 "centre" can be assured to be 24 ticks minus 2 (worse case synchronizer uncertainty) later. From then on, every next data bit centre is 16 clock ticks later. Figure 2 illustrates this point.

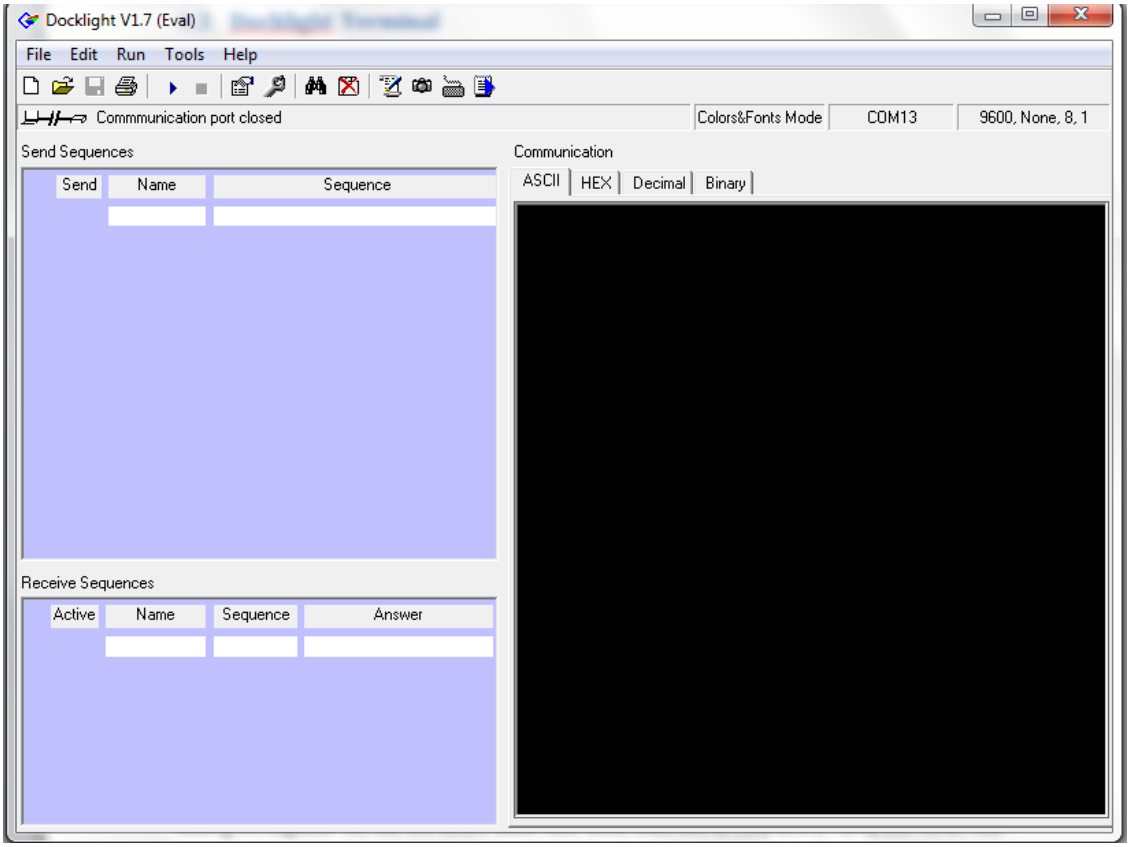
Once the start bit is detected, the subsequent data bits are assembled in a de-serializer. Error condition maybe generated if the parity/stop bits are incorrect or missing.




**Data sampling points by the UART receiver.**

## 10.3 Docklight Terminal

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.




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 Atmega128. 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,

```
uart1_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 Atmega128, 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.

## 10.4 Data Communication Functions

We will be using Peter Fleury's libraries for the programming of UART protocol. The required files for UART are uart.c, uart.h. The functions available in these files are described as:

| | |
|---|---|
| **#1** | `void uart_init(unsigned int baudrate);` |
| **Info** | Initialize UART and set baudrate.<br>Specify baudrate using macro UART_BAUD_SELECT() |
| **#2** | `unsigned int uart_getc(void);` |
| **Info** | Get received byte from ringbuffer. |
| **#3** | `void uart_putc(unsigned char data);` |
| **Info** | Put byte to ringbuffer for transmitting via UART. |
| **#4** | `void uart_puts(const char *s);` |
| **Info** | Put string to ringbuffer for transmitting via UART. |
| **#5** | `void uart_puts_p(const char *s);` |
| **Info** | Put string from program memory to ringbuffer for transmitting via UART. |
| **#6** | `void uart1_init(unsigned int baudrate);` |
| **Info** | Initialize USART1 (only available on selected ATmegas). |
| **#7** | `unsigned int uart1_getc(void);` |
| **Info** | Get received byte of USART1 from ringbuffer (only available on selected ATmega). |
| **#8** | `void uart1_puts(const char *s);` |
| **Info** | Put string to ringbuffer for transmitting via UART. (only available on selected ATmega). |
| **#9** | `void uart1_putc(unsigned char data);` |
| **Info** | Put byte to ringbuffer for transmitting via UART. (only available on selected ATmega). |
| **#10** | `void uart1_puts_p(const char *s);` |
| **Info** | Put string from program memory to ringbuffer for transmitting via UART. (only available on selected ATmega). |

For Details visit: http://homepage.hispeed.ch/peterfleury/group__pfleury__uart.html

# Chapter 11 - SPI: Serial Peripheral Interface

## 11.1 Introduction

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.

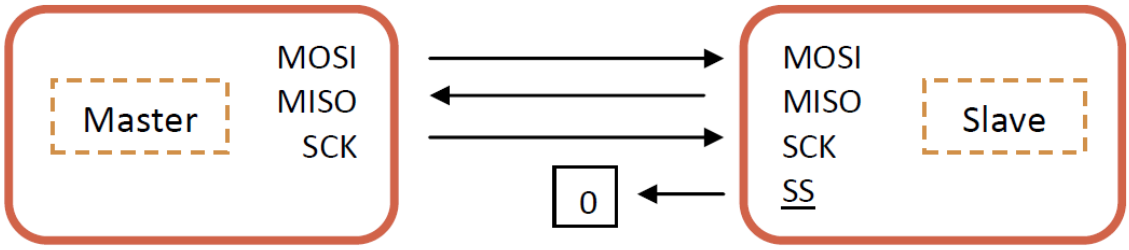
## 11.2 Theory of Operation

This communication protocol consists of folliwng lines or pins,

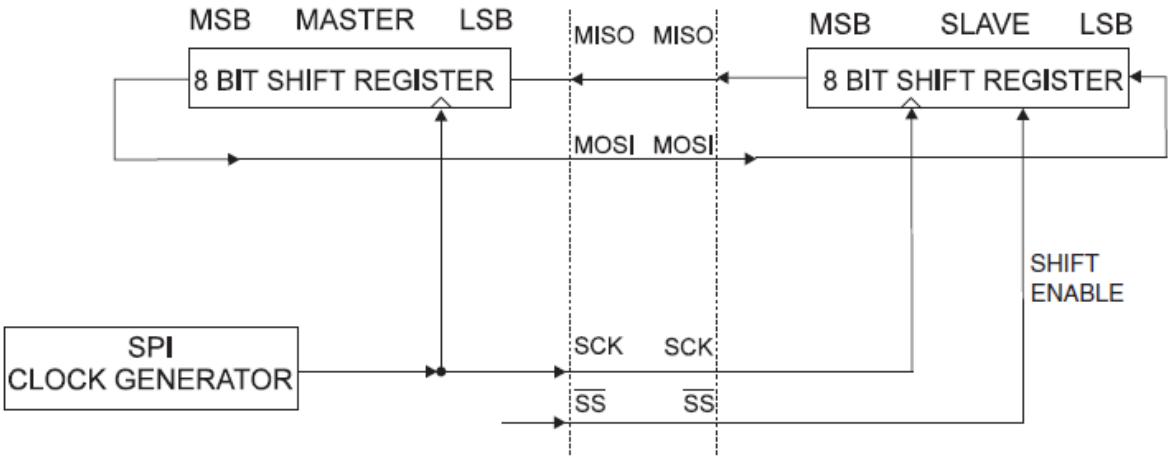
1. **MOSI** : **M**aster **O**ut **S**lave **I**n (Tx for Master and Rx for Slave)
2. **MISO** : **M**aster **I**n **S**lave **O**ut (Rx for Master Tx for Slave)
3. **SCK** : **S**erial **C**loc**k** (Clock line)
4. **SS** : **S**lave **S**elect (To select Slave chip) (if given 0 device acts as slave)

**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. **SS** for Master will be disconnected.

**Slave:** This device accepts the clock from master device. **SS** for this has to be made 0 externally.




Pin connections for SPI protocol




**SPI Data Communication**

# Chapter 12 - I2C Communication

## 12.1 Introduction

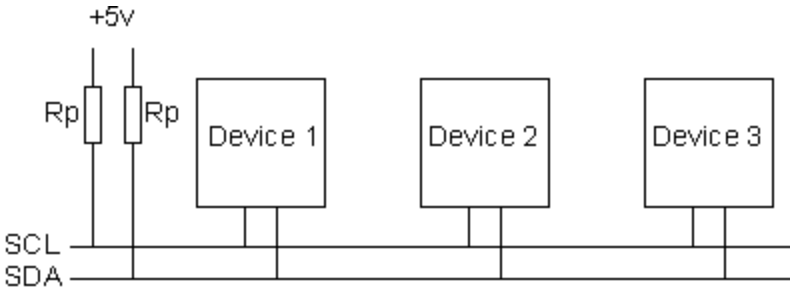
The I2C (Inter Integrated Circuit) Protocol is very popular for interfacing ICs with microcontroller designed by Phillips. It uses only 2 bi-directional lines for communication with microcontroller. I2C is a synchronous data transfer protocol & uses master/slave technique, where master (usually the microcontroller) initiates the communication, while the slave (any I2C device) works according to master. Multiple devices can connect at the same time, each having a unique 7-bit address. It can allow up-to 128 devices on a single bus.

## 12.2 Theory of Operation

Two bi-directional lines used for communication are:

1. **SDA (serial data):** Data transfer is done via this line
2. **SCL (serial clock):** Used for the Synchronous Clock

SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. There may also be a 5volt wire is power is being distributed to the devices. Both SCL and SDA lines are "open drain" drivers. What this means is that the chip can drive its output low, but it cannot drive it high. For the line to be able to go high you must provide pull-up resistors to the 5v supply. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You only need one set of pull-up resistors for the whole I2C bus, not for each device, as illustrated below:



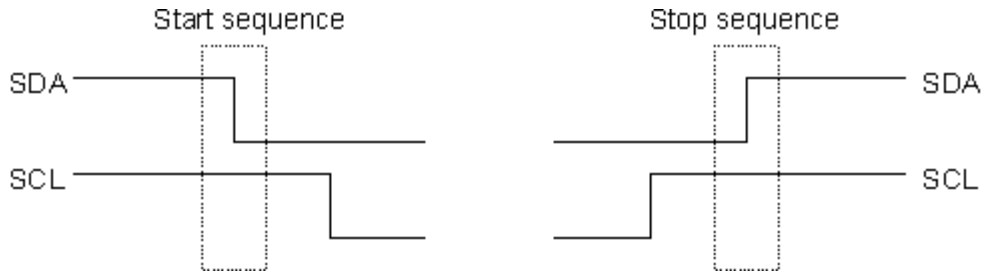

### 12.2.1 Masters and Slaves

The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. It is possible to have multiple masters, but it is unusual and not covered here. On your board, the master will be your controller and the slaves will be our modules such as the TMP-275 or APDS-9300. Slaves will never initiate a transfer. Both master and slave can transfer data over the I2C bus, but that transfer is always controlled by the master.

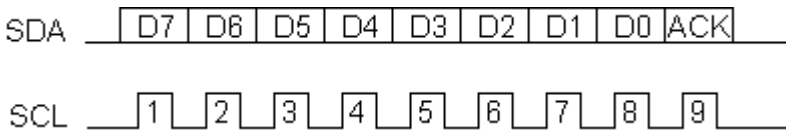
### 12.2.2 The I2C Physical Protocol

When the master (your controller) wishes to talk to a slave (our TMP-275 for example) it begins by issuing a start sequence on the I2C bus. A start sequence is one of two special sequences defined for

the I2C bus, the other being the stop sequence. The start sequence and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change whilst SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device.




Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer each 8 bit byte of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back a high then it is indicating it cannot accept any further data and the master should terminate the transfer by sending a stop sequence.

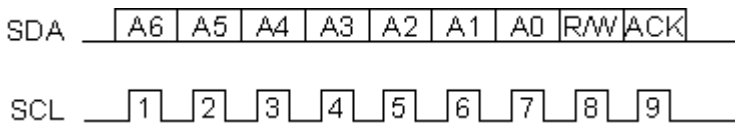



**How fast?**

The standard clock (SCL) speed for I2C up to 100KHz. Philips do define faster speeds: Fast mode, which is up to 400KHz and High Speed mode which is up to 3.4MHz. All of our modules are designed to work at up to 100KHz. We have tested our modules up to 1MHz but this needs a small delay of a few uS between each byte transferred. In practical robots, we have never had any need to use high SCL speeds. Keep SCL at or below 100KHz and then forget about it.

### 12.2.3 I2C Device Addressing

All I2C addresses are of 7 bits. This means that you can have up to 128 devices on the I2C bus, since a 7-bit number can be from 0 to 127. When sending out the 7 bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is **writing** to it or **reading** from it. If the bit is zero are master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7 bit address is placed in the upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).



The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses as write only, and the odd addresses as the read address for the same device. To take our TMP-275 for example, this is at address 0x9E. You would uses 0x9E to write to the TMP-275 and 0x9F to read from it. So the read/write bit just makes it an odd/even address.

### 12.2.4 The I2C Software Protocol

The first thing that will happen is that the master will send out a start sequence. This will alert all the slave devices on the bus that a transaction is starting and they should listen in in case it is for them. Next the master will send out the device address. The slave that matches this address will continue with the transaction, any others will ignore the rest of this transaction and wait for the next. Having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has. Some very simple devices do not have any, but most do, including all of our modules. Having sent the I2C address and the internal register address the master can now send the data byte (or bytes, it doesn't have to be just one). The master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction. So to write to a slave device:

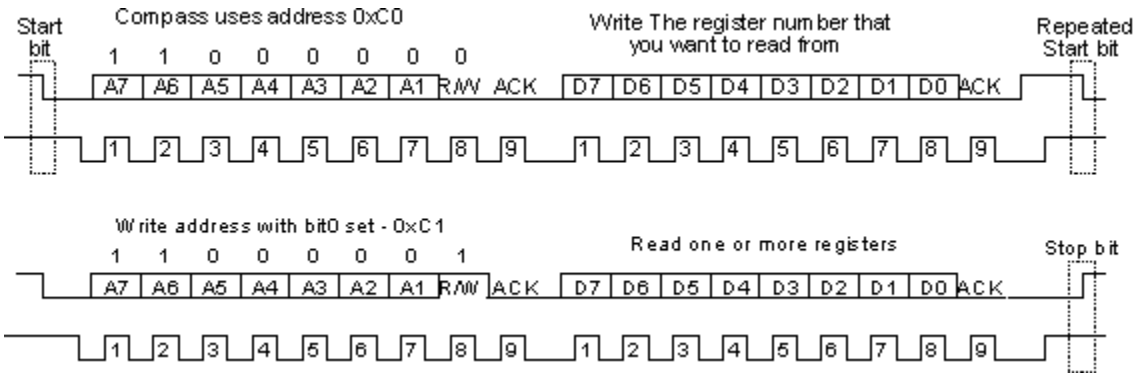
1. Send a start sequence
2. Send the I2C address of the slave with the R/W bit low (even address)
3. Send the internal register number you want to write to
4. Send the data byte
5. [Optionally, send any further data bytes]
6. Send the stop sequence.

### 12.2.5 Reading from the Slave

Before reading data from the slave device, you must tell it which of its internal addresses you want to read. For this you need to configure few registers. So a read of the slave actually starts off by writing to it. This is the same as when you want to write to it. You send the start sequence, the I2C address of the slave with the R/W bit low (even address) and the internal register number you want to write to. Now you send another start sequence (sometimes called a restart) and the I2C address again - this time with the read bit set. You then read as many data bytes as you wish and terminate the transaction with a stop sequence. So to read the both temperature bytes from the TMP-275 module:

1. Send a start sequence
2. Send I2C address with the R/W bit low (even address) to Write
3. Send Internal Register address of the Device, on which you want to write to.
4. Send a start sequence again (repeated start)
5. Send I2C address with the R/W bit high (odd address) to Read
6. Read data byte from Device
7. Send the stop sequence.

The bit sequence will look like this:




## 12.3 Data Communication Functions

We will be using **i2cmaster** driver, which is Open Source and is fully compatible with At-Mega Microcontrollers. It is purely written in assembly.

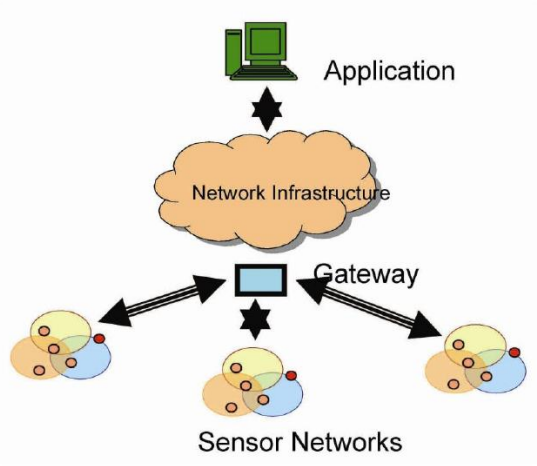
| #1 | `void i2c_init(void);` |
|---|---|
| **Info** | Initialize the I2C master interface. Need to be called only once. |
| **#2** | `void i2c_stop(void);` |
| **Info** | Terminates the data transfer and releases the I2C bus. |
| **#3** | `unsigned char i2c_start(unsigned char addr);` |
| **Info** | Issues a start condition and sends address and transfer direction.<br>**For Read Operation**, we send **device_address** + 1.<br>**For Write Operation,** we send **device_address** + 0.<br>For example, if device address is 0x6A, then;<br>**I2c_start(0x6A + 1)** will initiate connection with read request. |
| **#4** | `unsigned char i2c_rep_start(unsigned char addr);` |
| **Info** | Issues a repeated start condition and sends address and transfer direction. |
| **#5** | `void i2c_start_wait(unsigned char addr);` |
| **Info** | Issues a start condition and sends address and transfer direction. |
| **#6** | `unsigned char i2c_write(unsigned char data);` |
| **Info** | Send one byte to I2C device. |
| **#7** | `unsigned char i2c_readAck(void);` |
| **Info** | Read one byte from the I2C device, request more data from device. |
| **#8** | `unsigned char i2c_readNak(void);` |
| **Info** | Read one byte from the I2C device, read is followed by a stop condition. |
| **#9** | `unsigned char i2c_read(unsigned char ack);` |
| **Info** | Read one byte from the I2C device. |

For Details visit: http://homepage.hispeed.ch/peterfleury/group__pfleury__ic2master.html

# Chapter 13 - Wireless Sensor Network (WSN)

A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants and to cooperatively pass their data through the network to a main location.

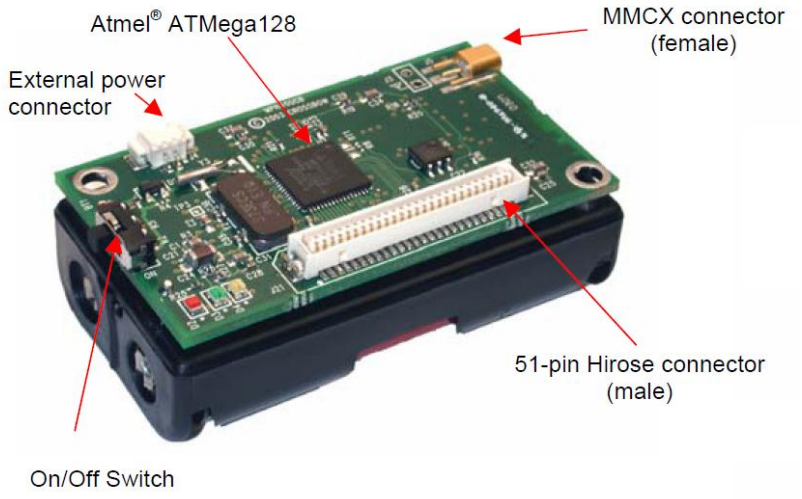
Today such networks are used in many industrial and consumer application, such as industrial process monitoring and control, machine health monitoring, environment and habitat monitoring, healthcare applications, home automation, and traffic control.




The WSN is built of "nodes" – from a few to several hundreds or even thousands, where each node is connected to sensors.

Each such sensor network node has typically following parts:

- Radio transceiver with an internal antenna or connection to an external antenna
- Low Power Microcontroller
- Electronic circuit for interfacing with the sensors
- Energy source, usually a battery or an embedded form of energy harvesting.



Nodes are of following types,

**Sensor Node:** Sensors like temperature, humidity, ambient light, motion, pressure etc.

**Actuator Node:** Actuators like LEDs, AC/Fan, Bulbs, Motors, HVAC, Alarm etc.

**Gateway Node:** Connects wireless sensor network to other world by converting data from wireless to other protocol like USB, Ethernet, WiFi etc.

## 13.1 Usage of sensor networks

Sensor networks have been useful in a variety of domains. The primary domains at which sensor are deployed follow:

**Environmental observation.** Sensor networks can be used to monitor environmental changes. An example could be water pollution detection in a lake that is located near a factory that uses chemical substances. Sensor nodes could be randomly deployed in unknown and hostile areas and relay the exact origin of a pollutant to a centralized authority to take appropriate measures to limit the spreading of pollution. Other examples include forest fire detection, air pollution and rainfall observation in agriculture.

**Military monitoring.** Military uses sensor networks for battlefield surveillance; sensors could monitor vehicular traffic, track the position of the enemy or even safeguard the equipment of the side deploying sensors.

**Building monitoring.** Sensors can also be used in large buildings or factories monitoring climate changes. Thermostats and temperature sensor nodes are deployed all over the building's area. In addition, sensors could be used to monitor vibration that could damage the structure of a building.

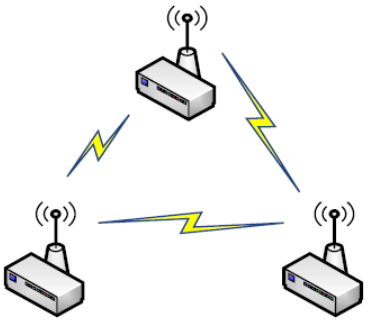
**Healthcare.** Sensors can be used in biomedical applications to improve the quality of the provided care. Sensors are implanted in the human body to monitor medical problems like cancer and help patients maintain their health.

## 13.2 Characteristics of WSN

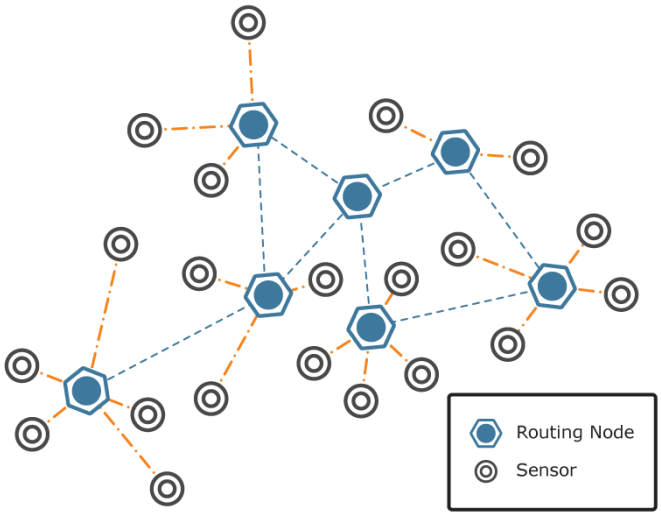
- Power consumption constrains for nodes using batteries or energy harvesting
- Ability to cope with node failures
- Mobility of nodes
- Dynamic network topology
- Communication failures
- Heterogeneity of nodes
- Scalability to large scale of deployment
- Ability to withstand harsh environmental conditions
- Ease of use
- Unattended operation.

## 13.3 Common Sensor Network Topologies

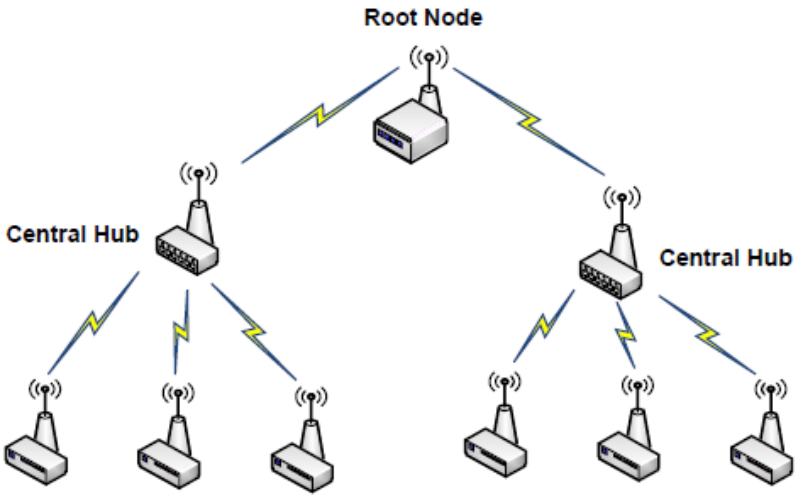
**Point-to-Point** networks allow each node to communicate directly with another node without needing to go through a centralized communications hub. Each Peer device is able to function as both a “client” and a “server” to the other nodes on the network.



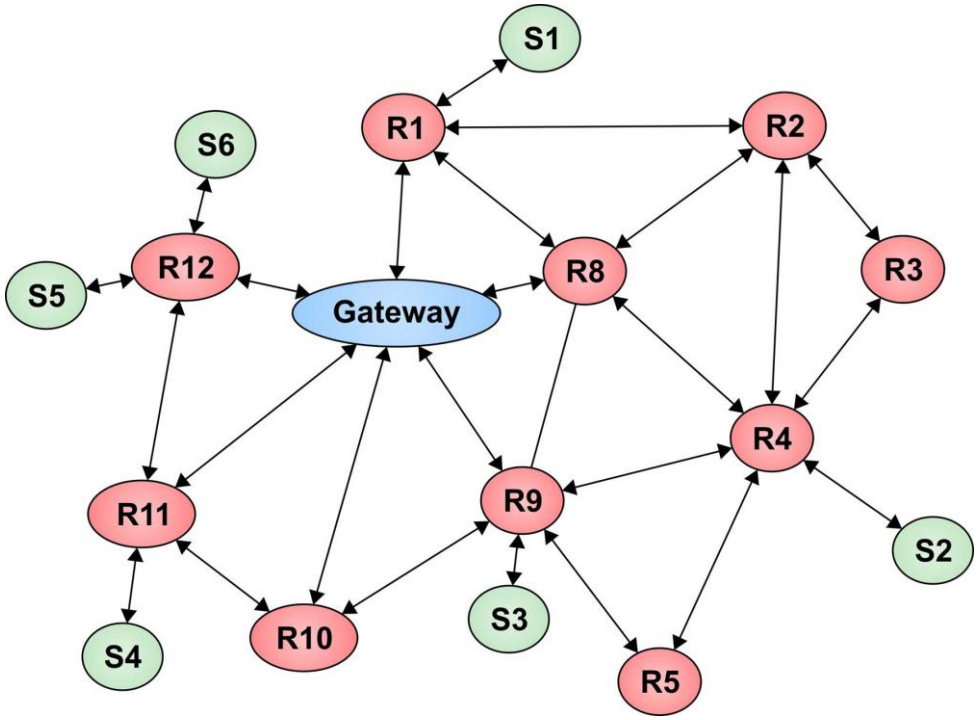
**Star networks** are connected to a centralized communications hub. Each node cannot communicate directly with one another; all communications must be routed through the centralized hub. Each node is then a “client” while the central hub is the “server”.



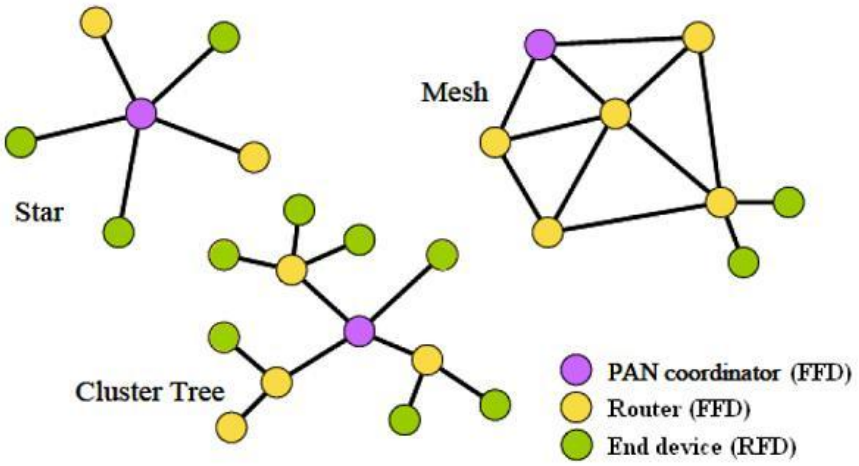

**Tree networks** use a central hub called a Root node as the main communications router. One level down from the Root node in the hierarchy is a Central hub. This lower level then forms a Star network. The Tree network can be considered a hybrid of both the Star and Peer to Peer networking topologies.




**Mesh networks** allow data to "hop" from node to node, this allows the network to be self-healing. Each node is then able to communicate with each other as data is routed from node to node until it reaches the desired location. This type of network is one of the most complex and can cost a significant amount of money to deploy properly.



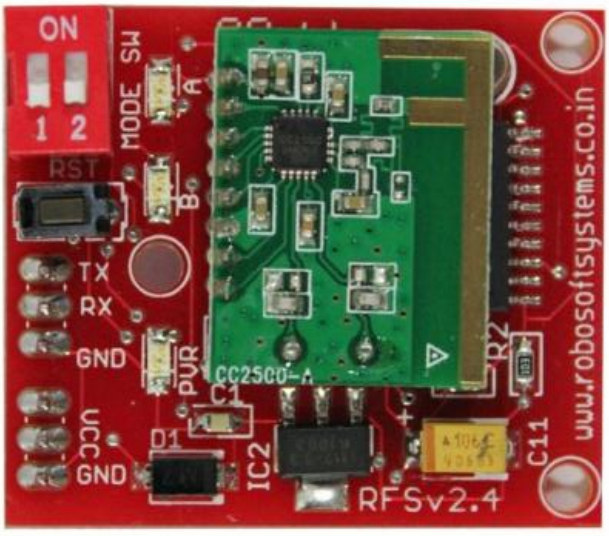
**Comparision of Star, Tree and Mesh.**




## 13.4 CC-2500 Wireless Module

This High Speed CC2500 Based Wireless module is a plug and play replacement for the wired Serial Port (UART) supporting baud rates upto 38400.

1) Baud Rate: 38400. Serial UART Mode
2) Packet Length: Variable (0 to 40) or Fixed (1 packet).
3) 60+ meters range Line of Sight / 30 meters range indoors
4) Multiple channel selection enabling upto 255 different pairs to work in the same area
5) Modes of operation: Config mode and Run mode.
6) On-board jumper Setting for Config/Run Mode and Packet/Byte Mode.
7) Direct Replacement for wired Serial Cable for and serial communication.



# Chapter 14 - Network Communication

## 14.1 OSI Model

### 14.1.1 Introduction

Historically, various companies used their own approaches for network communication specific to their technology. Like, IBM had their own standards which worked on IBM machines only. This restricted communication between devices over the network drastically.

OSI (Open Systems Interconnect) Model is a standardized approach which defines communication between two or more devices communicating over some network. OSI provides a uniform way to access networked resources.

Various Advantages offered by OSI are:

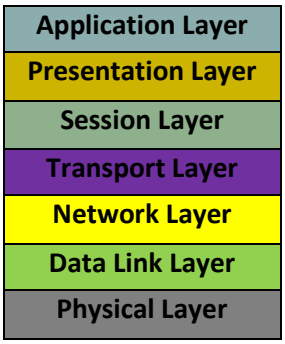
1. Device/Operating System Independent Communication
2. Universally Accepted Method
3. Reliable Data Communication

### 14.1.2 Layers in OSI Model

OSI follows Layered architecture. A layered architecture has number of logical layers where each layer performs some specific operation. It divides the task of data transmission into subtasks where each subtask is performed by a layer. It makes it easier for programmers to transmit data between applications.

OSI model is a logical model consisting of 7 layers. At sender's side, each layer performs its operation on the data, adds information using headers and sends it to the layer below it. At receiver's side, the same operations occur in reverse order. The layers get striped at receiver's side.

Below is a simplified and minified introduction to various layers of OSI Model.




1. **Application Layer:** This is the layer where all of user applications are running. This layer comprises applications like Internet Explorer, Yahoo! Messenger etc. which have data to send/receive.
   Various Protocols which develop this Layer are HTTP, SMTP, Telnet, DNS etc.

2. **Presentation Layer:** This layer performs operations like Data Encryption, Compression. This presents data in formatted way and hence its name.

3. **Session Layer:** This layer maintains communication between two network applications and is also responsible for Authentication and authorization. This layer is responsible for maintaining sessions of various applications.

4. **Transport Layer:** Transport Layer receives the data to be sent from Session Layer and creates segments of variable length. The original data/message is also called payload data. Transport layer. It multiplexes data of various applications, creates virtual end-to-end connections, and provides transparent way to transmit data in a network.
   Protocols which develop this layer may be TCP (Transmission Control Protocol) or UDP (Uniform Datagram Protocol).

5. **Network Layer:** It is responsible for packetizing data. Network layer assigns Source and Destination IP addresses to the packet. The IP address is a unique number which identifies a network device. Destination IP Address specifies the device which will receive the packet.
   IP (Internet Protocol) implements this layer. Routing of packets is done at this level using IP Addresses.

6. **Data Link Layer:** This layer divides the data packet received from Network Layer into fixed sized blocks of data, called Frames. The size of frame is defined using MTU (Max Transmission Unit) which is the maximum size of packet that can be transferred. The Data Link Layer is subdivided into 2 sub-layers:
   a. **MAC – Media Access Control =>** Frames are given a MAC (Media Access Control) address, which is a 48-bit number and is unique on the Local network. The task of assigning MAC address is done by this layer.
   b. **LLC - Logical Link Control =>** This layer is responsible for Error Checking and flow control.

7. **Physical Layer:** This layer denotes the physical medium through which data is sent. Physical layer defines the cable or physical medium itself, e.g., thinnet, thicknet, unshielded twisted pairs (UTP). The data may be sent in the form of electrical signals or in the form of light impulses.

While Programming, we can shorten the OSI model to TCP/IP model as:

| Application Layer |
|---|
| Transport Layer |
| Internet Layer |
| Network Interface Layer |

Here Application Layer consists of Application Layer + Presentation Layer + Session Layer.

Also, Network Interface layer consists of Data Link Layer and Physical layer. This is the layer where Network protocols like Ethernet, Token Ring etc. come into the play. As Ethernet is the most popular topology used now-a-days, we will be studying about Ethernet only.

## 14.2 Ethernet

### 14.2.1 Introduction

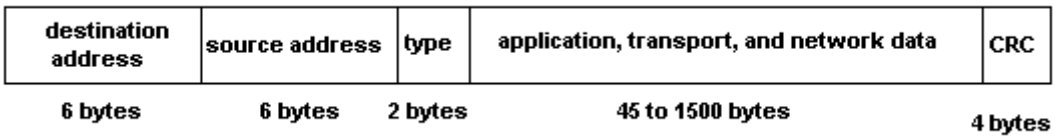
Ethernet is a standard protocol used for data communication over the Local networks. IEEE 802.3 standard defines Ethernet at the physical and data link layers of the OSI network model. It uses CSMA carrier signalling to transmit signals at lower level.

Various functions performed by Ethernet are:

1. Packs the packets of data into fixed length frames.
2. Assigns a locally Unique MAC (Media Access Control) Address.

### 14.2.2 Ethernet Frame

Ethernet packs data in the form of Frames. The structure of a Frame is shown as:




1. **Destination Address=>** MAC address of receiver side.
2. **Source Address=>** MAC address of sender side.
3. **Type=>** Type can have any of these values:
    a. 0800 IP Datagram
    b. 0806 ARP request/reply
    c. 8035 RARP request/reply
4. **Data=>** Payload Data including all the Headers from layers above it.
5. **CRC=>** Cyclic Redundancy Check, attached as a Trailer, while all the other information is attached in the form of Header Fields.

There is a maximum size of each data packet for the Ethernet protocol. This size is called the maximum transmission unit **(MTU)**. What this means is that sometimes packets may be broken up as they are passed through networks with MTUs of various sizes.

### 14.2.3 MAC Addressing

A MAC address is a 48-bit address. It is generally represented in Hex notation. An example of a MAC address is **00-17-C4-A3-76-69**.

MAC address is also called Physical Address or Layer 2 Address. It is used to identify a machine uniquely in a Local Network.

More information about the MAC addresses can be obtained from Wikipedia/Google.

| ? Did you Know |
| --- |
| You can get MAC address of your PC with the command "**getmac**" on Windows and "**ifconfig**" on Linux PC. |

### 14.2.4 Ethernet using ENC28J60

We will be using ENC28J60 IC for the Ethernet Support on our board. The ENC28J60 is a small chip with 28 pins only and has a SPI interface which is easy to use from any microcontroller. We can also call it Ethernet to SPI converter. A simple block diagram illustrating the ENC28J60 IC can be shown as:

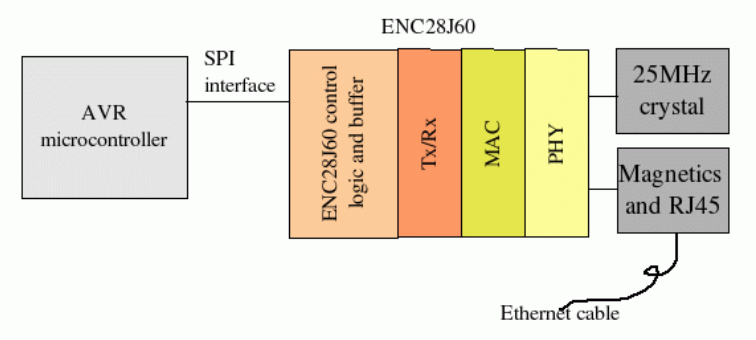



**Figure 14-1 Block Diagram of ENC28J60 Communication - Taken From tuxgraphics.org**

## 14.3 TCP/IP Stack

### 14.3.1 Introduction

TCP/IP stack plays an important role in network functioning. The TCP/IP Stack is special software which implements the Transport and Network Layer of OSI or TCP/IP Model. While implementing the TCP/IP stack, we follow TCP/IP Model for convenience. The applications/protocols in Application Layer directly depend on Transport Layer protocols. The following table shows various networking protocols and their position in TCP/IP Model.

| | | | | | |
|---|---|---|---|---|---|
| **Application Layer** | **HTTP FTP Telnet Finger SSH DNS POP3/IMAP SMTP Gopher BGP Time/NTP Whois TACACS+ SSL** | **DNS SNMP RIP RADIUS Traceroute tftp** | **Ping** | | |
| **Transport Layer** | **TCP** | **UDP** | **ICMP** | **OSPF** | |
| **Internet Layer** | **IP** | | | | **ARP** |
| **Network Interface Layer** | **Ethernet/802.3 Token Ring (802.5) SNAP/802.2 X.25 FDDI ISDN Frame Relay SMDS ATM Wireless (WAP, CDPD, 802.11) Fibre Channel DDS/DS0/T-carrier/E-carrier SONET/SDH DWDM PPP HDLC SLIP/CSLIP xDSL Cable Modem (DOCSIS)** | | | | |

As you can easily guess, it is not possible to include all of the protocols available in this table. This table contains a fairly limited set of the protocols. The dependency of each application layer protocol is illustrated. As an example, HTTP protocol depends on TCP protocol, RADIUS protocol depends on UDP protocol. Protocols like DNS in application layer can work with both TCP and UDP. Ping command needs a special Transport Layer protocol called ICMP to work.

The Network Interface Layer is usually implemented in our Ethernet Hardware itself. With the TCP/IP stack in hand, we have Transport and Network Layer implementation. The implementation of

various Application Layer protocols like HTTP, Telnet, Ping is discussed in later sections. Hence, we have a complete implementation of all the layers. Once we have all the layers implemented, we have a complete network application.

A comprehensive list of features provided by TCP/IP Stack is:

- Forms the **Transport Layer and Network Layer** of OSI Model.
- Provides connectionless (UDP)/connection oriented (TCP) data transfer modes.
- Performs Error Checking & Correction wherever necessary (TCP only). Hence data is delivered efficiently and reliably.
- Breaks big chunks of data into small packets.
- Provides a unique location (IP Address) to each networked device at Network Layer.
- Provides a unique channel (Port number) to each application, which helps in separating data of each application from any other application.
- Managing Flow control of data using Window Protocols at Transport Layer.

We will now briefly discuss about TCP and IP protocols.

### 14.3.2 TCP

TCP or Transmission Control Protocol is a protocol which is used for controlling data transmission between two devices. It is a reliable, connection oriented protocol. Connection oriented basically means that we first need to setup a virtual connection between the communicating parties before they can actually send/receive the data. This process is known as Handshake. After the connection is established, data transmission can begin. The connection must be closed once we are finished with the data to be transmitted.

Prime features of TCP:

- **End to end reliability=>** The data transfer in TCP happens to be reliable due to built-in support for Error checking. Also, a virtual connection is established to ensure more efficient and reliable delivery of data.
- **Data packet Re-Sequencing=>** The data packets are always sequenced and it ensures the data to be delivered correctly in case of multi-packet data. Also, arrival of packet 5 before 3, won't result in data loss.
- **Flow control=>** Flow control is a mechanism used for controlling the normal flow of data so that the communicating parties remain unaffected when transfer rates are unequal. This will prevent condition of Stack Buffer Overflow/Underflow.

#### *14.3.2.1 Three Way Handshake*

As the name suggests, TCP connection initiation is done in 3 steps. Various steps needed for the handshake are:

1. **SYN=>** The active open is performed by the client sending a SYN to the server. It sets the segment's sequence number to a random value A.

2. **SYN-ACK=>** In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number (A + 1), and the sequence number that the server chooses for the packet is another random number, B.
3. **ACK=>** Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value i.e. A + 1, and the acknowledgement number is set to one more than the received sequence number i.e. B + 1.

The three-way handshake is described in detail at: http://support.microsoft.com/kb/172983.

### *14.3.2.2 TCP Header*

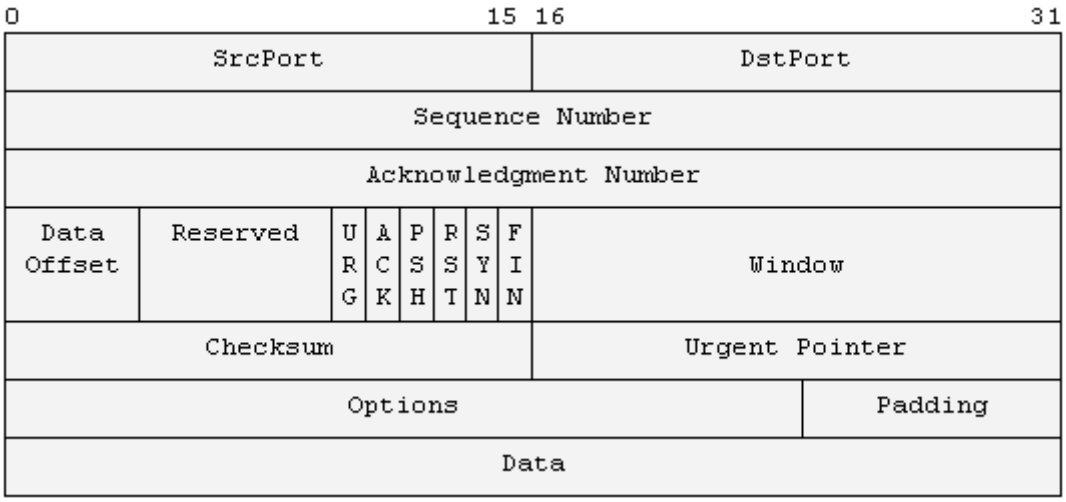



**Figure 14-2 TCP header**

The fields of TCP Header are explained below:

4. **SrcPort =>** The source port is port number of requesting client. It can be any random value for a new connection. But the value must remain the same till the connection is open.
5. **DstPort =>** The destination port is port at which server is running.
6. **Sequence No=>** Data at application layer is broken into multiple segments at Transport Layer, which are ordered according to sequence number.
7. **Acknowledgement No=>** Every packet that is sent and a valid part of a connection is acknowledged with an empty TCP segment with the ACK flag set.
8. **Reserved=>** This is unused and contains binary zeroes.
9. **Data Offset=>** The segment offset specifies the length of the TCP header in 32bit/4byte blocks. Without tcp header options, the value is 5.
10. **TCP Flags=>** This field consists of six binary flags as:
    a. **Urgent Pointer (URG) =>** Segment will be routed faster, used for termination of a connection.

b. **Acknowledgement (ACK) =>** Used to acknowledge data and in the second and third stage of a TCP connection initiation.
c. **Push (PSH) =>** The IP stack will not buffer the segment and forward it to the application immediately.
d. **Reset (RST) =>** Tells the peer that the connection has been terminated.
e. **Synchronization (SYN) =>** A segment with the SYN flag set indicates that client wants to initiate a new connection to the destination port.
f. **Final (FIN) =>** The connection should be closed, the peer is supposed to answer with one last segment with the FIN flag set as well.

11. **Window=>** The amount of bytes that can be sent before the data should be acknowledged with an ACK before sending more segments.
12. **Checksum=>** The checksum of pseudo header, tcp header and payload. The pseudo is a structure containing IP source and destination address, 1 byte set to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes (unsigned short) containing the total length of the TCP segment.
13. **Urgent pointer=>** Only used if the urgent flag is set, else zero. It points to the end of the payload data that should be sent with priority.

Transport layer can also have protocol like UDP, which is a connectionless data transmission protocol. The discussion of UDP Protocol is out of the scope for this course.

### 14.3.3 IP – Internet Protocol

This protocol works at Network Layer. It manages device addressing. Each device is given a unique logical address called IP address (discussed later in detail). Various other features provided by IP:

1. Creation of packets
2. Routing of packets
3. Providing unique address to each and every device

#### *14.3.3.1 IP Header*

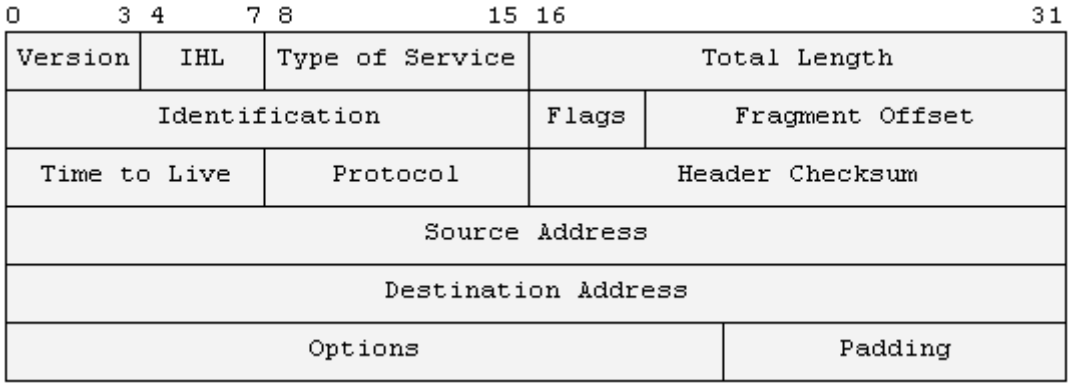



**Figure 14-3 IP Header**

4. **IP Version=>** It may be 4 or 6 depending on IP version. We will be using IPv4 only, so the value of this field will always be 4.
5. **IP Header Length (IHL) =>** IP header length in 32bit octets. This means a value of 5 for this field means 20 bytes (5 * 4).

6. **Type of Service=>** type of service controls the priority of the packet. 0x00 is normal. The first 3 bits stand for routing priority, the next 4 bits for the type of service (delay, throughput, reliability and cost).
7. **Total Length=>** Total length must contain the total length of the IP datagram. This includes IP header and ICMP/UDP/TCP header and payload size in bytes.
8. **Identification =>** the id sequence number is mainly used for reassembly of fragmented IP datagrams. When sending single datagrams, each can have an arbitrary ID.
9. **Fragment Offset, Flags=>** The fragment offset is used for reassembly of fragmented datagrams. The first 3 bits are the fragment flags, the first one always 0, the second the do-not-fragment bit (set by ip_off |= 0x4000) and the third the more-flag or more-fragments-following bit (ip_off |= 0x2000). The following 13 bits is the fragment offset, containing the number of 8-byte big packets already sent.
10. **Time to Live=>** It is the amount of hops (routers to pass) before the packet is discarded, and an icmp error message is returned. The maximum value of this field can be 255.
11. **Protocol=>** Transport layer protocol. can be tcp (6), udp(17), icmp(1), or whatever protocol follows the IP header.
12. **Checksum=>** The datagram checksum for the whole ip datagram. every time anything in the datagram changes, it needs to be recalculated, or the packet will be discarded by the next router.
13. **Source IP Address=>** Source IP address, converted to long int format.
14. **Destination IP Address=>** Destination IP address, converted to long int format.

We can manually create and fill these headers to create and send a packet.

| ? Did you Know |
| --- |
| We can use a Packet Analyser like "**Wireshark**" to analyse the data at each layer of OSI model. Wireshark gives us detailed view of all the headers of the packet. |

### 14.3.4 IP Address

An IP address is a unique address given to any networked device. The IP addresses come in 2 flavors. IPv4 and IPv6. We will be only discussing the IPv4 addresses. An IPv4 address is a 32-bit unique number. It is subdivided into 4 octets of 8 bits each, separated by Dots. An example of an IP address is 10.0.0.5.

| ? Did you Know |
| --- |
| You can get the IP address assigned to your PC using "**ipconfig**" on Windows, "**ifconfig**" on Linux. |

#### *14.3.4.1 Valid IP Addresses*

Valid IP Addresses begin from 1.0.0.1 – 223.255.255.254. IPs ranging from 224.0.0.0 to 255.255.255.255 are reserved and cannot be used for general purposes. Any IP address cannot have a zero in the last octet, like x.x.x.0. Also, the address ending in 255, like x.x.x.255 is used as a broadcast address for the given network. When destination address is the broadcast address, then everyone on the network receives the packet.

**Valid IP Address Examples:** 192.168.1.1, 10.24.32.5, 117.56.3.1

**Invalid IP Addresses Examples:** 0.1.5.10, 10.14.240.0

Even in the range of valid IP addresses, we have few ranges reserved for some specific purposes. Like, 127.x.x.x is reserved for loopback interface.

There are 3 IP network addresses reserved for private networks.

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.168.0.0 to 192.168.255.255

There are two parts of an IP address:

- **Network ID=>** Identifies a network uniquely.
- **Host ID=>** Identifies a unique host/device within a network.

| **IP Address** | 192 | 168 | 1 | 1 |
|---|---|---|---|---|
| **Binary Form** | **11000000** | **10101000** | **00000001** | **00000001** |
| **Subnet Mask** | 255 | 255 | 255 | 0 |
| **Binary Form** | **11111111** | **11111111** | **11111111** | **00000000** |
| **Address Parts** | **Network ID(192.168.1.0)** | | | **Host ID(0.0.0.1 or 1)** |
| Binary representation of an IP Address illustrating the division of IP address into Network and host ID | | | | |

As we can see, the part of IP address which is covered by 1's in Subnet Mask is called Network ID. The part of IP Address, which is covered by 0's of Subnet mask is Host ID. Based on the subnet mask, we can categorize IP addresses into various classes.

### *14.3.4.2 IP Address Classes*

The various classes of networks specify additional or fewer octets to designate the network ID versus the host ID.

| **Class** | **1st Octet** | **2nd Octet** | **3rd Octet** | **4th Octet** |
|---|---|---|---|---|
| A | Net ID | Host ID | | |
| B | Net ID | | Host ID | |
| C | Net ID | | | Host ID |

The addressing scheme for class A through E networks is shown below:

| **Network Type** | **Address Range** | **Subnet Mask** | **Comments** |
|---|---|---|---|
| Class A | 001.x.x.x to 126.x.x.x | 255.0.0.0 | For very large networks |
| Class B | 128.1.x.x to 191.254.x.x | 255.255.0.0 | For medium size networks |
| Class C | 192.0.1.x to 223.255.254.x | 255.255.255.0 | For small networks |
| Class D | 224.x.x.x to 239.255.255.255 | | Used to support multicasting |
| Class E | 240.x.x.x to 247.255.255.255 | | Kept for Research purposes |

## 14.4 IP Address in Plain Integer Format

The Standard Dot notation of IP addresses is just used for our understanding. Internally Dotted notation is treated as a string which must be converted to plain 32-bit integer before use. If the IP address we need to convert is 192.168.1.8, the conversion process can be illustrated as:

| **Given IP Address** | 192 | 168 | 1 | 8 |
|---|---|---|---|---|
| **Multiply with** | $256^3$ | $256^2$ | 256 | 1 |
| **Solve** | 16777216x198 | 65536x168 | 256x1 | 8 |
| **Add** | 3321888768 | 11010048 | 256 | 8 |

**Result = 3332899080**.

**? Did you Know**

If a firewall has blocked access to any website, we can get IP address of the website and convert it to integer format. Using the integer value to open website can actually befool the firewall. Example: "**http://3232235784**".

## 14.5 Raw Packet Creation

Raw Packet creation involves manually filling up the Header fields of TCP and IP. This process is automatically done by TCP/IP stack. In this section, we will briefly cover how we can manually create and send a packet. Various steps involved in packet creation are:

1. Taking an arbitrary message to be sent. This is the data to be sent by application layer.
2. Defining the TCP and IP headers as C structures.
3. Defining functions for checksum calculation, Converting IP address into 32-bit Integer format.
4. Filling up the values of header fields.
5. Sending the manually filled packet to Network Interface Layer.

**? Did you Know**

Manually creating the packets and then inserting them into the network, is called **"Packet Injection"**.

The network uses Big Endian notation, as compared to Little Endian notation used in Intel Processors. While creating the packets manually, we need to know this thing and explicitly change the byte-order. Endianness is discussed in Appendix Section.

## 14.6 Ping

### 14.6.1 Introduction

"In December of 1983 I encountered some odd behaviour of the IP network at BRL. Recalling Dr. Mills' comments, I quickly coded up the PING program, which revolved around opening an ICMP style SOCK_RAW AF_INET Berkeley-style socket(). The code compiled just fine, but it didn't work -- there was no kernel support for raw ICMP sockets! Incensed, I coded up the kernel support and had everything working well before sunrise. Not surprisingly, Chuck Kennedy (aka "Kermit") had found and fixed the network hardware before I was able to launch my very first "ping" packet. But I've used it a few times since then. If I'd known then that it would be my most famous accomplishment in life, I might have worked on it another day or two and added some more options."

- By Ping Author

Ping is a network command, which is used to detect if a Host is responding or not. Ping uses ICMP (Internet Control Message Protocol) Protocol, which works at Network Layer.

| ? Did you Know |
| --- |
| Ping command can be used to test if you are connected to internet. Like: "***ping*** ***www.google.com***". This command also resolves the google.com DNS to its IP address. |

### 14.6.2 ICMP Header

The format of ICMP Packet headers is shown as:

| type | code | checksum |
| --- | --- | --- |
| identifier | | sequence number |

1. **Type**=> Message type, for example 0 - echo reply, 8 - echo request, 3 - destination unreachable.
2. **Code =>** This is significant when sending an error message (unreachable), and specifies the kind of error.
3. **Checksum=>** The checksum for the ICMP header + data. It is same as the IP checksum.
4. **Identifier=>** used in echo request/reply messages, to identify the request.
5. **ICMP Sequence =>** identifies the sequence of echo messages, if more than one is sent.

We can easily fill up all the fields of ICMP packet header.

### 14.6.3 Example ICMP Request

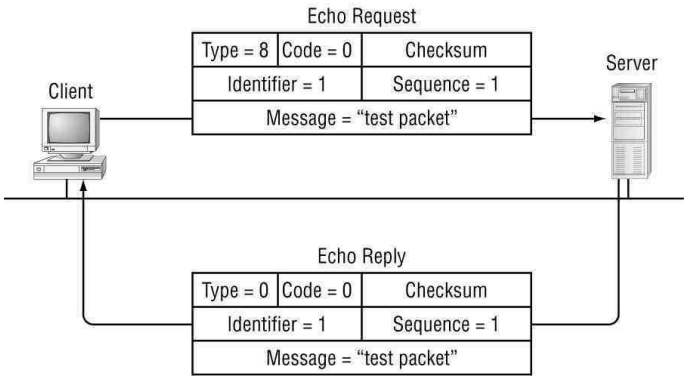



**Figure 14-4 ICMP Request Example - Image taken from codeidol.com**

From the illustration, we can see that in addition to normal fields of ICMP headers, we can also add some text/message in the packet. Here, **Type=8 is Echo Request** which is sent from the client side. The server then replies the request with **Type=0, Echo Reply**. The detailed list of messages supported by the ICMP protocol is available on the web.

Ping is a very useful utility, which can provide us some other diagnostic information also. It also shows time server took to reply. You can try experimenting with the Ping command, exploring the features it provides.

A good article for ICMP/Ping is available at: http://support.microsoft.com/kb/170292.

## 14.7 HTTP: Hyper Text Transfer Protocol

### 14.7.1 Introduction

Ever wondered, how does Firefox take you to Google.com? In this section, we will be studying how we make requests to Web Servers and how we get response to our requests.

HTTP is the underlying Application Layer protocol used to communicate on the Web. We can send various requests using it and in return, we get some answer from Web Server.

### 14.7.2 Web Client/Browser

Browser is the application we use to surf the internet. When we open **www.google.com**, a request is sent to Web Server, which is shown as:

**GET / HTTP/1.1**
**User-Agent : Firefox**
**Host : www.google.com**

In this case, User-Agent **Firefox** is requesting the **default page (/)** of Host **www.google.com**. The request is made using **HTTP Protocol version 1.1**. Here User-Agent corresponds to the Brower used.

The **GET** method used above is one of various methods available in HTTP, some of which are:

1. GET – Used to request some information from a Server
2. HEAD – Similar to GET, but returns only Headers without any data
3. POST – Used to submit form data on the server
4. PUT – Used to upload a file on server
5. DELETE – Deletes a file from the Server

The general form of the request can be shown as:

[HTTP_METHOD] [PATH_OF_FILE_ON_WEBSERVER] HTTP/1.1
User-Agent: Name of Web Client
Host: Address of Web Server

In addition to that, some other information can be attached in the form of Headers.

**? Did you Know**

You can use an HTTP sniffer to sniff HTTP traffic on your computer. Sniffers can also show you the actual values of HTTP Request & Response Header. An example of such software is **Effetech HTTP Sniffer**. Just Google the term "**HTTP sniffer**" to get more information about it.

### 14.7.3 Web Server

Web Server is the application which serves Browser(Web Client). For building a Web Server, we need to study how a Web Server responds according to various situations.

In above example, we sent a request to the Web Server. The response we got was:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

Following these Headers is our data/webpage we had requested.

#### *14.7.3.1 Web Server Status Codes*

In the first line of Response,

```
HTTP/1.1 200 OK
```

200 is the Status Code. The meaning of **200 is Success**. Status Codes determine whether we got success or some error.

Various Status Codes can be categorized as:

| Response Code | Meaning of the Message |
|---|---|
| 1xx | Informational Message |
| 2xx | Success |
| 3xx | Redirection is required |
| 4xx | Error At User End |
| 5xx | Error At Server End |

Most Important Codes that we need to implement in our Web Server are:

200 – **Success,** The requested data is also sent
301 – Resource we requested has been **Moved Permanently**
404 – Requested resource was **Not Found**
500 – Internal **Server Error**

**? Did you Know**

A very basic server serving only 1 or 2 files can be created only 5 lines of coding in perl. The perl language is provided by default on Linux machines while we can use ActivePerl on Windows platforms.

## 14.8 Telnet

A telnet is a simple Application Layer Protocol which allows us logging in to remote computers. A client is used to connect to the Server. Once our login session is started, we can send and receive data between client and server using a window called Terminal. Telnet runs on Port number 23.

### 14.8.1 Telnet Messages

Telnet uses raw TCP packets. It does not add any data in the form of headers. Hence it is a very simple form of communication. It provides a set of Messages/Commands. Messages can be transmitted back and forth between Server and client.

Various messages provided by telnet for communication are:

| Name | Code | Meaning | Comment |
|---|---|---|---|
| **SE** | 240 | End of subnegotiation parameters | |
| **NOP** | 241 | No operation | |
| **DM** | 242 | Data mark | Indicates the position of a Synch event within the data stream. This should always be accompanied by a TCP urgent notification. |
| **BRK** | 243 | Break | Indicates that the "break" or "attention" key was hi. |
| **IP** | 244 | Suspend | Interrupt or abort the process to which the NVT is connected. |
| **AO** | 245 | Abort output | Allows the current process to run to completion but does not send its output to the user. |
| **AYT** | 246 | Are you there | Send back to the NVT some visible evidence that the AYT was received. |
| **EC** | 247 | Erase character | The receiver should delete the last preceding undeleted character from the data stream. |
| **EL** | 248 | Erase line | Delete characters from the data stream back to but not including the previous CRLF. |
| **GA** | 249 | Go ahead | Under certain circumstances used to tell the other end that it can transmit. |
| **SB** | 250 | Subnegotiation | Subnegotiation of the indicated option follows. |
| **WILL** | 251 | Will | Indicates the desire to begin performing, or confirmation that you are now performing, the indicated option. |
| **WONT** | 252 | Wont | Indicates the refusal to perform, or continue performing, the indicated option. |
| **DO** | 253 | Do | Indicates the request that the other party perform, or confirmation that you are expecting the other party to perform, the indicated option. |
| **DON'T** | 254 | Don't | Indicates the demand that the other party stop performing, or confirmation that you are no longer expecting the other party to perform, the indicated option. |
| **IAC** | 255 | Interpret as command | Interpret as a command |

*Table referenced from http://support.microsoft.com/kb/231866.

### 14.8.2 Implementation

Telnet can be implemented very easily. The implementation is similar as we implemented HTTP Server. In case of telnet, we need to send these messages for communication instead of forming HTTP responses as earlier.

### 14.8.3 Summary

Telnet was widely used during initial times for router configurations and other various purposes. As it was realized that telnet does not provide any security mechanism and hence data can be traced very easily, the usage of telnet was considerably reduced. A new protocol called SSH(Secure SHell) is used now a days which uses Data Encryption to securely deliver information. It also provides many advanced feature not available in Telnet.

## 14.9 Terms and Definitions

1. **Network** – A network is defined as group of two or more devices communicating with each other.
2. **Host/Network Device** – A device which has ability to talk to other devices on a network. Say, Computer, Mobile, Router etc.
3. **NIC (Network Interface Card) –** A hardware which enables a device to communicate over the network.
4. **Protocol** – Set of Rules that govern some Operation. In simple words, Protocols describe the way communication occurs between sender and receiver at various levels.
5. **IP Address** – A unique number that identifies a network device. It is a 32-bit Address and is unique globally in case of internet.
6. **Port number** – A port number is a simple 16-bit integer value, which uniquely identifies an application.
7. **MAC Address** – A 48-bit value which uniquely identifies a network device on a Local Area Network.
8. **HTTP** – Hyper Text Transfer Protocol, is a protocol which defines method of communication between a web client and web server.
9. **PING** – Packet INternet Groper, is a network command, which is used to check if a host is alive or dead. This command works at Network Layer.
10. **ARP** – Address Resolution Protocol, is used for getting MAC address of a device if we know the IP address. In other words, it resolves IP address to MAC address.

# Chapter 15 - Tux-Graphics TCP/IP Stack

## 15.1 Introduction

As we have discussed briefly about the TCP/IP stacks. Now, it is the time to actually build up an Application using the stack. But, before building any application, we will be briefly discussing about the Stack we will be using.

TuxGraphics.org provides an Open Source stack which can be used freely. In this section, we will be briefly discussing the functions provided in the stack.

| ? Did you Know |
|---|
| Programming an application which can work over a network of PCs is called Socket Programming. The standard API for Socket Programming is BSD Socket API. The programming style of Socket programming is similar both in Windows and Linux. |

## 15.2 Stack Files

The stack consists of following files:

| Filename | Description |
|---|---|
| ip_arp_udp_tcp.(c\|h) | These files contain the TCP/IP stack functions, forming the Network, Transport Layer. |
| enc28j60.(c\|h) | These files implement the Data Link Layer. Also, it contains functions to communicate with ENC28J60 using SPI protocol. Basically, it is a driver to communicate with ENC28J60. |
| net.h | This contains some constant values of various header fields of TCP, IP, Ethernet headers. |
| avr_compat.h | Compatibility File used if you are using older version of AVR. It contains some definitions which are used in thee Stack definition. |
| timeout.h | It also contains some compatibility file includes related to delay functions. |

## 15.3 Defining MAC and IP Address

We have already discussed about MAC and IP address earlier. A MAC address is basically an array of six 8-bit integers. An IP address is basically an array of four 8-bit integers.

Representation of MAC Address

```
uint8_t mymac[6] = {0x54,0x90,0x58,0x10,0x65,0x77};
```

Here, mymac is the array of six 8-bit unsigned integers (uint8_t).

Representation of IP Address

```
uint8_t myip[6] = {10,0,0,50};
```

Here, myip is the array of four 8-bit unsigned integers (uint8_t).

## 15.4 Functional Overview

To work with the stack in a good manner, we need to go through all the functions which are provided in the stack library. Various functions available in the Stack are:

**TCP/IP Stack Functions=> Ip_arp_udp_tcp**

| | |
|---|---|
| **#1** | `void init_ip_arp_udp_tcp(uint8_t *mymac,uint8_t *myip,uint8_t wwwp);` |
| **Info** | This function initializes the Ethernet, Network, Transport layer of the Stack.<br>We need to provide a MAC address, IP address and the port on which web-server should be started(use 80 as default) as arguments to this function. |
| **#2** | `uint8_t eth_type_is_arp_and_my_ip(uint8_t *buf,uint8_t len);` |
| **Info** | This function checks if the packet received contains an ARP request. It also checks if the request is for our IP or not.<br>This function takes the Packet Buffer as input, and also the length of the packet. |
| **#3** | `eth_type_is_ip_and_my_ip(uint8_t *buf,uint8_t len);` |
| **Info** | This function checks if the packet is destined for us and contains some data like ICMP packet/TCP packet/UDP packet.<br>This function takes the Packet Buffer as input, and also the length of the packet. |
| **#4** | `void make_arp_answer_from_request(uint8_t *buf,uint8_t len);` |
| **Info** | This is used to create a packet as a response to ARP request.<br>This function takes the Packet Buffer as input, and also the length of the packet. |
| **#5** | `void make_echo_reply_from_request(uint8_t *buf,uint8_t len);` |
| **Info** | If someone on the network tries to ping us, we need to reply to that request. This function can be used to create the Reply packet.<br>This function takes the Packet Buffer as input, and also the length of the packet. |
| **#6** | `void make_tcp_synack_from_syn(uint8_t *buf);` |
| **Info** | This function is used in TCP handshaking. It acknowledges SYN request received from any device.<br>This function takes the Packet Buffer as input. |
| **#7** | `uint16_t get_tcp_data_pointer(void);` |
| **Info** | This function returns the address of TCP packet in the stack buffer as an integer.<br>This function takes no arguments. |
| **#8** | `uint16_t fill_tcp_data(uint8_t *buf,uint16_t pos, const char *s);` |
| **Info** | This function is used to add payload data into the Stack Buffer.<br>This function takes as input the Stack buffer (buf), the location to add the data (pos) and the data array itself. |
| **#9** | `void make_tcp_ack_from_any(uint8_t *buf);` |
| **Info** | This function makes a TCP acknowledgement packet in response of any data packet received. |

**ENC28J60 functions**

| #1 | `void enc28j60Init(uint8_t* macaddr);` |
|---|---|
| **Info** | This initializes the ENC28J60 IC. The Physical and data link layers are also initialized by calling this function.<br>It takes MAC address as argument. |
| **#2** | `void enc28j60PacketSend(uint16_t len, uint8_t* packet);` |
| **Info** | This function sends the packet to the Physical layer.<br>This function takes the Packet Buffer as input, and also the length of the packet. |
| **#3** | `uint16_t enc28j60PacketReceive(uint16_t maxlen, uint8_t* packet);` |
| **Info** | This function receives the packet from Physical layer.<br>It takes MAC address as argument. |
| **#4** | `void enc28j60PhyWrite(uint8_t address, uint16_t data);` |
| **Info** | This function writes the given data at the address given inside ENC28J60 hardware. We will be using this function to initialize the LEDs of ENC28J60. |

Some other functions are also available in those files. Go through the header files to know the extra functions available in the Stack and ENC28J60 driver.

# Chapter 16 - Appendix

## 16.1 ASCII Table

| Decimal | Hexadecimal | Binary | Octal | Char |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | [NULL] |
| 1 | 1 | 1 | 1 | [START OF HEADING] |
| 2 | 2 | 10 | 2 | [START OF TEXT] |
| 3 | 3 | 11 | 3 | [END OF TEXT] |
| 4 | 4 | 100 | 4 | [END OF TRANSMISSION] |
| 5 | 5 | 101 | 5 | [ENQUIRY] |
| 6 | 6 | 110 | 6 | [ACKNOWLEDGE] |
| 7 | 7 | 111 | 7 | [BELL] |
| 8 | 8 | 1000 | 10 | [BACKSPACE] |
| 9 | 9 | 1001 | 11 | [HORIZONTAL TAB] |
| 10 | A | 1010 | 12 | [LINE FEED] |
| 11 | B | 1011 | 13 | [VERTICAL TAB] |
| 12 | C | 1100 | 14 | [FORM FEED] |
| 13 | D | 1101 | 15 | [CARRIAGE RETURN] |
| 14 | E | 1110 | 16 | [SHIFT OUT] |
| 15 | F | 1111 | 17 | [SHIFT IN] |
| 16 | 10 | 10000 | 20 | [DATA LINK ESCAPE] |
| 17 | 11 | 10001 | 21 | [DEVICE CONTROL 1] |
| 18 | 12 | 10010 | 22 | [DEVICE CONTROL 2] |
| 19 | 13 | 10011 | 23 | [DEVICE CONTROL 3] |
| 20 | 14 | 10100 | 24 | [DEVICE CONTROL 4] |
| 21 | 15 | 10101 | 25 | [NEGATIVE ACKNOWLEDGE] |
| 22 | 16 | 10110 | 26 | [SYNCHRONOUS IDLE] |
| 23 | 17 | 10111 | 27 | [ENG OF TRANS. BLOCK] |
| 24 | 18 | 11000 | 30 | [CANCEL] |
| 25 | 19 | 11001 | 31 | [END OF MEDIUM] |
| 26 | 1A | 11010 | 32 | [SUBSTITUTE] |
| 27 | 1B | 11011 | 33 | [ESCAPE] |
| 28 | 1C | 11100 | 34 | [FILE SEPARATOR] |
| 29 | 1D | 11101 | 35 | [GROUP SEPARATOR] |
| 30 | 1E | 11110 | 36 | [RECORD SEPARATOR] |
| 31 | 1F | 11111 | 37 | [UNIT SEPARATOR] |
| 32 | 20 | 100000 | 40 | [SPACE] |
| 33 | 21 | 100001 | 41 | ! |
| 34 | 22 | 100010 | 42 | " |
| 35 | 23 | 100011 | 43 | # |
| 36 | 24 | 100100 | 44 | $ |
| 37 | 25 | 100101 | 45 | % |
| 38 | 26 | 100110 | 46 | & |
| 39 | 27 | 100111 | 47 | ' |
| 40 | 28 | 101000 | 50 | ( |
| 41 | 29 | 101001 | 51 | ) |
| 42 | 2A | 101010 | 52 | * |
| 43 | 2B | 101011 | 53 | + |
| 44 | 2C | 101100 | 54 | , |
| 45 | 2D | 101101 | 55 | - |
| 46 | 2E | 101110 | 56 | . |
| 47 | 2F | 101111 | 57 | / |
| 48 | 30 | 110000 | 60 | 0 |
| 49 | 31 | 110001 | 61 | 1 |
| 50 | 32 | 110010 | 62 | 2 |
| 51 | 33 | 110011 | 63 | 3 |
| 52 | 34 | 110100 | 64 | 4 |
| 53 | 35 | 110101 | 65 | 5 |
| 54 | 36 | 110110 | 66 | 6 |
| 55 | 37 | 110111 | 67 | 7 |
| 56 | 38 | 111000 | 70 | 8 |
| 57 | 39 | 111001 | 71 | 9 |
| 58 | 3A | 111010 | 72 | : |
| 59 | 3B | 111011 | 73 | ; |
| 60 | 3C | 111100 | 74 | < |
| 61 | 3D | 111101 | 75 | = |
| 62 | 3E | 111110 | 76 | > |
| 63 | 3F | 111111 | 77 | ? |
| 64 | 40 | 1000000 | 100 | @ |
| 65 | 41 | 1000001 | 101 | A |
| 66 | 42 | 1000010 | 102 | B |
| 67 | 43 | 1000011 | 103 | C |
| 68 | 44 | 1000100 | 104 | D |
| 69 | 45 | 1000101 | 105 | E |
| 70 | 46 | 1000110 | 106 | F |
| 71 | 47 | 1000111 | 107 | G |
| 72 | 48 | 1001000 | 110 | H |
| 73 | 49 | 1001001 | 111 | I |
| 74 | 4A | 1001010 | 112 | J |
| 75 | 4B | 1001011 | 113 | K |
| 76 | 4C | 1001100 | 114 | L |
| 77 | 4D | 1001101 | 115 | M |
| 78 | 4E | 1001110 | 116 | N |
| 79 | 4F | 1001111 | 117 | O |
| 80 | 50 | 1010000 | 120 | P |
| 81 | 51 | 1010001 | 121 | Q |
| 82 | 52 | 1010010 | 122 | R |
| 83 | 53 | 1010011 | 123 | S |
| 84 | 54 | 1010100 | 124 | T |
| 85 | 55 | 1010101 | 125 | U |
| 86 | 56 | 1010110 | 126 | V |
| 87 | 57 | 1010111 | 127 | W |
| 88 | 58 | 1011000 | 130 | X |
| 89 | 59 | 1011001 | 131 | Y |
| 90 | 5A | 1011010 | 132 | Z |
| 91 | 5B | 1011011 | 133 | [ |
| 92 | 5C | 1011100 | 134 | \ |
| 93 | 5D | 1011101 | 135 | ] |
| 94 | 5E | 1011110 | 136 | ^ |
| 95 | 5F | 1011111 | 137 | _ |
| 96 | 60 | 1100000 | 140 | ` |
| 97 | 61 | 1100001 | 141 | a |
| 98 | 62 | 1100010 | 142 | b |
| 99 | 63 | 1100011 | 143 | c |
| 100 | 64 | 1100100 | 144 | d |
| 101 | 65 | 1100101 | 145 | e |
| 102 | 66 | 1100110 | 146 | f |
| 103 | 67 | 1100111 | 147 | g |
| 104 | 68 | 1101000 | 150 | h |
| 105 | 69 | 1101001 | 151 | i |
| 106 | 6A | 1101010 | 152 | j |
| 107 | 6B | 1101011 | 153 | k |
| 108 | 6C | 1101100 | 154 | l |
| 109 | 6D | 1101101 | 155 | m |
| 110 | 6E | 1101110 | 156 | n |
| 111 | 6F | 1101111 | 157 | o |
| 112 | 70 | 1110000 | 160 | p |
| 113 | 71 | 1110001 | 161 | q |
| 114 | 72 | 1110010 | 162 | r |
| 115 | 73 | 1110011 | 163 | s |
| 116 | 74 | 1110100 | 164 | t |
| 117 | 75 | 1110101 | 165 | u |
| 118 | 76 | 1110110 | 166 | v |
| 119 | 77 | 1110111 | 167 | w |
| 120 | 78 | 1111000 | 170 | x |
| 121 | 79 | 1111001 | 171 | y |
| 122 | 7A | 1111010 | 172 | z |
| 123 | 7B | 1111011 | 173 | { |
| 124 | 7C | 1111100 | 174 | \| |
| 125 | 7D | 1111101 | 175 | } |
| 126 | 7E | 1111110 | 176 | ~ |
| 127 | 7F | 1111111 | 177 | [DEL] |

Taken from: http://lehre.hki.uni-koeln.de/seminare/sites/default/files/userfiles/1/1000px-ASCII-Table.jpg

## 16.2 Big and Little Endian

A long time ago, in a very remote island known as Lilliput, society was split into two factions: Big-Endians who opened their soft-boiled eggs at the larger end ("the primitive way") and Little-Endians who broke their eggs at the smaller end. As the Emperor commanded all his subjects to break the smaller end, this resulted in a civil war with dramatic consequences: 11.000 people have, at several times, suffered death rather than submitting to breaking their eggs at the smaller end. Eventually, the 'Little-Endian' vs. 'Big-Endian' feud carried over into the world of computing as well, where it refers to the order in which bytes in multi-byte numbers should be stored, most-significant first (Big-Endian) or least-significant first (Little-Endian) to be more precise.

- **Big-Endian** means that the most significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.
- **Little-Endian** means that the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.

For example, consider the 32-bit number, **0xDEADBEEF**. Following the Big-Endian convention, a computer will store it as follows:

| Memory Location | Value |
|---|---|
| Base Address + 0 | DE |
| Base Address + 1 | AD |
| Base Address + 2 | BE |
| Base Address + 3 | EF |

**Big-Endian:** The most significant byte is stored at the lowest byte address.

Whereas architectures that follow the Little-Endian rules will store it as depicted in following Figure

| Memory Location | Value |
|---|---|
| Base Address + 0 | EF |
| Base Address + 1 | BE |
| Base Address + 2 | AD |
| Base Address + 3 | DE |

**Little-Endian:** Least significant byte is stored at the lowest byte address.

The Intel x86 family and Digital Equipment Corporation architectures (PDP-11, VAX, Alpha) are representatives of Little-Endian, while the Sun SPARC, IBM 360/370, and Motorola 68000 and 88000 architectures are Big-Endians. Still, other architectures such as PowerPC, MIPS, and Intel's 64 IA-64 are Bi-Endian, i.e. they are capable of operating in either Big-Endian or Little-Endian mode.

## 16.3 Two disasters caused by computer arithmetic errors

Just for information ☺

### Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soliders. A report of the General Accounting office, GAO/IMTEC-92-26, entitled *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia* reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. Specifically, the time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, lead to a significant error. Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds. (The number 1/10 equals $1/2^4+1/2^5+1/2^8+1/2^9+1/2^{12}+1/2^{13}+....$ In other words, the binary expansion of 1/10 is 0.0001100110011001100110011001100.... Now the 24 bit register in the Patriot stored instead 0.00011001100110011001100 introducing an error of 0.0000000000000000000000011001100... binary, or about 0.000000095 decimal. Multiplying by the number of tenths of a second in 100 hours gives 0.000000095×100×60×60×10=0.34.) A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.

The following paragraph is excerpted from the GAO report.

*The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563...miles per hour). Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g., 32, 33, 34...). The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of the the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted.*

## Explosion of the Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

The following paragraphs are extracted from report of the Inquiry Board.

*On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.*

*The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.*

*The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*

*SRI stands for Système de Référence Inertielle or Inertial Reference System.

## 16.4 Experiment List Advanced Embedded Course HPES

| | Experiment | Tool | Peripheral | Done |
|---|---|---|---|---|
| 0 | LED Glow | AVR-GCC | GPIO-LED | |
| 1 | LED Glow | AVR-Studio | GPIO-LED | |
| 2 | LED Blink | AVR-Studio | GPIO-LED | |
| 3 | LED Pattern | AVR-Studio | GPIO-LED | |
| 4 | Debugging | AVR-Studio | GPIO-LED | |
| 5 | LED Glow | Make-File | GPIO-LED | |
| 6 | LED Blink | Make-File | GPIO-LED | |
| 7 | LED Pattern | Make-File | GPIO-LED | |
| 8 | Switch-LED-Glow | AVR-Studio | GPIO-LED-Switch | |
| 9 | Switch-LED Blink | AVR-Studio | GPIO-LED-Switch | |
| 10 | Switch-LED Pattern | AVR-Studio | GPIO-LED-Switch | |
| 11 | Debugging | AVR-Studio | GPIO-LED-Switch | |
| 12 | Switch-LED-Glow | Make-File | GPIO-LED-Switch | |
| 13 | Switch-LED Blink | Make-File | GPIO-LED-Switch | |
| 14 | Switch-LED Pattern | Make-File | GPIO-LED-Switch | |
| 15 | LCD Display String | AVR-Studio | LCD | |
| 16 | LCD Display Integer | AVR-Studio | LCD | |
| 17 | LCD Display Float | AVR-Studio | LCD | |
| 18 | LCD Moving String | AVR-Studio | LCD | |
| 19 | Switch Control LCD | AVR-Studio | GPIO-LCD-Switch | |
| 20 | Debugging | AVR-Studio | GPIO-LED-Switch | |
| 21 | LCD Display String | Make-File | LCD | |
| 22 | LCD Display Integer | Make-File | LCD | |
| 23 | LCD Display Float | Make-File | LCD | |
| 24 | LCD Moving String | Make-File | LCD | |
| 25 | Switch Control LCD | Make-File | GPIO-LCD-Switch | |
| 26 | Relay Control via Switch/Delay | Make-File | GPIO-Relay | |
| 27 | Sending Data to PC | Make-File | UART-FTDI | |
| 28 | Receiving Data to uC Display on LCD | Make-File | UART-FTDI | |
| 29 | Controlling LED/LCD through PC | Make-File | UART-FTDI | |
| 30 | Relay Control via PC | Make-File | Relay-UART-FTDI | |
| 31 | EEPROM Data saving | Make-File | EEPROM | |
| 32 | EEPROM Data configuring via PC | Make-File | EEPROM-UART | |
| 33 | ENC-MagJack LED Blink | Make-File | SPI-ENC28J60 | |
| 34 | TMP-275 High byte Read to LCD/PC | Make-File | I2C-TMP275 | |
| 35 | TMP-275 Low byte Read to LCD/PC | Make-File | I2C-TMP275 | |
| 36 | TMP-275 both byte Read to LCD/PC | Make-File | I2C-TMP275 | |
| 37 | TMP-275 to Celcius driver Read to LCD/PC | Make-File | I2C-TMP275 | |
| 38 | APDS9300 Channel-1 Read to LCD/PC | Make-File | I2C-APDS9300 | |
| 39 | APDS9300 Channel-2 Read to LCD/PC | Make-File | I2C-APDS9300 | |

| | | | | |
|---|---|---|---|---|
| 40 | APDS9300 both Channel Read to LCD/PC | Make-File | I2C-APDS9300 | |
| 41 | APDS9300 to Lux driver Read to LCD/PC | Make-File | I2C-APDS9300 | |
| 42 | CC2500 open Mode Rx Tx short Test | Make-File | UART-CC2500 | |
| 43 | CC2500 open Mode uC to PC Test | Make-File | UART-CC2500 | |
| 44 | CC2500 config Mode uC to PC Test | Make-File | UART-CC2500 | |
| 45 | CC2500 config Mode PC to uC Test | Make-File | UART-CC2500 | |
| 46 | CC2500 config Mode uC to uC Test | Make-File | UART-CC2500 | |
| 47 | CC2500 Temperature uC to Other Board LCD/PC | Make-File | UART-CC2500 | |
| 48 | CC2500 Light uC to other Board LCD/PC | Make-File | UART-CC2500 | |
| 49 | CC2500 Relay Control via Temp/Light Sensor | Make-File | Relay-UART-CC2500 | |
| 50 | ENC Echo-Ping | Make-File | SPI-ENC | |
| 51 | ENC Plain Webserver | Make-File | SPI-ENC | |
| 52 | ENC http GET Packet on PC | Make-File | SPI-ENC | |
| 53 | ENC Temperature on Webserver | Make-File | SPI-ENC-TMP275 | |
| 54 | ENC Lux on Webserver | Make-File | SPI-ENC-APDS9300 | |
| 55 | ENC LED Control | Make-File | SPI-ENC-LED | |
| 56 | ENC Relay Control | Make-File | SPI-ENC-Relay | |
| 57 | ENC Webserver CC2500 LED Control | Make-File | SPI-ENC-LED-CC2500 | |
| 58 | ENC Webserver CC2500 Relay Control | Make-File | SPI-ENC-Relay-CC2500 | |
| 59 | ENC Telnet Server | Make-File | SPI-ENC | |
| 60 | ENC Telnet Server-EEPROM configuration | Make-File | SPI-ENC-EEPROM | |
| 61 | Eagle - AVR JTAG schematic design | N.A. | Eagle | |
| 62 | Eagle - AVR JTAG Board design | N.A. | Eagle | |
| 63 | Proteus - Simulating basic code on AVR | Any | Proteus | |
| 64 | Proteus - Simulating UART on AVR | Any | Proteus | |
| 65 | Proteus - Simulating basic code on ARM | Any | Proteus | |

# Chapter 17 - References


[1] Wikipedia: http://www.wikipedia.org
[2] Atmel: http://www.atmel.com
[3] Oreilly: http://www.oreilly.com
[4] Microsoft Support: http://support.microsoft.com
[5] NPTel: http://nptel.iitm.ac.in
[6] TuxGraphics: http://www.tuxgraphics.org
[7] Firewall.cx: http://firewall.cx
[8] The Story of Ping: http://ftp.arl.army.mil/~mike/ping.html
[9] Security-Freak: http://www.security-freak.net
[10] Wireless Sensor Network Topologies: http://www.stevekos.com
[11] CodeProject: http://www.codeproject.com
[12] AVRFreaks: http://www.avrfreaks.net
[13] Robot Electronics: http://www.robot-electronics.co.uk
[14] Ping: http://www.codeidol.com