UNIT-I - OVERVIEW OF EMBEDDED SYSTEMS

Embedded System

. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. For example, a fire alarm is an embedded system; it will sense only smoke.

An embedded system has three components −

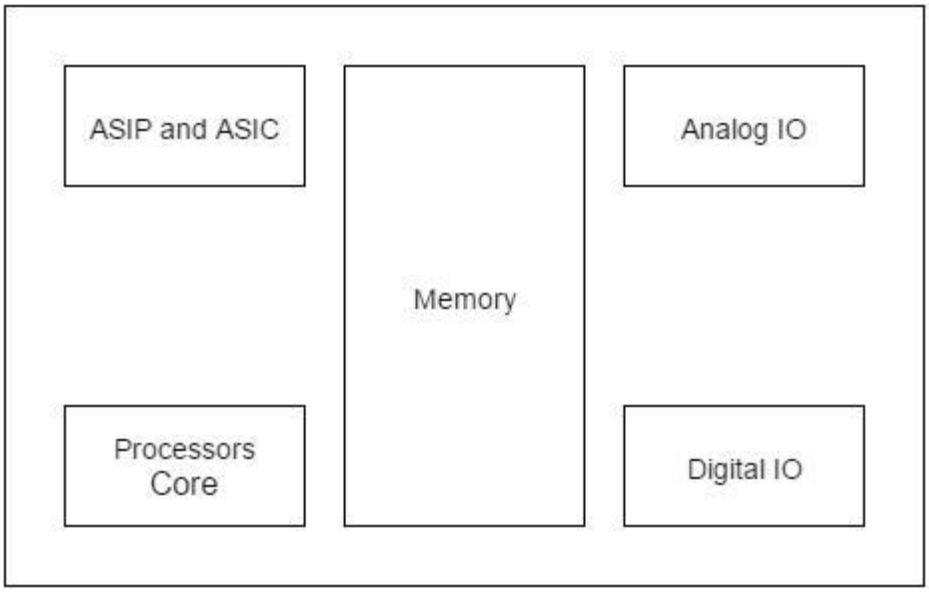
- It has hardware.
- It has application software.
- It has Real Time Operating system (RTOS) that supervises the application software and provide mechanism to let the processor run a process as per scheduling by following a plan to control the latencies. RTOS defines the way the system works. It sets the rules during the execution of application program. A small scale embedded system may not have RTOS.

So we can define an embedded system as a Microcontroller based, software driven, reliable, real-time control system.

Characteristics of an Embedded System

- **Single-functioned** − An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.
- **Tightly constrained** − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.

- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- **Microprocessors based** − It must be microprocessor or microcontroller based.
- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- **Connected** − It must have connected peripherals to connect input and output devices.
- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.



Advantages

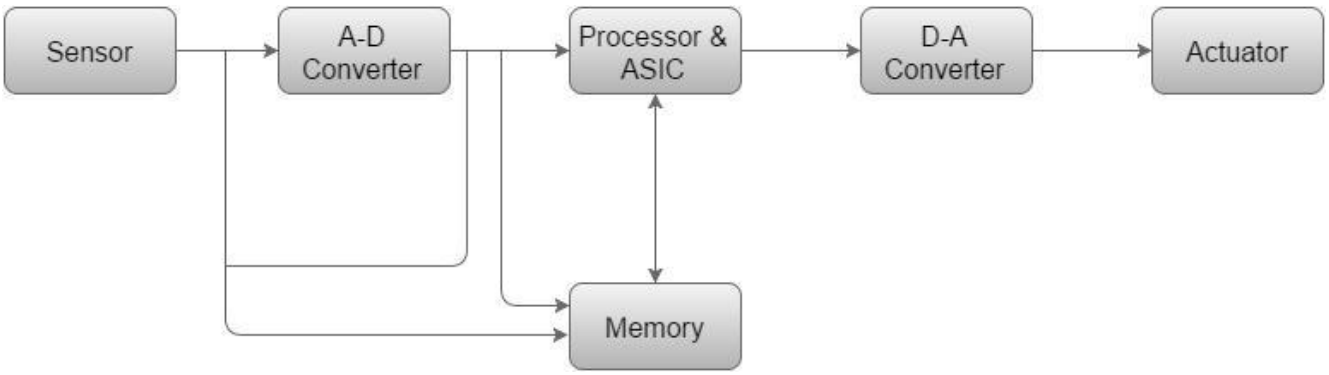
- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

Disadvantages

- High development effort
- Larger time to market

Basic Structure of an Embedded System

The following illustration shows the basic structure of an embedded system −



- **Sensor** − It measures the physical quantity and converts it to an electrical signal which can be read by an observer or by any electronic instrument like an A2D converter. A sensor stores the measured quantity to the memory.
- **A-D Converter** − An analog-to-digital converter converts the analog signal sent by the sensor into a digital signal.
- **Processor & ASICs** − Processors process the data to measure the output and store it to the memory.
- **D-A Converter** − A digital-to-analog converter converts the digital data fed by the processor to analog data
- **Actuator** − An actuator compares the output given by the D-A Converter to the actual (expected) output stored in it and stores the approved output.

Processor is the heart of an embedded system. It is the basic unit that takes inputs and produces an output after processing the data. For an embedded system designer, it is necessary to have the knowledge of both microprocessors and microcontrollers.

Processors in a System

A processor has two essential units −

- Program Flow Control Unit (CU)
- Execution Unit (EU)

The CU includes a fetch unit for fetching instructions from the memory. The EU has circuits that implement the instructions pertaining to data transfer operation and data conversion from one form to another.

The EU includes the Arithmetic and Logical Unit (ALU) and also the circuits that execute instructions for a program control task such as interrupt, or jump to another set of instructions.

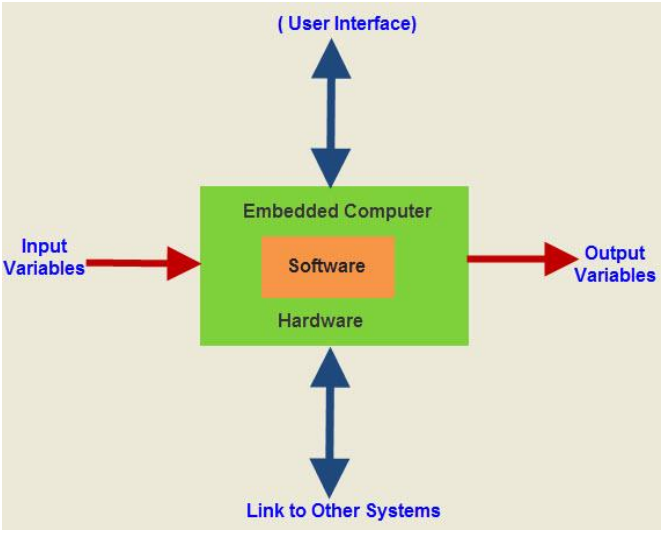
A processor runs the cycles of fetch and executes the instructions in the same sequence as they are fetched from memory.

Types of Processors

Processors can be of the following categories −

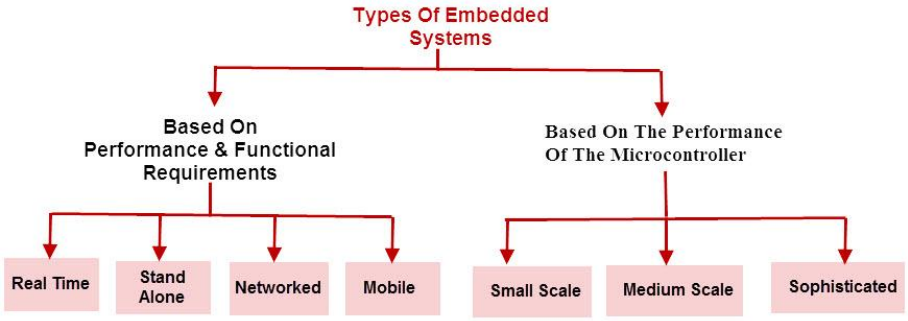
- General Purpose Processor (GPP)
    - Microprocessor
    - Microcontroller
    - Embedded Processor
    - Digital Signal Processor
    - Media Processor
- Application Specific System Processor (ASSP)
- Application Specific Instruction Processors (ASIPs)

.The Embedded system hardware includes elements like user interface, Input/Output interfaces, display and memory, etc.Generally, an embedded system comprises power supply, processor, memory, timers, serial communication ports and system application specific circuits.



Types of Embedded Systems

Embedded systems can be classified into different types based on performance, functional requirements and performance of the microcontroller.



Types of Embedded systems

Embedded systems are classified into four categories based on their performance and functional requirements:

- Stand alone embedded systems
- Real time embedded systems

- Networked embedded systems
- Mobile embedded systems

Embedded Systems are classified into three types based on the performance of the microcontroller such as

- Small scale embedded systems
- Medium scale embedded systems
- Sophisticated embedded systems

Stand Alone Embedded Systems

Stand alone embedded systems do not require a host system like a computer, it works by itself. It takes the input from the input ports either analog or digital and processes, calculates and converts the data and gives the resulting data through the connected device-Which either controls, drives and displays the connected devices. Examples for the stand alone embedded systems are mp3 players, digital cameras, video game consoles, microwave ovens and temperature measurement systems.

Real Time Embedded Systems

A real time embedded system is defined as, a system which gives a required o/p in a particular time.These types of embedded systems follow the time deadlines for completion of a task. Real time embedded systems are classified into two types such as soft and hard real time systems.

Networked Embedded Systems

These types of embedded systems are related to a network to access the resources. The connected network can be LAN, WAN or the internet. The connection can be any wired or wireless. This type of embedded system is the fastest growing area in embedded system applications. The embedded web server is a type of system wherein all embedded devices are connected to a web server and accessed and controlled by a web browser. Example for the LAN networked embedded system is a home security system wherein all sensors are connected and run on the protocol TCP/IP

Mobile Embedded Systems

Mobile embedded systems are used in portable embedded devices like cell phones, mobiles, digital cameras, mp3 players and personal digital assistants, etc.The basic limitation of these devices is the other resources and limitation of memory.

Small Scale Embedded Systems

These types of embedded systems are designed with a single 8 or 16-bit microcontroller, that may even be activated by a battery. For developing embedded software for small scale embedded systems, the main programming tools are an editor, assembler, cross assembler and integrated development environment (IDE).

Medium Scale Embedded Systems

These types of embedded systems design with a single or 16 or 32 bit microcontroller, RISCs or DSPs. These types of embedded systems have both hardware and software complexities. For developing embedded software for medium scale embedded systems, the main programming tools are C, C++, JAVA, Visual C++, RTOS, debugger, source code engineering tool, simulator and IDE.

Sophisticated Embedded Systems

These types of embedded systems have enormous hardware and software complexities, that may need ASIPs, IPs, PLAs, scalable or configurable processors. They are used for cutting-edge applications that need hardware and software Co-design and components which have to assemble in the final system.

Applications of Embedded Systems:

Embedded systems are used in different applications like automobiles, telecommunications, smart cards, missiles, satellites, computer networking and digital consumer electronics.

Embedded System Initialization

It takes just minutes for a developer to compile and run a Hello World! application on a non-embedded system. On the other hand, for an embedded developer, the task is not so trivial. It might take days before seeing a successful result. This process can be a frustrating experience for a developer new to embedded system development.

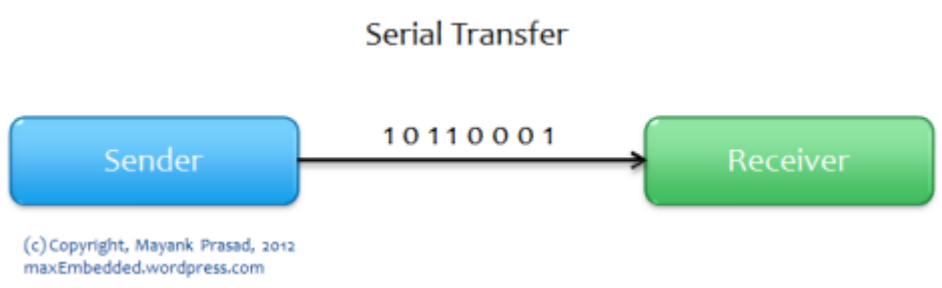
Booting the target system, whether a third-party evaluation board or a custom design, can be a mystery to many newcomers. Indeed, it is daunting to pick up a programmer s reference manual for the target board and pore over tables of memory addresses and registers or to review the hardware component interconnection diagrams, wondering what it all means, what to do with the information (some of which makes little sense), and how to relate the information to running an image on the target system.

Questions to resolve at this stage are

- how to load the image onto the target system,
- where in memory to load the image,
- how to initiate program execution, and
- how the program produces recognizable output.

We answer these questions in this chapter and hopefully reduce frustration by demystifying the booting and initialization process of embedded systems.

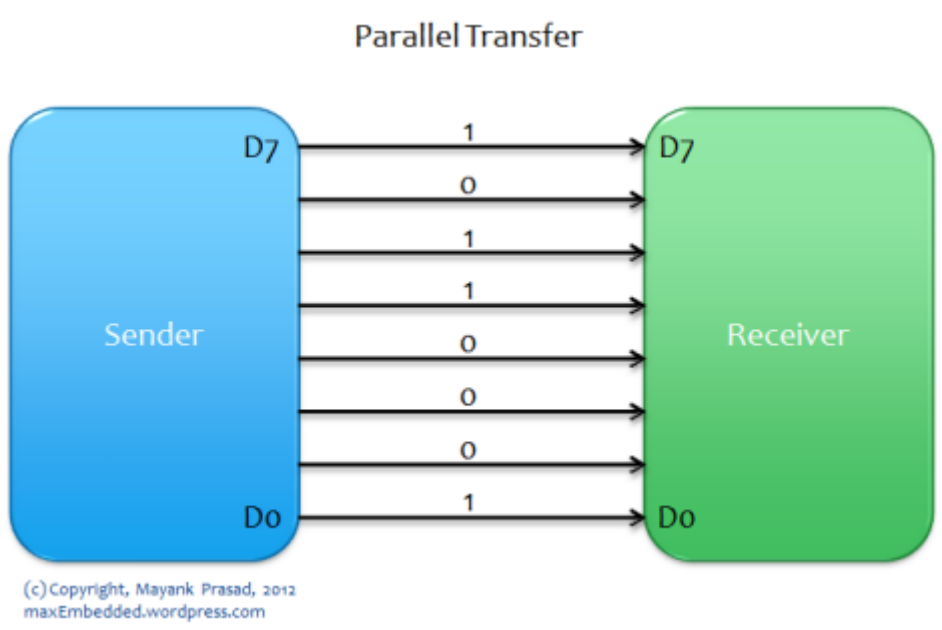
Serial Communication



Serial Transfer

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

Parallel Communication



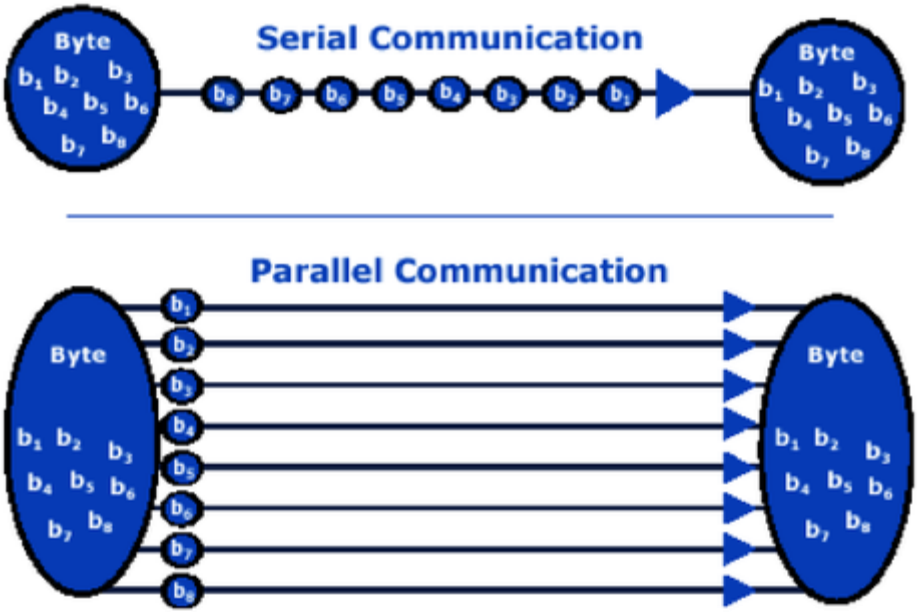
Parallel Transfer

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

Serial vs Parallel Communication

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

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



Input/output Devices

The Address Bus

Recall from our discussion earlier about microprocessors, that every CPU has a number of pins, which work together, called an address bus. The address bus is normally used to read or write to memory, most often RAM chips. Most modern microprocessors use the address bus for more than just reading and writing to memory however.

By toggling a special pin, the CPU can switch from using the address bus for accessing RAM, to using the address bus to talk to other semi-intelligent chips that are also connected to the address bus. When used in this way, we are said to be using I/O port addressing, instead of normal memory addresses. Sometimes a port will be referred to as a register, but I find this a bit confusing, since a register normally means an internal CPU register. The semi-intelligent device chips are only activate when they detect that the special I/O pin is asserted and the address bus holds the memory value that points to that specific chip.

This is how most input and output occurs from devices like serial ports, parallel ports, floppy, hard drive and other controllers. Once the CPU has placed the proper address on the address bus and it asserts the special I/O pin, all RAM chips are temporarily disabled and the external I/O chips are read or written from instead. The bytes of data are actually transferred on a second set of pins called the data bus.

The Data Bus

The data bus is nothing more than a series of pins on the processor that are used to get data into, or out of, the processor chip itself. All memory and I/O devices are connected to the data bus, but depending on the current state of the address bus and other control pins on the processor, only one chip can actually be connected to the data bus at any given moment.

Depending on the exact processor used, the data bus may be 4, 8, 16, 32 or perhaps 64-bits wide. A wider data bus allows the processor to read and write more bits of data in a single operation. This technique is used with PCI-based cards on PC-compatibles to achieve faster I/O operations for certain devices. In other cases however, using more bits is a waste of time, because the device connected at the other end of the data bus only supports 4 or 8 bit transfers at a time. In this case it is very important to ignore the unused bits, generally by using a bit masking operation to force the unused bits to a zero value.

Interrupt Requests

In addition to the processor using the data bus, address bus and special I/O pin to communicate with external devices; the external devices use another pin when they need the attention of the

processor. This is referred to as an Interrupt Request Line or IRQ Line. For example, whenever you press a key on the keyboard, the keyboard controller device generally signals the main processor that a key is available by asserting the interrupt line.

The interrupt handler must be small and efficiently designed, since in some cases it could be invoked hundreds or maybe even thousands of times a second. Generally an interrupt handler performs the minimum amount of work necessary to service the device, and then exits. At that point, the processor returns to running the process that was interrupted as if nothing happened.

There are normally two different types of interrupt lines on all processors. The first is the kind we have been discussing at this point, called maskable interrupts. Maskable in this case means that interrupts can be selectively enabled or disabled by the software. The other kind of interrupt is called a non-maskable interrupt. The software can never disable this kind of interrupt. It most often used to perform the DRAM refresh on memory chips, which MUST occur at regular intervals in order to keep memory contents alive.

Memory Mapped I/O

I/O Port addressing is not the only way the processor can communicate with external devices however. Another commonly used technique is called memory mapped I/O. In this case, instead of asserting the I/O pin and addressing a data port, the processor just accesses a memory address directly. The external device can have a small amount of RAM or ROM that the processor just reads or writes as needed.

Direct Memory Access

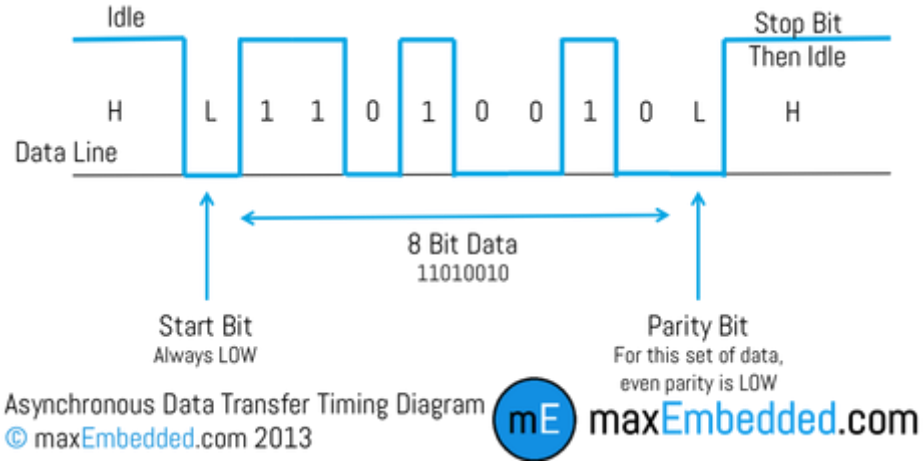
One technique that has been used for years to speed transfer of data from main memory to an external device's memory is the direct memory access feature (DMA). The processor on the external device executes DMA transfers, without any assistance from the main processor. The processors must cooperate for this to work obviously. While the DMA transfer is in progress, the main processor is free to tend to other tasks, but should not attempt to modify the information in the buffer being transfer, until the transfer is complete.

Once the transfer is stared, the main processor is free to tend to other tasks. The external processor will take over the address and data lines periodically and execute the DMA transfer. Once the transfer is complete, the external device usually notifies the main processor of this by raising an interrupt request.

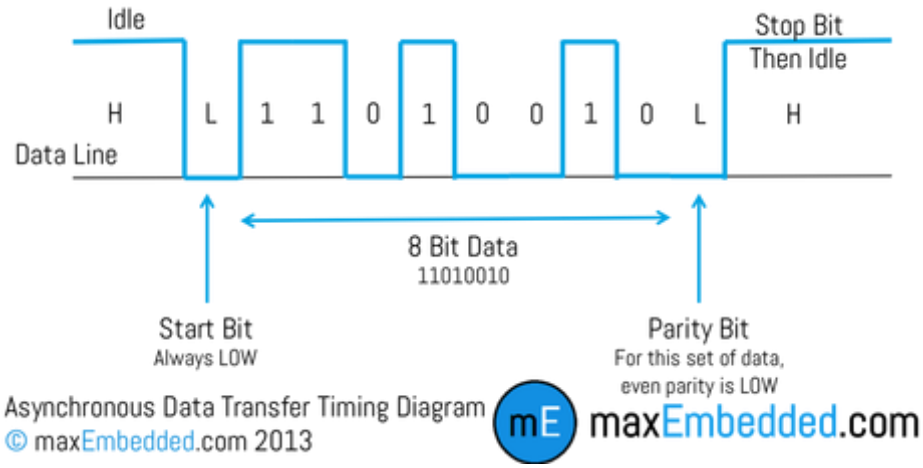
DMA's main advantage is that the main processor does not have to transfer data into one of its register, then save that to a memory address for each and every byte of data. Another advantage is the fact that whiles the DMA transfer is in progress, the CPU is free to work on other tasks. This leads to an apparently overall increase in speed.

Synchronous, Asynchronous and Iso-Synchronous Communication

1- In **Synchronous** data transfer, each basic unit of data (such as a bit) is transferred in accordance to a clock COMMUNICATION signal or in other words the data is transferred at a pre-decided rate. So for this data transfer method a clock signal is needed. Moreover Synchronous data transfer systems usually have an error checking mechanism to guarantee data integrity over transmission.



2- In **Asynchronous** data transfer systems, the data can be sent at irregular intervals and there is no pre-decided data rate of transmission. Special bits such as Start and stop bits are reserved to detect the start and end of data transmission in these systems and they are also equipped with an error checking mechanism.



3-**Isochronous** data transfer lies somewhat in between the two other data transmission types. It sends Asynchronous data over a Synchronous transmission system. In such systems each data source is given only a fixed time to transmit its data. In that fixed interval of time, that data source can transfer data at whatever intervals it wants. If it has data which requires less time than the time allotted then it simply wastes the extra time by staying idle. Otherwise if it has data which requires more time to transmit than given then it sends the remaining data in its next turn. These systems do not have error check mechanism because it is not possible to re-transmit the data after an error due to strict timing conditions.

Synchronous, asynchronous, and isosynchronous transmission are not three of a kind but two unrelated pairs, where the asynchronous transmission that differs from synchronous transmission may not be the same as the asynchronous transmission that differs from isosynchronous transmission. Of course, both pairs are about timing (see Chronos in Wikipedia).

Serial Communication Protocols

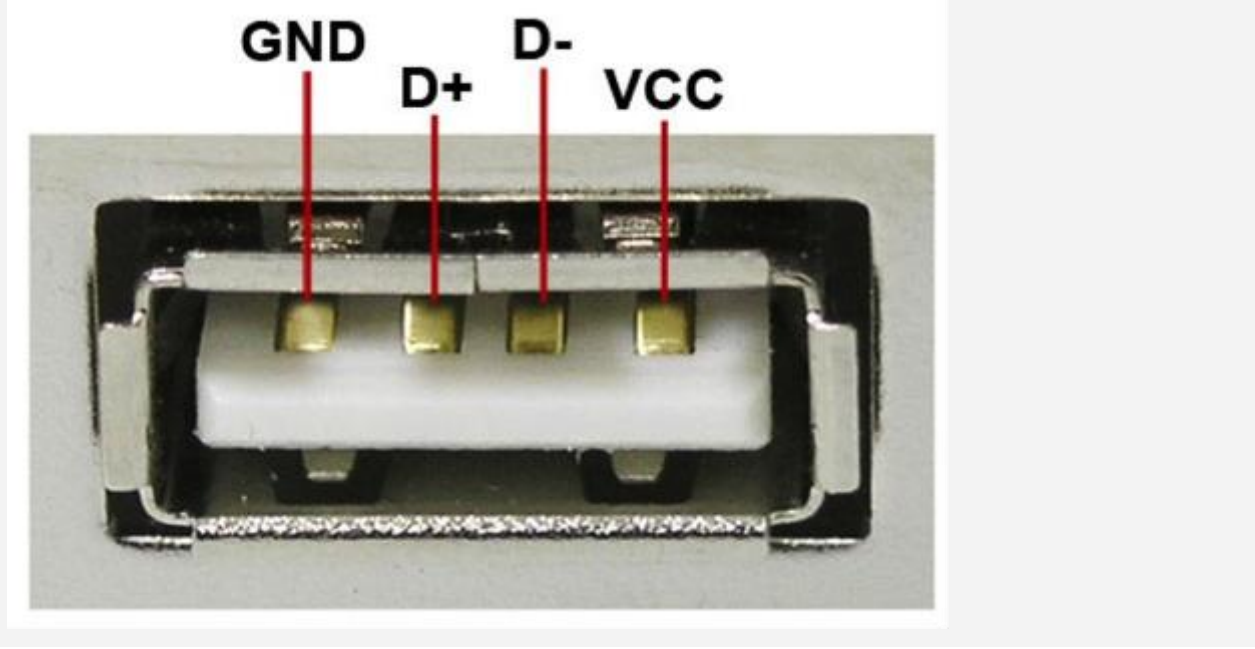
A variety of communication protocols have been developed based on serial communication in the past few decades. Some of them are:

1. **SPI – Serial Peripheral Interface:** It is a three-wire based communication system. One wire each for Master to slave and Vice-versa, and one for clock pulses. There is an additional SS (Slave Select) line, which is mostly used when we want to send/receive data between multiple ICs.
2. **I2C – Inter-Integrated Circuit:** Pronounced eye-two-see or eye-square-see, this is an advanced form of USART. The transmission speeds can be as high as a whopping 400KHz. The I2C bus

has two wires – one for clock, and the other is the data line, which is bi-directional – this being the reason it is also sometimes (not always – there are a few conditions) called **Two Wire Interface (TWI)**. It is a pretty new and revolutionary technology invented by Philips.

3. **FireWire** – Developed by Apple, they are high-speed buses capable of audio/video transmission. The bus contains a number of wires depending upon the port, which can be either a 4-pin one, or a 6-pin one, or an 8-pin one.

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



USB Pins

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

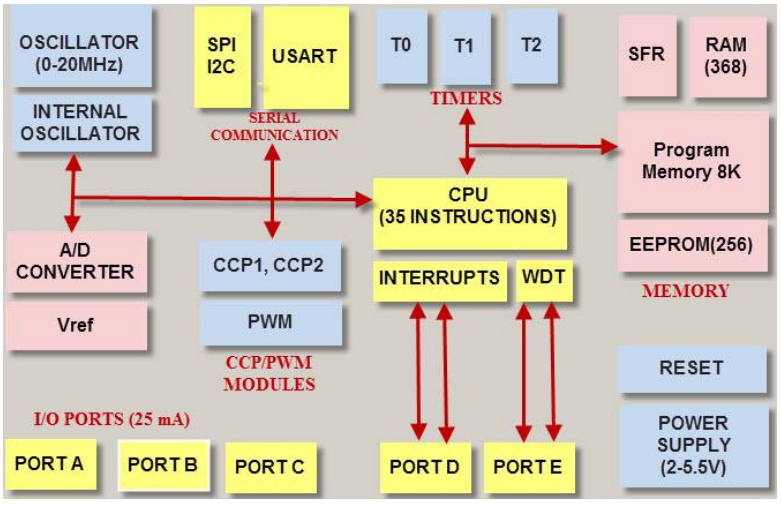
Unit-II - CPU Architecture of PIC Microcontroller

PIC Microcontroller : PIC (Programmable Interface Controllers) microcontrollers are the world's smallest microcontrollers that can be programmed to carry out a huge range of tasks. These microcontrollers are found in many electronic devices such as phones, computer control systems, alarm systems, embedded systems, etc. Various types of microcontrollers exist, even though the best are found in the GENIE range of programmable microcontrollers. These microcontrollers are programmed and simulated by circuit-wizard software.

Every PIC microcontroller architecture consists of some registers and stack where registers function as Random Access Memory(RAM) and stack saves the return addresses. The main features of PIC microcontrollers are RAM, flash memory, Timers/Counters, EEPROM, I/O Ports, USART, CCP (Capture/Compare/PWM module), SSP, Comparator, ADC (analog to digital converter), PSP(parallel slave port), LCD and ICSP (in circuit serial programming) The 8-bit PIC microcontroller is classified into four types on the basis of internal architecture such as Base Line PIC, Mid Range PIC, Enhanced Mid Range PIC and PIC18

Architecture of PIC Microcontroller

The PIC microcontroller architecture comprises of CPU, I/O ports, memory organization, A/D converter, timers/counters, interrupts, serial communication, oscillator and CCP module which are discussed in detailed below.



CPU (Central Processing Unit)

It is not different from other microcontrollers CPU and the PIC microcontroller CPU consists of the ALU, CU, MU and accumulator, etc. Arithmetic logic unit is mainly used for arithmetic operations and to take logical decisions. Memory is used for storing the instructions after processing. To control the internal and external peripherals, control unit is used which are connected to the CPU and the accumulator is used for storing the results and further process.

Memory Organization

The memory module in the PIC *microcontroller* architecture consists of RAM (Random Access Memory), ROM (Read Only Memory) and STACK.

Random Access Memory (RAM)

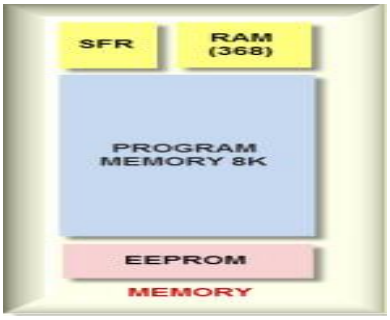
RAM is an unstable memory which is used to store the data temporarily in its registers. The RAM memory is classified into two banks, and each bank consists of so many registers. The RAM registers are classified into two types: Special Function Registers (SFR) and General Purpose Registers (GPR).

General Purpose Registers (GPR)

These registers are used for general purpose only as the name implies. For example, if we want to multiply two numbers by using the PIC microcontroller. Generally, we use registers for multiplying and storing the numbers in other registers. So these registers don't have any special function,- CPU can easily access the data in the registers.

- *Special Function Registers*

These registers are used for special purposes only as the name SFR implies. These registers will perform according to the functions assigned to them, and they cannot be used as normal registers. For example, if you cannot use the STATUS register for storing the data, these registers are used for showing the operation or status of the program. So, user cannot change the function of the SFR; the function is given by the retailer at the time of manufacturing.



Memory Organization

Read Only Memory (ROM)

Read only memory is a stable memory which is used to store the data permanently. In PIC microcontrollerarchitecture, the architecture ROM stores the instructions or program, according to the program the microcontroller acts. The ROM is also called as program memory, wherein the user will write the program for microcontroller and saves it permanently, and finally the program is executed by the CPU. The microcontrollers performance depends on the instruction, which is executed by the CPU.

Electrically Erasable Programmable Read Only Memory (EEPROM)

In the normal ROM, we can write the program for only once we cannot use again the microcontroller for multiple times. But, in the EEPROM, we can program the ROM multiple times.

Flash Memory

Flash memory is also programmable read only memory (PROM) in which we can read, write and erase the program thousands of times. Generally, the PIC microcontroller uses this type of ROM.

Stack

When an interrupt occurs, first the PIC microcontroller has to execute the interrupt and the existing process address. Then that is being executed is stored in the stack. After completing the

execution of the interrupt, the microcontroller calls the process with the help of address, which is stored in the stack and get executes the process.

I/O Ports

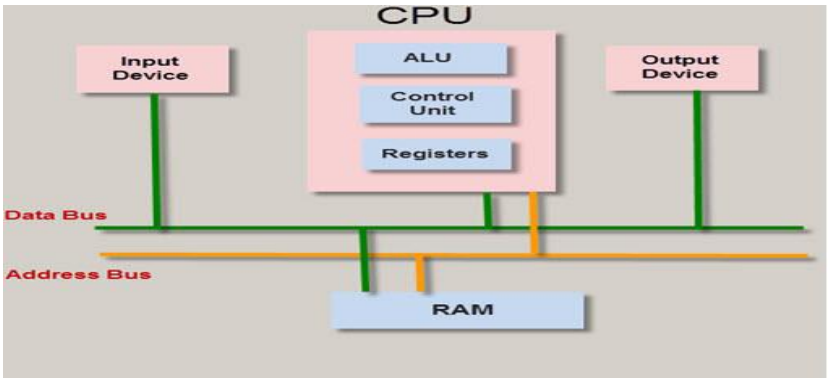
- The series of PIC16 consists of five ports such as Port A, Port B, Port C, Port D & Port E.
- Port A is an 16-bit port that can be used as input or output port based on the status of the TRISA (Tradoc Intelligence Support Activity) register.
- Port B is an 8- bit port that can be used as both input and output port.
- Port C is an 8-bit and the input of output operation is decided by the status of the TRISC register.
- Port D is an 8-bit port acts as a slave port for connection to the microprocessor BUS.
- Port E is a 3-bit port which serves the additional function of the control signals to the analog to digital converter.

BUS

BUS is used to transfer and receive the data from one peripheral to another. It is classified into two types such as data bus and address.

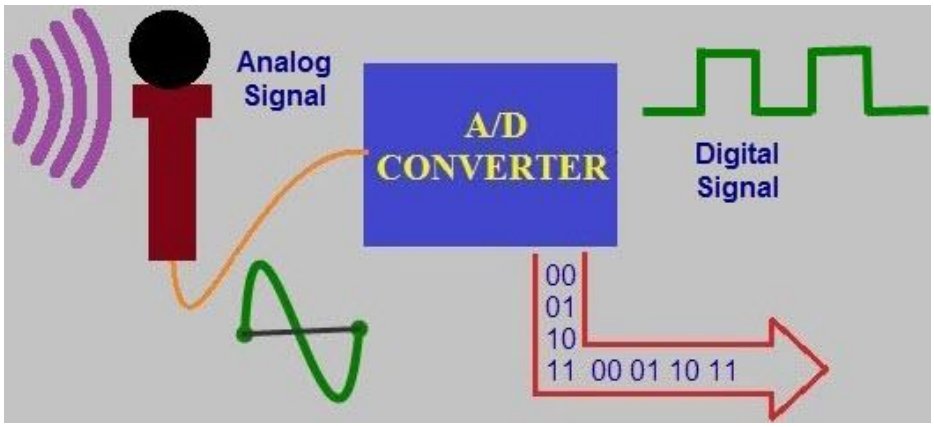
Data Bus: It is used for only transfer or receive the data.

Address Bus: Address bus is used to transmit the memory address from the peripherals to the CPU. I/O pins are used to interface the external peripherals; UART and USART both are serial communication protocols which are used for interfacing serial devices like GSM, GPS, Bluetooth, IR , etc.



A/D converters

The main intention of this analog to digital converter is to convert analog voltage values to digital voltage values. A/D module of PIC microcontroller consists of 5 inputs for 28 pin devices and 8 inputs for 40 pin devices. The operation of the analog to digital converter is controlled by ADCON0 and ADCON1 special registers. The upper bits of the converter are stored in register ADRESH and lower bits of the converter are stored in register ADRESL. For this operation, it requires 5V of an analog reference voltage.



A/D CONVERTER

Timers/ Counters

PIC microcontroller has four timer/counters wherein the one 8-bit timer and the remaining timers have the choice to select 8 or 16-bit mode. Timers are used for generating accuracy actions, for example, creating specific time delays between two operations.

Interrupts

PIC microcontroller consists of 20 internal interrupts and three external interrupt sources which are associated with different peripherals like ADC, USART, Timers, and so on.

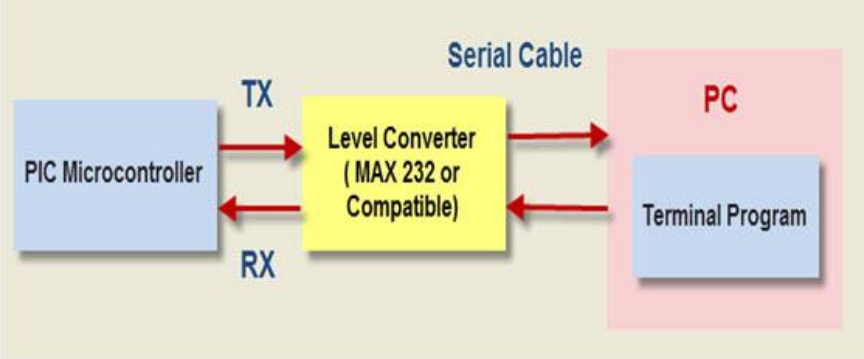
Serial Communication

- **USART:** The name USART stands for Universal synchronous and Asynchronous Receiver and Transmitter which is a serial communication for two protocols. It is used

for transmitting and receiving the data bit by bit over a single wire with respect to clock pulses. The PIC microcontroller has two pins TXD and RXD. These pins are used for transmitting and receiving the data serially.

- **SPI Protocol:** The term SPI stands for Serial Peripheral Interface. This protocol is used to send data between PIC microcontroller and other peripherals such as SD cards, sensors and shift registers. PIC microcontroller support three wire SPI communications between two devices on a common clock source. The data rate of SPI protocol is more than that of the USART.

- **I2C Protocol:** The term I2C stands for Inter Integrated Circuit , and it is a serial protocol which is used to connect low speed devices such as EEPROMS, microcontrollers, A/D converters, etc. PIC microcontroller support two wire Interface or I2C communication between two devices which can work as both Master and Slave device.



Serial Communication

Oscillators

Oscillators are used for timing generation. Pic microcontroller consist of external oscillators like RC oscillators or crystal oscillators. Where the crystal oscillator is connected between the two oscillator pins. The value of the capacitor is connected to every pin that decides the mode of the operation of the oscillator. The modes are crystal mode, high-speed mode and the low-power mode. In case of RC oscillators, the value of the resistor & capacitor determine the clock frequency and the range of clock frequency is 30KHz to 4MHz.

CCP module

The name CCP module stands for capture/compare/PWM where it works in three modes such as capture mode, compare mode and PWM mode.

- **Capture Mode:** Capture mode captures the time of arrival of a signal, or in other words, when the CCP pin goes high, it captures the value of the Timer1.
- **Compare Mode:** Compare mode acts as an analog comparator. When the timer1 value reaches a certain reference value, then it generates an output.
- **PWM Mode:** PWM mode provides pulse width modulated output with a 10-bit resolution and programmable duty cycle.

PIC Microcontroller Applications

The PIC microcontroller projects can be used in different applications, such as peripherals, audio accessories, video games, etc. For better understanding of this PIC microcontroller, the following project demonstrates PIC microcontroller's operations.

Street Light that Glows on Detecting Vehicle Movement:

The main intention of this project is to detect the movement of vehicles on highways to switch on a block of street lights ahead of it, and also switch off the trailing lights to conserve energy. In this project, a PIC microcontroller is done by using assembly language or embedded C.

The power supply gives the power to the total circuit by stepping down, rectifying, filtering and regulating AC mains supply. When there are no vehicles on highway, then all lights will turn OFF so that the power can be conserved. The IR sensors are placed on the road to sense the vehicle movement. When there are vehicles on highway, then the IR sensor senses the vehicle movement immediately, it sends the commands to the PIC microcontroller to switch ON/OFF the LEDs. A bunch of LEDS will be turned on when a vehicle come near to the sensor and once the vehicle passes away from the sensor the intensity will become lower than the LEDs will turn OFF

Advantages of PIC Microcontroller:

- PIC microcontrollers are consistent a**nd** faulty of PIC percentage is very less. The performance of the PIC microcontroller is very fast because of using RISC architecture.
- When comparing to other microcontrollers, power consumption is very less and programming is also very easy.
- Interfacing of an analog device is easy without any extra circuitry

Disadvantages of PIC Microcontroller:

- The length of the program is high due to using RISC architecture (35 instructions)
- One single accumulator is present and program memory is not accessible

Reset Function

Reset function is one of the most advanced features that is available on all modern microcontrollers. The PIC16F8xx series have various kinds of resets. The various kinds of reset options that are available on PIC 16F877 series are given below.

• Power-on Reset (POR).

• MCLR Reset during normal operation.

• MCLR Reset during Sleep.

• WDT Reset (during normal operation).

• WDT Wake-up (during Sleep).

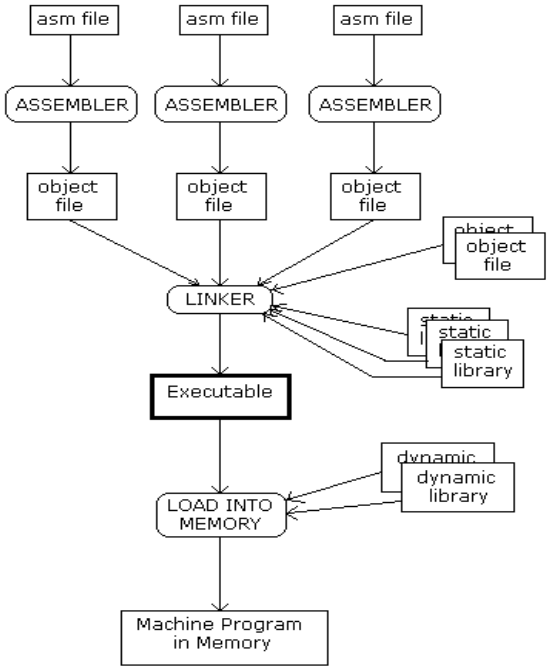
• Brown-out Reset (BOR).

A simplified block diagram of the on-chip Reset circuit is shown in the figure below. his document contains very brief examples of assembly language programs for the x86. The topic of x86 assembly language programming is messy because:

- There are many different assemblers out there: MASM, NASM, gas, as86, TASM, a86, Terse, etc. All use radically different assembly languages.
- There are differences in the way you have to code for Linux, OS/X, Windows, etc.
- Many different object file formats exist: ELF, COFF, Win32, OMF, a.out for Linux, a.out for FreeBSD, rdf, IEEE-695, as86, etc.
- You generally will be calling functions residing in the operating system or other libraries so you will have to know some technical details about how libraries are linked, and not all linkers work the same way.
- Modern x86 processors run in either 32 or 64-bit mode; there are quite a few differences between these.

We'll give examples written for NASM, MASM and gas for both Win32 and Linux. We will even include a section on DOS assembly language programs for historical interest. These notes are not intended to be a substitute for the documentation that accompanies the processor and the assemblers, nor is it intended to teach you assembly language. Its only purpose is to show how to assemble and link programs using different assemblers and linkers.

Assembler linkers



Each assembly language file is assembled into an "object file" and the object files are linked with other object files to form an executable. A "static library" is really nothing more than a collection of (probably related) object files. plication programmers generally make use of libraries for things like I/O and math.

Assemblers you should know about include

- **MASM**, the Microsoft Assembler. It outputs OMF files (but Microsoft's linker can convert them to win32 format). It supports a massive and clunky assembly language. **Memory addressing is not intuitive.** The directives required to set up a program make programming unpleasant.
- **GAS**, the GNU assember. This uses the rather ugly AT&T-style syntax so many people do not like it; however, you can configure it to use and understand the Intel-style. It was designed to be part of the back end of the GNU compiler collection (gcc).
- **NASM**, the "Netwide Assembler." It is free, small, and best of all it can output zillions of different types of object files. The language is much more sensible than MASM in many respects.

There are many object file formats. Some you should know about include

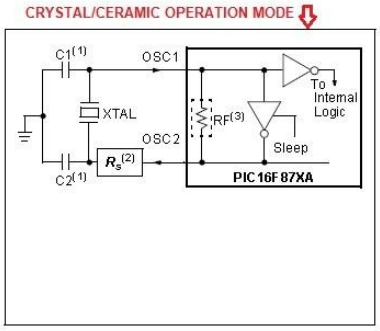
- **OMF**: used in DOS but has 32-bit extensions for Windows. Old.
- **AOUT**: used in early Linux and BSD variants
- **COFF**: "Common object file format"
- **Win, Win32**: Microsoft's version of COFF, not exactly the same! Replaces OMF.
- **Win64**: Microsoft's format for Win64.
- **ELF, ELF32**: Used in modern 32-bit Linux and elsewhere
- **ELF64**: Used in 64-bit Linux and elsewhere
- **macho32**: NeXTstep/OpenStep/Rhapsody/Darwin/OS X 32-bit
- **macho64**: NeXTstep/OpenStep/Rhapsody/Darwin/OS X 64-bit

Oscillator Selection Function

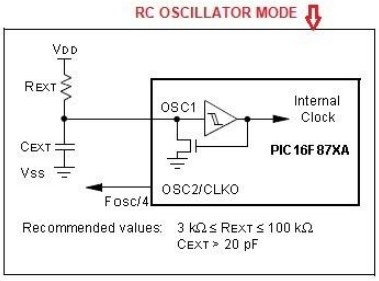
The PIC16F8xx series basically supports different types of oscillators and also PIC16F87XA devices. It also has a Watchdog Timer which can be shut-off only through configuration bits. It runs off its own RC oscillator for added reliability (Configurations as compared to normal microcontrollers/processors). The different oscillator modes can be easily selected by the user. The user can program two configuration bits (foscillator1 and foscillator0) to selection of the basic four modes. The basic oscillator modes and the typical values used for these oscillators are given in the picture below.

- LP Low-Power Crystal
- XT Crystal/Resonator
- HS High-Speed Crystal/Resonator
- RC resistor/capacitor oscillator

CRYSTAL/CERAMIC OPERATION MODE



RC OSCILLATOR MODE



CAPACITOR SELECTION FOR RC OSC.

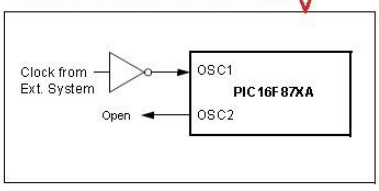
Osc Type	Crystal Freq.	Cap. Range C1	Cap. Range C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF

These values are for design guidance only.
See notes following this table.

Crystals Used

32 kHz	Epson C-001R32.768K-A	± 20 PPM
200 kHz	STD XTL 200.000KHz	± 20 PPM
1 MHz	ECS ECS-10-13-1	± 50 PPM
4 MHz	ECS ECS-40-20-1	± 50 PPM
8 MHz	EPSON CA-301 8.000M-C	± 30 PPM
20 MHz	EPSON CA-301 20.000M-C	± 30 PPM

EXTERNAL HP/LP OPERATION MODES



TYPICAL CERAMIC RESONATOR VALUES

Ranges Tested:			
Mode	**Freq.**	**OSC1**	**OSC2**
XT	455 kHz 2.0 MHz 4.0 MHz	68-100 pF 15-68 pF 15-68 pF	68-100 pF 15-68 pF 15-68 pF
HS	8.0 MHz 16.0 MHz	10-68 pF 10-22 pF	10-68 pF 10-22 pF

FOR DESIGN USE ONLY

Resonators Used:		
2.0 MHz	Murata Erie CSA2.00MG	± 0.5%
4.0 MHz	Murata Erie CSA4.00MG	± 0.5%
8.0 MHz	Murata Erie CSA8.00MT	± 0.5%
16.0 MHz	Murata Erie CSA16.00MX	± 0.5%

All resonators used did not have built-in capacitors.

UNIT-III -PIC PROGRAMMNG

INTERRUPTS

Interrupt is the signal which is sent to the microcontroller to mark the event that requires immediate attention. This signal **requests** the microcontroller to stop to perform the current program temporarily **time** to execute a special code. It means when external device finishes the task imposed on it, the microcontroller will be notified that it can access and receive the information and use it. Interrupts are just like waiting for the phone to ring.

INTERRUPT SOURCES in microcontrollers

The request to the microcontroller to stop to perform the current program temporarily can come from various sources:

- Through external hardware devices like pressing specific key on the keyboard, which sends Interrupt to the microcontroller to read the information of the pressed key.
- During execution of the program, the microcontroller can also send interrupts to itself to report an error in the code. For example, division by 0 will causes an interrupt.
- In the multi-processor system, the microcontrollers can send interrupts to each other to communicate. For example, to divide the work between them they will send signals between them.

INTERRUPT TYPES in PIC microcontrollers

There are 2 types of interrupts for PIC microcontroller that can cause break.

Software Interrupt: It comes from a program that is executed by microcontroller or we can say that it is generated by internal peripherals of the microcontroller and requests the processor to hold the running of program and go to make an interrupt.

Hardware Interrupt: These interrupts are sent by external hardware devices at certain pins of microcontroller.

External interrupt of PIC18F452 microcontroller

External INTERRUPT IN PIC18F452: Sometimes External devices are connected with **microcontroller**. If that external device has to send some information to microcontroller, then microcontroller needs to know about this situation to get that information. An example of such external device is the **digital thermometer**. It measures the **temperature** and at the end of measurements transmits results to the microcontroller. Now the purpose of this article to explain the fact that how does the microcontroller knows to get the required information from external device.

Types of interrupts

There are two methods of communication between the microcontroller and the external device:

- By using Polling
- By using Interrupts

POLLING

In this method, the external devices are not independent. We fix the time interval in which microcontroller has to contact the external device. The microcontroller accesses that device at the exact time interval and gets the required information. Polling method is just like picking up our phone after every few seconds to see if we have a call.The main drawback of this method is the waste of time of microcontroller. It needs to wait and check whether the new information has arrived not.

REGISTER CONFIGURATION for external interrupt

These are the registers for interrupt operation and minimum 1 register can be used to control the interrupt operation in PIC18F452 which are:

- RCON (Reset Control Register)
- INTCON, INTCON2, INTCON3 (Interrupt Control Registers)
- PIR1, PIR2 (Peripheral Interrupt Request Registers)
- PIE1, PIE2 (Peripheral Interrupt Enable Registers)

RCON Register:

- Reset control register
- IPEN bit to enable interrupt priority scheme, 1= enable priority level on interrupts
- Other bits used to indicate the cause of reset
  RI (Reset Instruction flag), TO (Watchdog Time Out flag), PD (Power on Detection flag), POR (Power on Reset status) and BOR (Brown Out Reset status bit)

INTCON Register:

- 3 Interrupt control registers INTCON1, INTCON2, INTCON3
- Readable and writable register which contains various enable and flag bits
- Interrupt flag bits get set when an interrupt condition occurs
- Contain enable, priority and flag bits for external interrupt, port B pin change and TMR0 overflow interrupt

PIE Register:

- Peripheral Interrupt Enable register
- May be multiple register (PIE1, PIE2), depending on the number of peripheral interrupt sources
- Contain the individual bits to enable/disable Peripheral interrupts for use

PIR Register:

- Peripheral Interrupt Flag register
- May be multiple register (PIR1, PIR2), depending on the number of peripheral interrupt sources
- Contain bits to identify which interrupt occurs (flags)
- Corresponding bits are set when the interrupt occurred

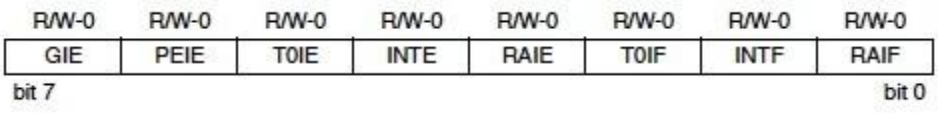
EXTERNAL INTERRUPT registers setting

INTCON registers are just used to configure the external PIC interrupts. This article also deals with external interrupts of PIC18F452 so we will discuss it in detail here..

FOR PERIPHERAL INTERRUPT:

The PIE (Peripheral Interrupt Enable) and PIR (Peripheral Interrupt Request) registers are used to configure the Peripheral (Internal) Interrupts.

INTCON **REGISTER:**



GIE: Global Interrupt Enable

This bit is set high to enable all interrupts of PIC18F452.

1 = Enable all interrupts

0 = Disable all interrupts

PEIE: Peripheral Interrupt Enable

This bit is set high to enable all the peripheral interrupts (Internal interrupts) of the microcontroller.

1 = Enable all peripheral interrupts

0 = Disable all peripheral interrupts

T0IE: TMR0 Overflow Interrupt Enable

This bit is set high to enable the External Interrupt 0.

1 = Enable TMR0 overflow interrupt

0 = Disable TMR0 overflow interrupt

INTE: INT External Interrupt Enable

This bit is set high to enable the external interrupts.

1 = Enables the INT external interrupt

0 = Disables the INT external interrupt

RBIE: RB Interrupt Enable

This bit is set high to enable the RB Port Change interrupt pin.

1 = Enables the RB port change interrupt

0 = Disables the RB port change interrupt

T0IF: TMR0 Overflow Interrupt Flag

1 = TMR0 register has overflowed (it must be cleared in software)

0 = TMR0 register has not overflowed

INTF: INT External Interrupt Flag

1 = The INT external interrupt occurred (it must be cleared in software)

0 = The INT external interrupt did not occur

RBIF: RB Port Change Interrupt Flag

1 = At least one of the RB7:RB4 pins changed the state (must be cleared in software)

0 = None of RB7:RB4 pins have changed the state

INTCON2 REGISTER:

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
RBPU	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
bit 7							bit 0

RBPU: Port B Pull up Enable

1 = All port B pull ups are disabled

0 = All port B pull ups are enabled by individual port latch values

INTEDG0, INTEDG1, INTEDG2: External Interrupt Edge select

These bits are used to select the triggering edge of the corresponding interrupt signal on which the microcontroller is to respond.

1 = Interrupt on rising edge

0 = Interrupt on falling edge

Bit 3, Bit 1: Unimplemented

It is always read as 0

TMR0IP: TMR0 overflow priority

1 = High priority

0 = Low priority

RBIP: RB Port change Interrupt Priority

1 = High priority

0 = Low priority

INTCON3 **REGISTER:**

R/W-1	R/W-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
bit 7							bit 0

INT1IP, INT2IP:

These bits are used to set priority of the interrupts 1 and 2, respectively.

1 = High priority

0 = Low priority

Bit 5, Bit 2:

Both are unimplemented and read as 0.

INT1IE, INT2IE:

These bits enable/disable the External Interrupt 1 and 2, respectively.

1 = Enables the External Interrupt x

0 = Disables the External Interrupt x

INT1IF, INT2IF:

These are External Interrupt 1 and 2 flag bits, respectively.

1 = The INTx External Interrupt occurred (must be cleared in software)

0 = The INTx External Interrupt did not occur

TIMERS

The timers of the PIC16F887 microcontroller can be briefly described in only one sentence. There are three completely independent timers/counters marked as TMR0, TMR1 and TMR2. But it's not as simple as that.

Timer TMR0

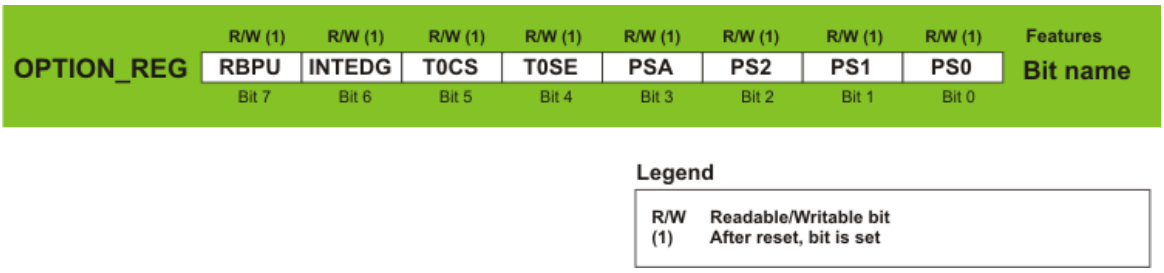
The timer TMR0 has a wide range of applications in practice. Very few programs don't use it in some way. It is very convenient and easy to use for writing programs or subroutines for generating pulses of arbitrary duration, time measurement or counting external pulses (events) with almost no limitations

The timer TMR0 module is an 8-bit timer/counter with the following features:

- 8-bit timer/counter;
- 8-bit prescaler (shared with Watchdog timer);
- Programmable internal or external clock source;
- Interrupt on overflow; and
- Programmable external clock edge selection.

Figure 4-1 below represents the timer TMR0 schematic with all bits which determine its operation. These bits are stored in the OPTION_REG Register.

OPTION_REG Register



- **RBPU – PORTB Pull-up enable bit**
  - 1 – PORTB pull-up resistors are disabled; and
  - 0 – PORTB pins can be connected to pull-up resistors.
- **INTEDG – Interrupt Edge Select bit**
  - 1 – Interrupt on rising edge of INT pin (0-1); and
  - 0 – Interrupt on falling edge of INT pin (1-0).
- **T0CS – TMR0 Clock Select bit**

- 1 – Pulses are brought to TMR0 timer/counter input through the RA4 pin; and
- 0 – Internal cycle clock (Fosc/4).

- **T0SE – TMR0 Source Edge Select bit**
  - 1 – Increment on high-to-low transition on TMR0 pin; and
  - 0 – Increment on low-to-high transition on TMR0 pin.
- **PSA – Prescaler Assignment bit**
  - 1 – Prescaler is assigned to the WDT; and
  - 0 – Prescaler is assigned to the TMR0 timer/counter.
- **PS2, PS1, PS0 – Prescaler Rate Select bit**
  - Prescaler rate is adjusted by combining these bits

    As seen in the table 4-1, the same combination of bits gives different prescaler rate for the timer/counter and watch-dog timer respectively.

PS2	PS1	PS0	TMR0	WDT
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128

The function of the PSA bit is shown in the two figures below:

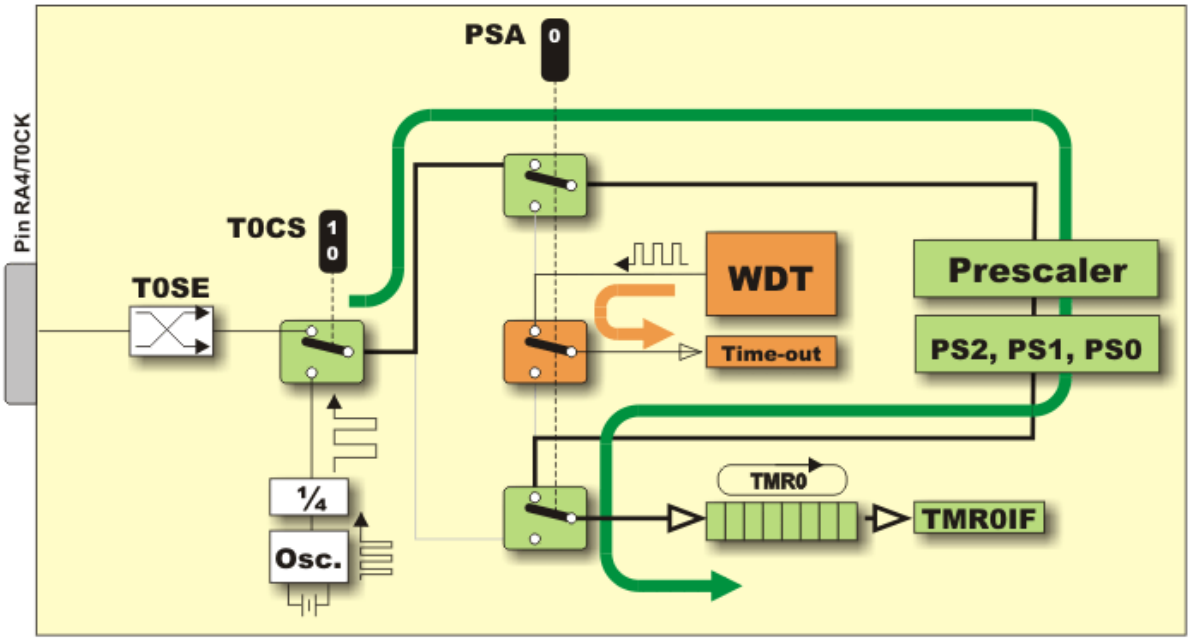


Fig. The function of the PSA bit 0

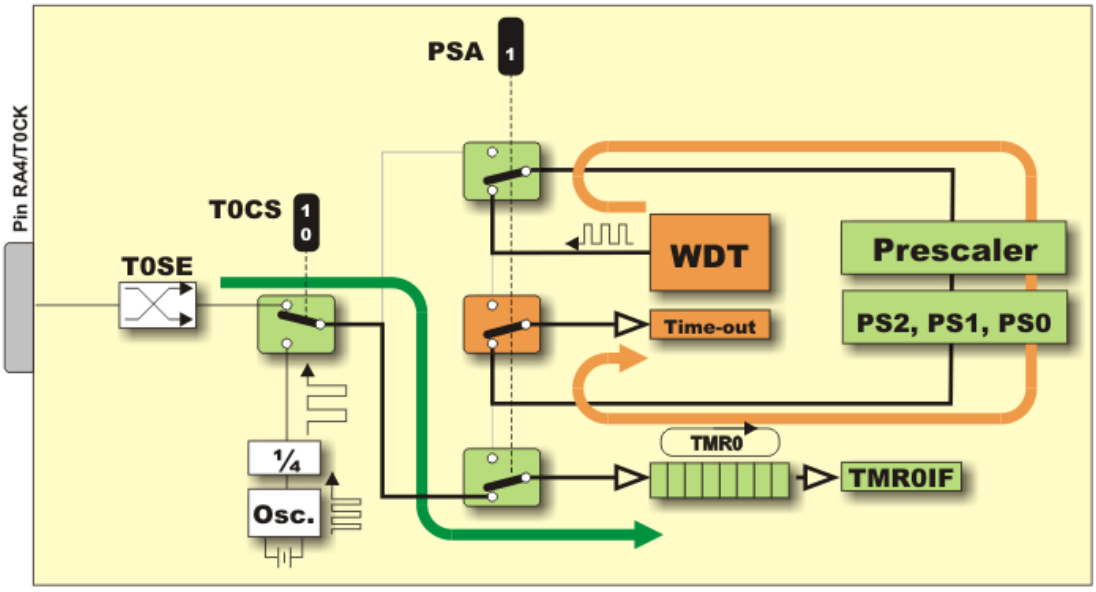


Fig. The function of the PSA bit 1

As seen, the logic state of the PSA bit determines whether the prescaler is to be assigned to the timer/counter or watch-dog timer.

Additionally it is also worth mentioning:

- When the prescaler is assigned to the timer/counter, any write to the TMR0 register will clear the prescaler;
- When the prescaler is assigned to watch-dog timer, a CLRWDT instruction will clear both the prescaler and WDT;
- Writing to the TMR0 register used as a timer, will not cause the pulse counting to start immediately, but with two instruction cycles delay. Accordingly, it is necessary to adjust the value written to the TMR0 register;
- When the microcontroller is setup in *sleep* mode, the oscillator is turned off. Overflow cannot occur since there are no pulses to count. This is why the TMR0 overflow interrupt cannot wake up the processor from Sleep mode;
- When used as an external clock counter without prescaler, a minimal pulse length or a pause between two pulses must be 2 Tosc + 20 nS. Tosc is the oscillator signal period;
- When used as an external clock counter with prescaler, a minimal pulse length or a pause between two pulses is 10nS;
- The 8-bit prescaler register is not available to the user, which means that it cannot be directly read or written to;
- When changing the prescaler assignment from TMR0 to the watch-dog timer, the following instruction sequence must be executed in order to avoid reset:

```
1       BANKSEL TMR0
2       CLRWDT              ;CLEAR WDT
3       CLRF    TMR0        ;CLEAR TMR0 AND PRESCALER
4       BANKSEL OPTION_REG
5       BSF     OPTION_REG,PSA ;PRESCALER IS ASSIGNED TO THE WDT
6       CLRWDT              ;CLEAR WDT
7       MOVLW   b’11111000’   ;SELECT BITS PS2,PS1,PS0 AND CLEAR
8       ANDWF   OPTION_REG,W  ;THEM BY INSTRUCTION “LOGICAL AND”
```

```
9       IORLW   b'00000101'   ;BITS PS2, PS1, AND PS0 SET
10      MOVWF   OPTION_REG    ;PRESCALER RATE TO 1:32
```

To select mode:

- Timer mode is selected by the T0CS bit of the OPTION_REG register, (T0CS: 0=timer, 1=counter);
- When used, the prescaler should be assigned to the timer/counter by clearing the PSA bit of the OPTION_REG register. The prescaler rate is set by using the PS2-PS0 bits of the same register; and
- When using interrupt, the GIE and TMR0IE bits of the INTCON register should be set.

To measure time:

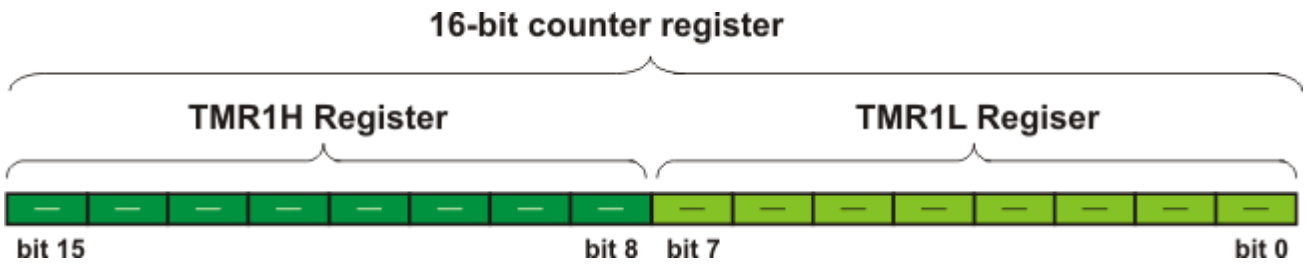
- Reset the TMR0 register or write some well-known value to it;
- Elapsed time (in microseconds when using quartz 4MHz) is measured by reading the TMR0 register; and
- The flag bit TMR0IF of the INTCON register is automatically set every time the TMR0 register overflows. If enabled, an interrupt occurs.

To count pulses:

- The polarity of pulses are to be counted is selected on the RA4 pin are selected by the TOSE bit of the OPTION register (T0SE: 0=positive, 1=negative pulses); and
- Number of pulses may be read from the TMR0 register. The prescaler and interrupt are used in the same manner as in timer mode.

Timer TMR1

Timer TMR1 module is a 16-bit timer/counter, which means that it consists of two registers (TMR1L and TMR1H). It can count up 65.535 pulses in a single cycle, i.e. before the counting starts from zero.

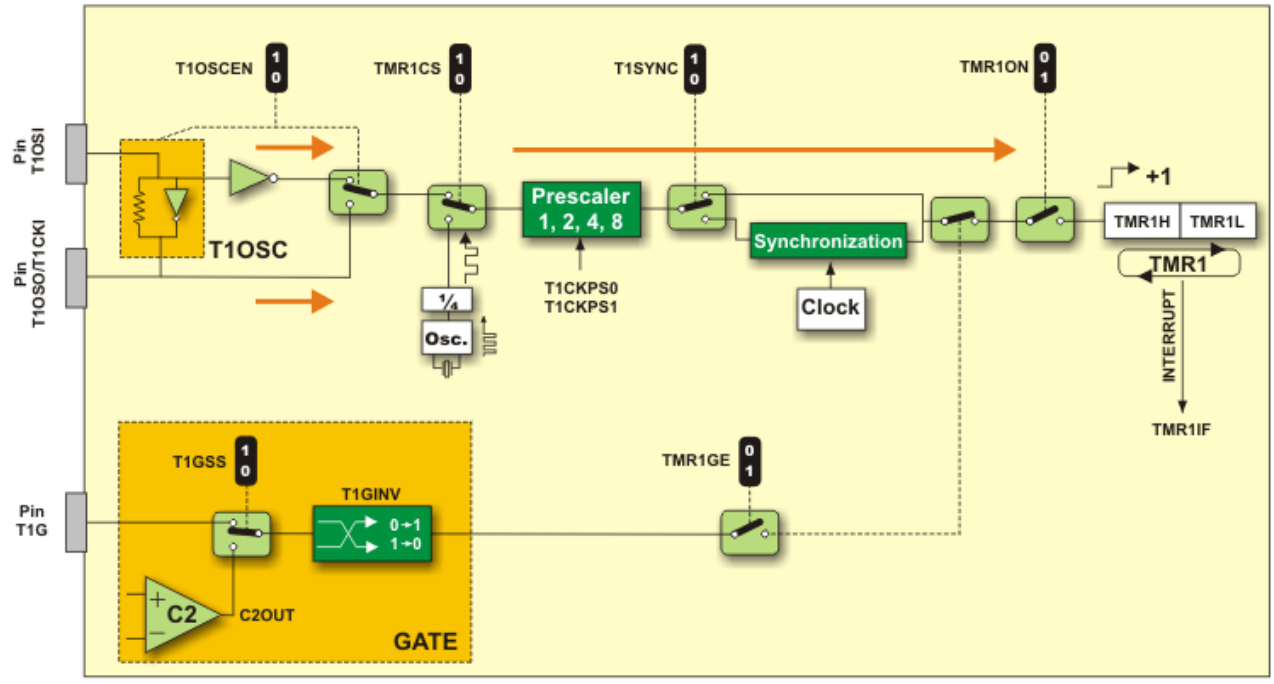


Timer TMR1

Similar to the timer TMR0, these registers can be read or written to at any moment. In case an overflow occurs, an interrupt is generated.

The timer TMR1 module may operate in one of two basic modes- as a timer or a counter. However, unlike the timer TMR0, each of these modules has additional functions.

Parts of the T1CON register are in control of the operation of the timer TMR1.



Timer TMR1 Overview

Timer TMR1 Prescaler

Timer TMR1 has a completely separate prescaler which allows 1, 2, 4 or 8 divisions of the clock input. The prescaler is not directly readable or writable. However, the prescaler counter is automatically cleared upon write to the TMR1H or TMR1L register.

Timer TMR1 Oscillator

RC0/T1OSO and RC1/T1OSI pins are used to register pulses coming from peripheral electronics, but they also have an additional function. As seen in figure 4-7, they are simultaneously configured as both input (pin RC1) and output (pin RC0) of the additional LP quartz oscillator (low power).

This additional circuit is primarily designed for operating at low frequencies (up to 200 KHz), more precisely, for using the 32,768 KHz quartz crystal. Such crystals are used in quartz watches because it is easy to obtain one-second-long pulses by simply dividing this frequency.

Since this oscillator does not depend on internal clocking, it can operate even in *sleep* mode. It is enabled by setting the T1OSCEN control bit of the T1CON register. The user must provide a software time delay (a few milliseconds) to ensure proper oscillator start-up.

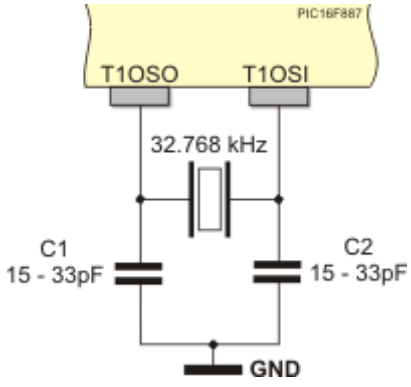


Table below shows the recommended values of capacitors to suit the quartz oscillator. These values do not have to be exact. However, the general rule is: the higher the capacitor's capacity the higher the stability, which, at the same time, prolongs the time needed for the oscillator stability.

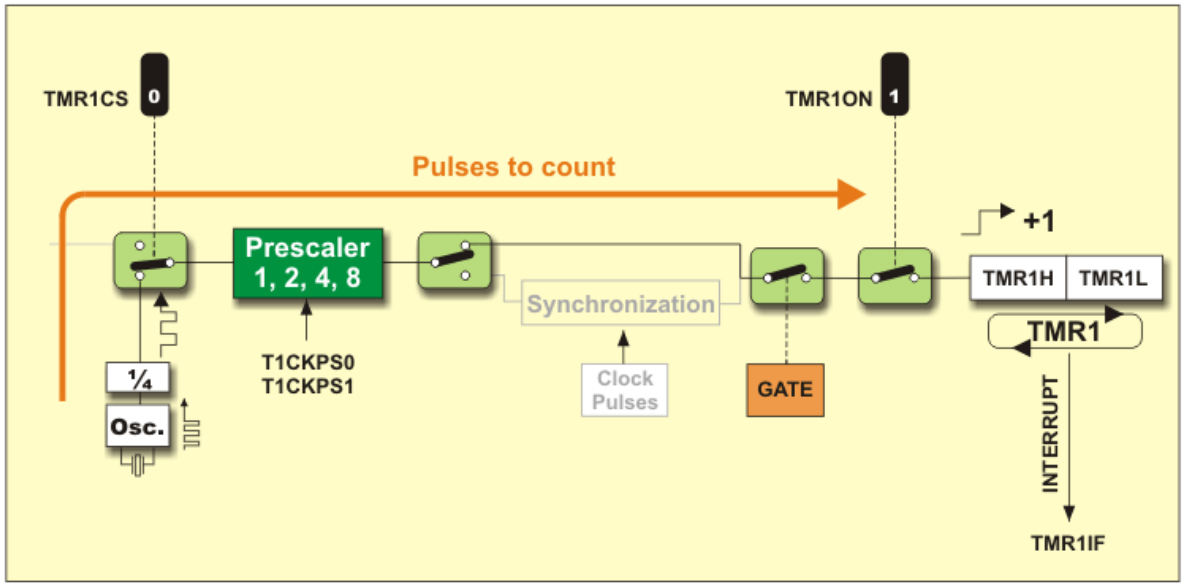
Timer TMR1 Gate

Timer 1 gate source is software configurable to be the T1G pin or the output of comparator C2. This gate allows the timer to directly time external events using the logic state on the T1G pin or analog events using the comparator C2 output. Refer to figure 4-7 above. In order to time a signals duration it is sufficient to enable such gate and count pulses having passed through it.

TMR1 in timer mode

In order to select this mode, it is necessary to clear the TMR1CS bit. After this, the 16-bit register will be incremented on every pulse coming from the internal oscillator. If the 4MHz quartz crystal is in use, it will be incremented every microsecond.

In this mode, the T1SYNC bit does not affect the timer because it counts internal clock pulses. Since the whole electronics uses these pulses, there is no need for synchronization.

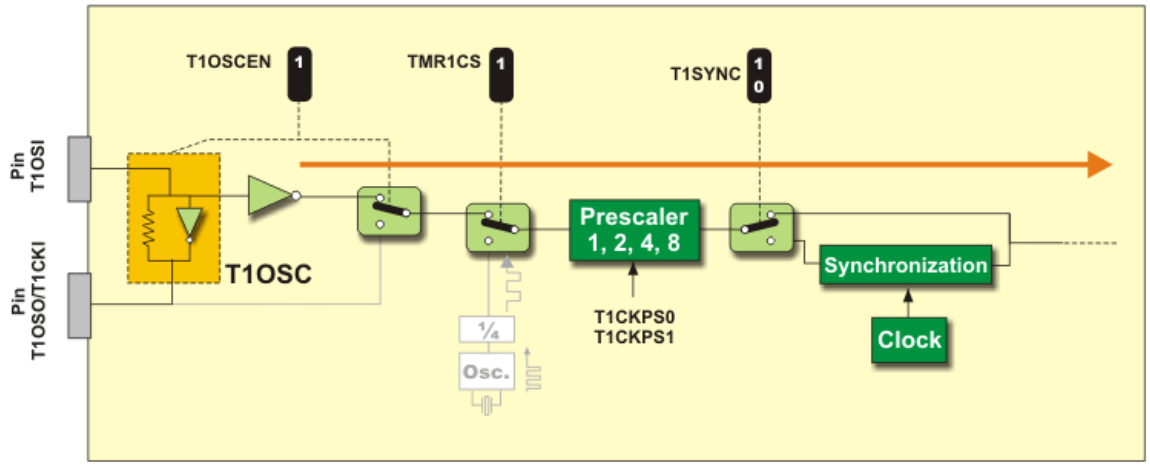


TMR1 in timer mode

The microcontroller's clock oscillator does not run during sleep mode so the timer register overflow cannot cause any interrupt.

Timer TMR1 Oscillator

The power consumption of the microcontroller is reduced to the lowest level in *Sleep* mode. The point is to stop the oscillator. Anyway, it is easy to set the timer in this mode- by writing a SLEEP instruction to the program. A problem occurs when it is necessary to wake up the microcontroller because only an interrupt can do that. Since the microcontroller "sleeps", an interrupt must be triggered by external electronics. It can all get incredibly complicated if it is necessary the 'wake up' occurs at regular time intervals…



Timer TMR1 Oscillator

In order to solve this problem, a completely independent *Low Power* quartz oscillator, able to operate in *sleep mode*, is built into the PIC16F887 microcontroller. Simply, what previously has been a separate circuit, it is now built into the microcontroller and assigned to the timer TMR1. The oscillator is enabled by setting the T1OSCEN bit of the T1CON register. After that, the TMR1CS bit of the same register then is used to determine that the timer TMR1 uses pulse sequences from that oscillator.

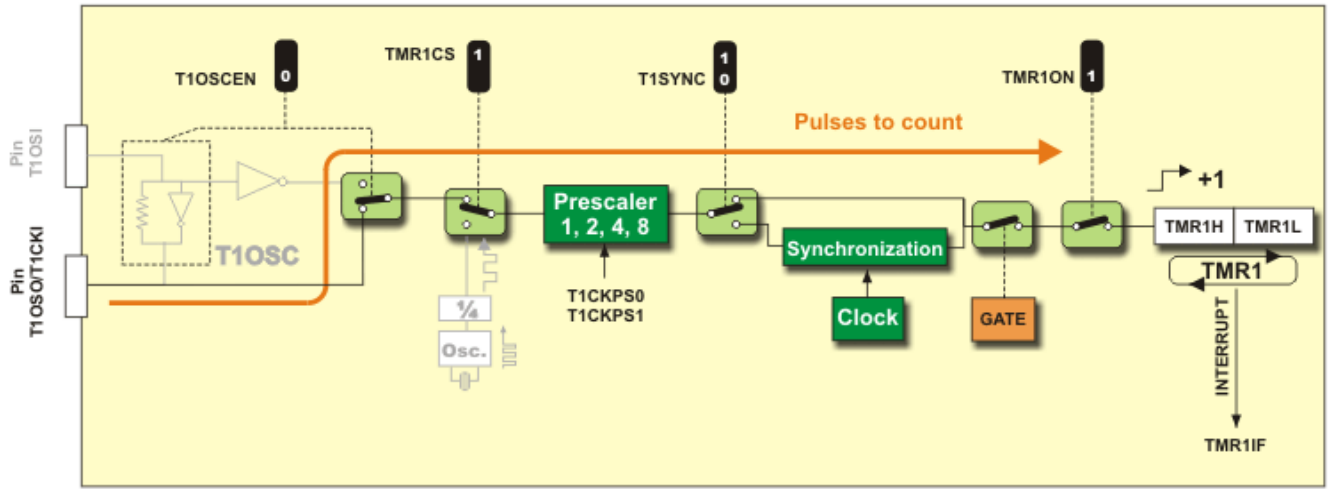
- The signal from this quartz oscillator is synchronized with the microcontroller clock by clearing the T1SYNC bit. In that case, the timer cannot operate in *sleep* mode. You wonder why? Because the circuit for synchronization uses the clock of microcontroller!; and

- The TMR1 register overflow interrupt may be enabled. Such interrupts will occur in *sleep* mode as well.

TMR1 in counter mode

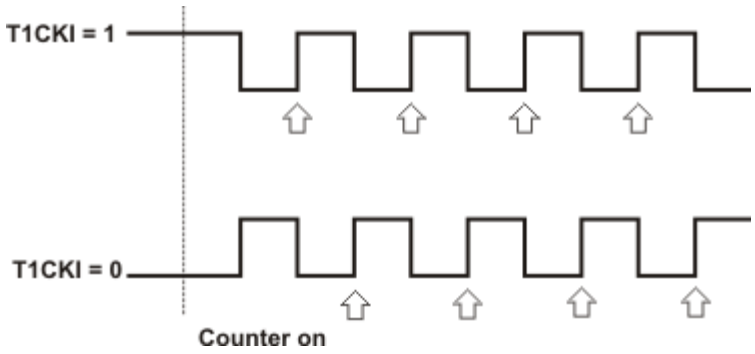
Timer TMR1 starts to operate as a counter by setting the TMR1CS bit. It means that the timer TMR1 is incremented on the rising edge of the external clock input T1CKI. If control bit T1SYNC of the T1CON register is cleared, the external clock inputs will be synchronized on their way to the TMR1 register. In other words, the timer TMR1 is synchronized to the microcontroller system clock and called a synchronous counter.

When the microcontroller ,operating in this way, is set in *sleep* mode, the TMR1H and TMR1L timer registers are not incremented even though clock pulses appear on the input pins. Simply, since the microcontroller system clock does not run in this mode, there are no clock inputs to use for synchronization. However, the prescaler will continue to run if there are clock pulses on the pins since it is just a simple frequency divider.



Counter Mode

This counter registers a logic one (1) on input pins. It is important to understand that at least one falling edge must be registered prior to the first increment on rising edge. Refer to figure on the left. The arrows in figure 4-11 denote counter increments.



T1CON Register

	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	Features
T1CON	T1GINV	TMR1GE	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend	
R/W	Readable/Writable bits
(0)	After reset, bit is cleared

T1GINV – Timer1 Gate Invert bit acts as logic state inverter on the T1G pin gate or the comparator C2 output (C2OUT) gate. It enables the timer to mea sure time whilst the gate is high or low.

- 1 – Timer 1 counts when the pin T1G or bit C2OUT gate is high (1); and
- 0 – Timer 1 counts when the pin T1G or bit C2OUT gate is low (0).

TMR1GE – Timer1 Gate Enable bit determines whether the pin T1G or comparator C2 output (C2OUT) gate will be active or not. This bit is functional only in the event that the timer TMR1 is on (bit TMR1ON = 1). Otherwise, this bit is ignored.

- 1 Timer TMR1 is on only if timer 1 gate is not active; and
- 0 Gate does not affect the timer TMR1.

T1OSCEN – LP Oscillator Enable Control bit

- 1 – LP oscillator is enabled for timer TMR1 clock (oscillator with low power consumption and frequency 32.768 kHz); and
- 0 – LP oscillator is off.

T1SYNC – Timer1 External Clock Input Synchronization Control bit enables synchronization of the LP oscillator input or T1CKI pin input with the microcontroller internal clock. When counting pulses from the local clock source (bit TMR1CS = 0), this bit is ignored.

- 1 – Do not synchronize external clock input; and
- 0 – Synchronize external clock input.

TMR1CS – Timer TMR1 Clock Source Select bit

- 1 – Counts pulses on the T1CKI pin (on the rising edge 0-1); and
- 0 – Counts pulses of the internal clock of microcontroller.

TMR1ON – Timer1 On bit

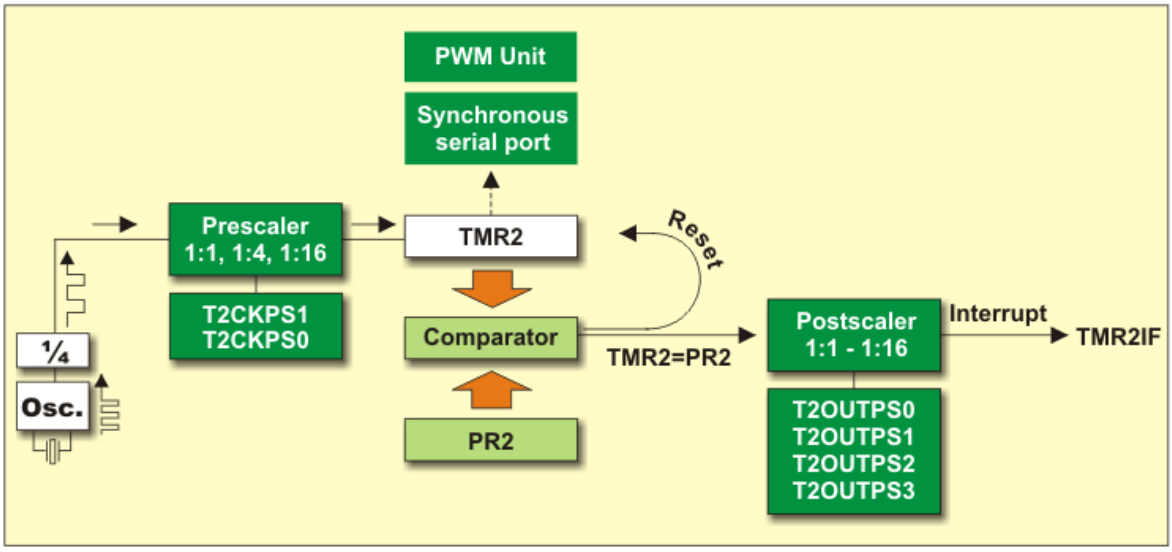
- 1 – Enables Timer TMR1; and
- 0 – Stops Timer TMR1.

In order to use the timer TMR1 properly, it is necessary to perform the following:

- Since it is not possible to turn off the prescaler, its rate should be adjusted by using bits T1CKPS1 and T1CKPS0 of the register T1CON (Refer to table 4-2);
- The mode should be selected by the TMR1CS bit of the same register (TMR1CS: 0= the clock source is quartz oscillator, 1= the clock source is supplied externally);
- By setting the T1OSCEN bit of the same register, the timer TMR1 is turned on and the TMR1H and TMR1L registers are incremented on every clock input. Counting stops by clearing this bit;
- The prescaler is cleared by clearing or writing the counter registers; and
- By filling both timer registers, the flag TMR1IF is set and counting starts from zero.

Timer TMR2

Timer TMR2 module is an 8-bit timer which operates in a very specific way.



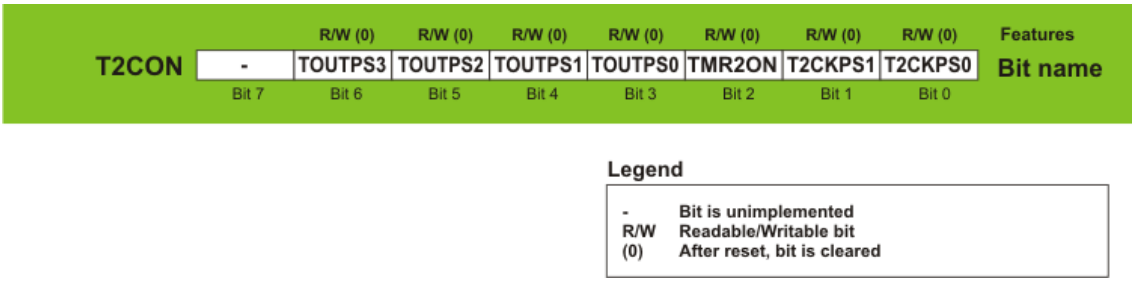
The pulses from the quartz oscillator first pass through the prescaler whose rate may be changed by combining the T2CKPS1 and T2CKPS0 bits. The output of the prescaler is then used to increment the TMR2 register starting from 00h. The values of TMR2 and PR2 are constantly compared and the TMR2 register keeps on being incremented until it matches the value in PR2. When a match occurs, the TMR2 register is automatically cleared to 00h. The timer TMR2 Postscaler is incremented and its output is used to generate an interrupt if it is enabled.

The TMR2 and PR2 registers are both fully readable and writable. Counting may be stopped by clearing the TMR2ON bit, which contributes to power saving.

As a special option, the moment of TMR2 reset may be also used to determine synchronous serial communication baud rate.

The timer TMR2 is controlled by several bits of the T2CON register.

T2CON Register



TMR2ON – Timer2 On bit turns the timer TMR2 on.

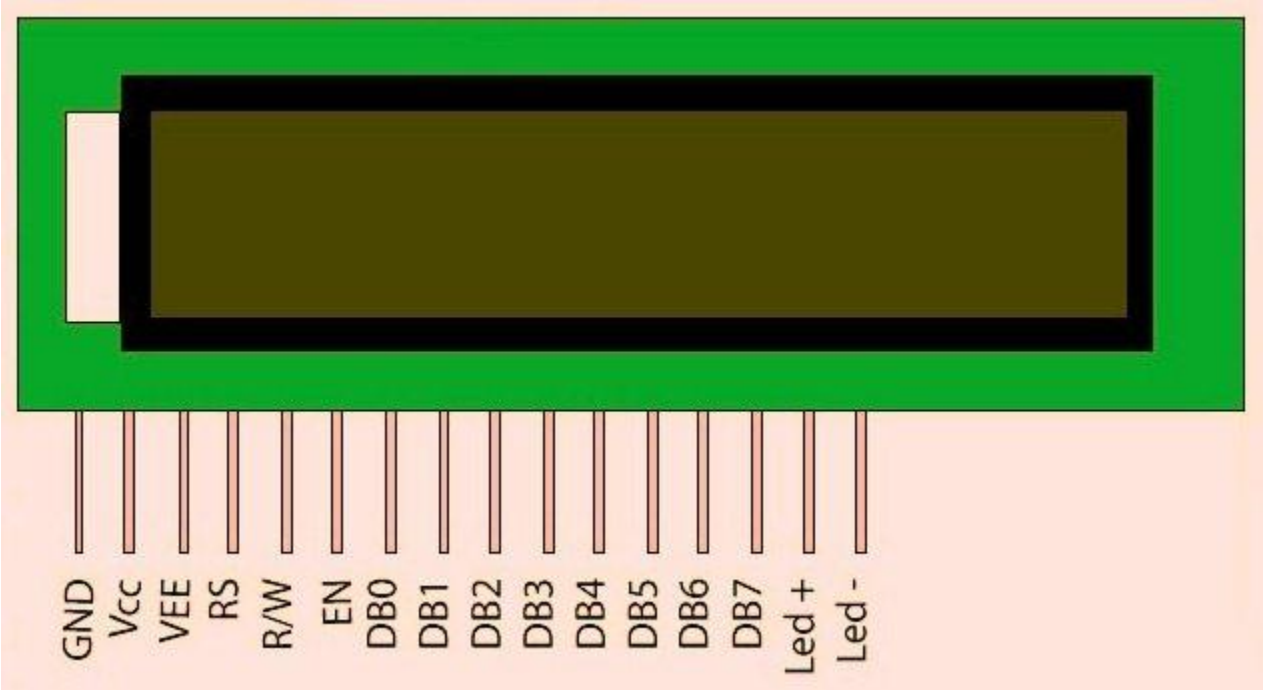
- 1 – Timer T2 is on; and
- 0 – Timer T2 is off.

When using the TMR2 timer, one should know several specific details that have to do with its registers:

- Upon power-on, the PR2 register contains the value FFh;
- Both prescaler and postscaler are cleared by writing to the TMR2 register;
- Both prescaler and postscaler are cleared by writing to the T2CON register; and
- On any reset, both prescaler and postscaler are cleared.

Interfacing LCD with PIC Microcontroller

16×2 Character LCD is a very basic LCD module which is commonly used in electronics projects and products. It contains 2 rows that can display 16 characters. Each character is displayed using 5×8 or 5×10 dot matrix. It can be easily interfaced with a microcontroller. In this tutorial we will see how to write data to an LCD with PIC Microcontroller using Hi-Tech C Compiler. Hi-Tech C has no built in LCD libraries so we require the hardware knowledge of LCD to control it. Commonly used LCD Displays uses HD44780 compliant controllers.



16×2 LCD Pin Diagram

This is the pin diagram of a 16×2 Character LCD display. As in all devices it also has two inputs to give power Vcc and GND. Voltage at VEE determines the Contrast of the display. A 10K potentiometer whose fixed ends are connected to Vcc, GND and variable end is connected to VEE can be used to adjust contrast. A microcontroller needs to send two informations to operate this LCD module, Data and Commands. Data represents the ASCII value (8 bits) of the character to be displayed and Command determines the other operations of LCD such as position to be displayed. Data and Commands are send through the same data lines, which are multiplexed using the RS (Register Select) input of LCD. When it is HIGH, LCD takes it as data to be displayed and when it is LOW, LCD takes it as a command. Data Strobe is given using E (Enable) input of the LCD. When the E (Enable) is HIGH, LCD takes it as valid data or command. The input signal R/W (Read or Write) determines whether data is written to or read from the LCD. In normal cases we need only writing hence it is tied to GROUND in circuits shown below.

The interface between this LCD and Microcontroller can be 8 bit or 4 bit and the difference between them is in how the data or commands are send to LCD. In the 8 bit mode, 8 bit data and commands are send through the data lines DB0 – DB7 and data strobe is given through E input

of the LCD. But 4 bit mode uses only 4 data lines. In this 8 bit data and commands are splitted into 2 parts (4 bits each) and are sent sequentially through data lines DB4 – DB7 with its own data strobe through E input. The idea of 4 bit communication is introduced to save pins of a microcontroller. You may think that 4 bit mode will be slower than 8 bit. But the speed difference is only minimal. As LCDs are slow speed devices, the tiny speed difference between these modes is not significant. Just remember that microcontroller is operating at high speed in the range of MHz and we are viewing LCD with our eyes

Functions in lcd.h

Lcd8_Init() & Lcd4_Init() : These functions will initialize the LCD Module connected to the following defined pins in 8 bit and 4 bit mode respectively.

8 Bit Mode :

```
#define RS RB6
#define EN RB7
#define D0 RC0
#define D1 RC1
#define D2 RC2
#define D3 RC3
#define D4 RC4
#define D5 RC5
#define D6 RC6
#define D7 RC7
```

4 Bit Mode :

```
#define RS RB2

#define EN RB3

#define D4 RB4

#define D5 RB5

#define D6 RB6

#define D7 RB7
```

These connections must be defined for the working of LCD library.

Lcd8_Clear() & Lcd4_Clear() : Calling these functions will clear the LCD Display when interfaced in 8 Bit and 4 Bit mode respectively.

Lcd8_Set_Cursor() & Lcd4_Set_Cursor() : These functions set the row and column of the cursor on the LCD Screen. By using this we can change the position of the character being displayed by the following functions.

Lcd8_Write_Char() & Lcd4_Write_Char() : These functions will write a character to the LCD Screen when interfaced through 8 Bit and 4 Bit mode respectively.

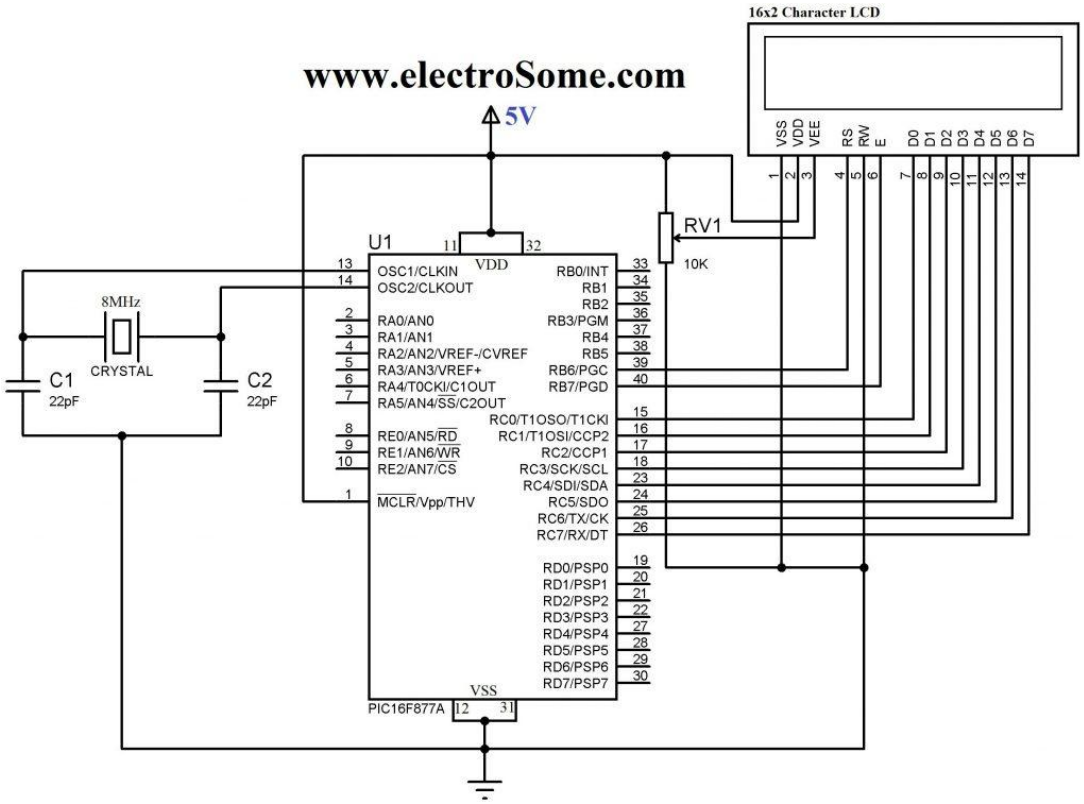
Lcd8_Write_String() & Lcd4_Write_String() : These functions are used to write strings to the LCD Screen.

Lcd8_Shift_Left() & Lcd4_Shift_Left() : These functions are used to shift the content on the LCD Display left without changing the data in the display RAM.

Lcd8_Shift_Right() & Lcd4_Shift_Right() : Similar to above functions, these are used to shift the content on the LCD Display right without changing the data in the display RAM.

8 Bit Mode

Circuit Diagram



Interfacing LCD with PIC Microcontroller – 8 Bit Mode

Hi-Tech C Code

```
#include<htc.h>

#include<pic.h>



#define RS RB6

#define EN RB7
```

```
#define D0 RC0

#define D1 RC1

#define D2 RC2

#define D3 RC3

#define D4 RC4

#define D5 RC5

#define D6 RC6

#define D7 RC7


#define _XTAL_FREQ 8000000


#include "lcd.h"


void main()

{

  int i;

  TRISB = 0x00;

  TRISC = 0x00;

  Lcd8_Init();

  while(1)

  {

    Lcd8_Set_Cursor(1,1);
```

```
    Lcd8_Write_String("electroSome LCD Hello World");
    for(i=0;i<15;i++)
    {
      __delay_ms(1000);
      Lcd8_Shift_Left();
    }
    for(i=0;i<15;i++)
    {
      __delay_ms(1000);
      Lcd8_Shift_Right();
    }
    Lcd8_Clear();
    Lcd8_Set_Cursor(2,1);
    Lcd8_Write_Char('e');
    Lcd8_Write_Char('S');
    __delay_ms(2000);
  }
}
```

Using ADC of PIC Microcontroller – MPLAB XC8

In this tutorial we will learn, how to use the ADC module of a PIC Microcontroller using MPLAB XC8 compiler. For demonstration we will use the commonly available PIC 16F877A microcontroller.

Every physical quantity found in nature like temperature, humidity, pressure, force is analog. We need to convert these analog quantities to digital to process it using a digital computer or a microcontroller. This is done by using Analog to Digital Converters. An Analog to Digital Converter or ADC is a device which converts **continuous analog quantity** (here : voltage) to corresponding **discrete digital values**.

PIC 16F877A microcontroller has 8 ADC inputs and it will convert analog inputs to a corresponding 10 bit digital number. For the sake of explanation take ADC Lower Reference as 0V and Higher Reference as 5V.

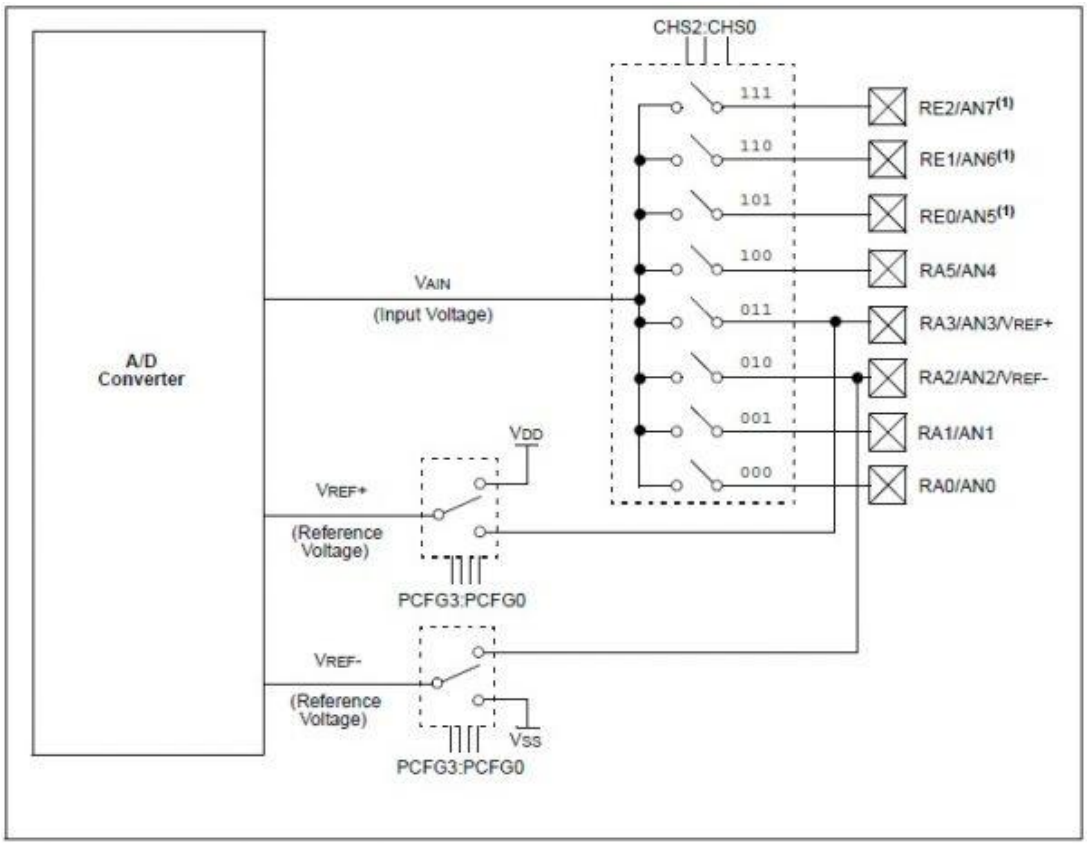
- Vref- = 0V
- Vref+ = 5V
- n = 10 bits
- **Resolution = (Vref+ – Vref-)/(2^n – 1)** = 5/1023 = 0.004887V

So ADC resolution is 0.00487V, which is the minimum required voltage to change a bit. See the examples below.

The ADC module of PIC 16F877A has 4 registers.

- ADRESH – A/D Result High Register
- ADRESL – A/D Result Low Register
- ADCON0 – A/D Control Register 0
- ADCON1 – A/D Control Register 1

ADC Block Diagram



From this block diagram you can easily understand the working of ADC channel selection and reference voltage selection.

A/D Acquisition Time

Holding capacitor (C_{HOLD}) must be charged to the input voltage to meet the accuracy specified by the datasheet. So we must provide a delay greater than the minimum required acquisition time to charge the capacitor. **19.72µs** is the minimum time specified in the datasheet. Please refer the datasheet for more details.

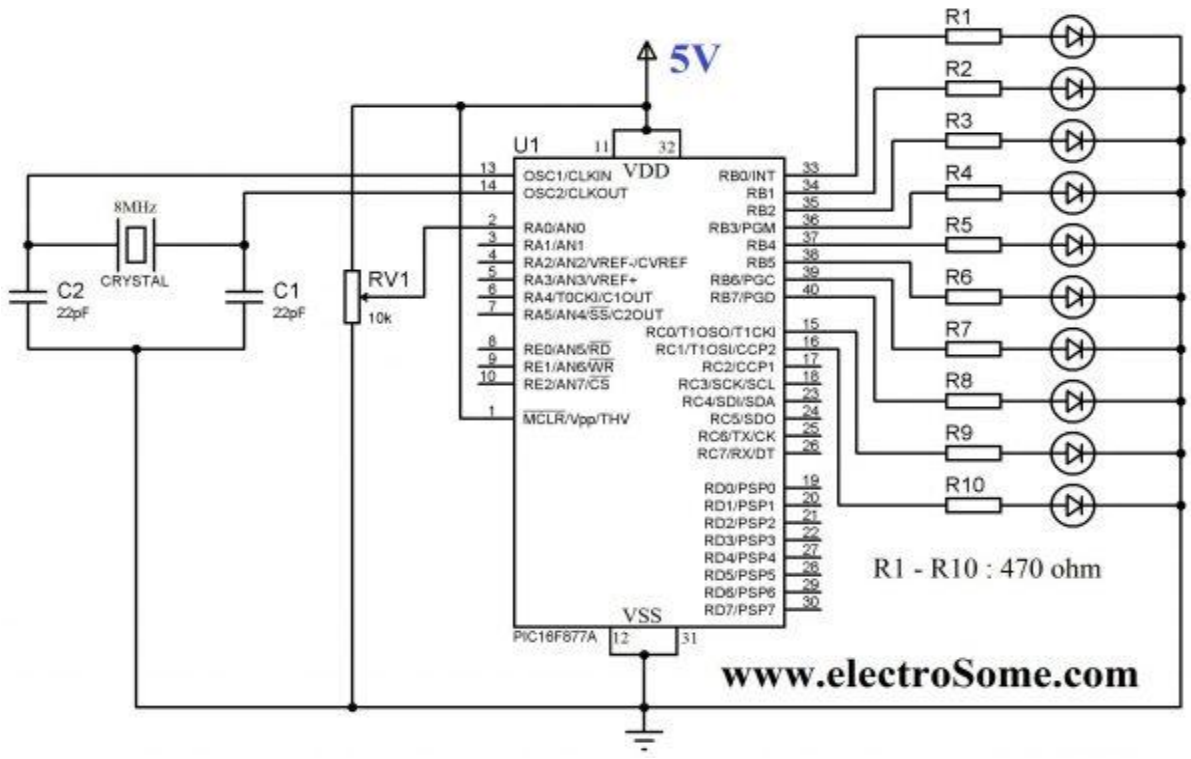
A/D Clock Selection :A/D conversion clock must be selected to ensure minimum T_{AD}. T_{AD} is the conversion time per bit, which is 1.6µs. Please refer the datasheet for more details.

AD Clock Source (TAD)		Maximum Device Frequency
Operation	ADCS2:ADCS1:ADCS0	
2 Tosc	000	1.25 MHz
4 Tosc	100	2.5 MHz
8 Tosc	001	5 MHz
16 Tosc	101	10 MHz
32 Tosc	010	20 MHz
64 Tosc	110	20 MHz
RC(1, 2)	x11	(Note 1)

Note 1: The RC source has a typical TAD time of 4 µs but can vary between 2-6 µs.

2: When the device frequencies are greater than 1 MHz, the RC A/D conversion clock source is only recommended for Sleep operation.

ADC Clock Selection Table – PIC 16F877A



Using Internal ADC Module of PIC Microcontroller

You can easily understand the circuit if you already go through our first tutorials, PIC Microcontroller MPLAB XC8 Tutorials. The reference voltages for A/D conversion is set to VDD (5V) and VSS (GND) in the software (see the code below). Analog input to Channel 0 is provided using a potentiometer such that we can vary the input voltage from 0 ~ 5V. A/D conversion will generate a 10 bit digital value (0 ~ 1023) corresponding to the analog input. This digital value is displayed using 10 LEDs connected PORTB and PORTC of the microcontroller.

MPLAB XC8 Code

```
// CONFIG

#pragma config FOSC = HS      // Oscillator Selection bits (HS oscillator)

#pragma config WDTE = OFF     // Watchdog Timer Enable bit (WDT disabled)

#pragma config PWRTE = OFF    // Power-up Timer Enable bit (PWRT disabled)

#pragma config BOREN = OFF    // Brown-out Reset Enable bit (BOR disabled)

#pragma config LVP = OFF      // Low-Voltage (Single-Supply) In-Circuit Serial Programming Enable bit (RB3 is digital I/O, HV on MCLR must be used for programming)

#pragma config CPD = OFF      // Data EEPROM Memory Code Protection bit (Data EEPROM code protection off)

#pragma config WRT = OFF      // Flash Program Memory Write Enable bits (Write protection off; all program memory may be written to by EECON control)

#pragma config CP = OFF       // Flash Program Memory Code Protection bit (Code protection off)


#include <xc.h>

#include <pic16f877a.h>
```

```
#define _XTAL_FREQ 8000000


void ADC_Init()
{
  ADCON0 = 0x81;               //Turn ON ADC and Clock Selection
  ADCON1 = 0x00;               //All pins as Analog Input and setting Reference Voltages
}


unsigned int ADC_Read(unsigned char channel)
{
  if(channel > 7)              //Channel range is 0 ~ 7
    return 0;


  ADCON0 &= 0xC5;              //Clearing channel selection bits
  ADCON0 |= channel<<3;        //Setting channel selection bits
  __delay_ms(2);               //Acquisition time to charge hold capacitor
  GO_nDONE = 1;                //Initializes A/D conversion
  while(GO_nDONE);             //Waiting for conversion to complete
  return ((ADRESH<<8)+ADRESL); //Return result
}
```

```
void main()
{
  unsigned int a;
  TRISA = 0xFF;              //Analog pins as Input
  TRISB = 0x00;              //Port B as Output
  TRISC = 0x00;              //Port C as Output
  ADC_Init();                //Initialize ADC

  do
  {
    a = ADC_Read(0);         //Read Analog Channel 0
    PORTB = a;               //Write Lower bits to PORTB
    PORTC = a>>8;            //Write Higher 2 bits to PORTC
    __delay_ms(100);         //Delay
  }while(1);                 //Infinite Loop
}
```

PIC HEX FILE FORMAT

Microchip have scored a winner over Atmel AVR by including all the information required to program a PIC microcontroller in one Hex file. This includes code, EEPROM data, User bytes (User ID) and most importantly, configuration words.

This makes it much easier to transfer the project from development to production or between engineers, as all the information needed is in one file. No equivalent mechanism exists for the

AVR and describing fuse settings for AVR is always a headache. Kanda

PIC18F usually use CONFIG directive, for example,
CONFIG WDT=OFF; disable watchdog timer
CONFIG MCLRE = ON; MCLEAR Pin on
CONFIG DEBUG = ON; Enable Debug Mode
CONFIG LVP = OFF;

Check your compiler documentation for more details. Configuration bit names vary from PIC device to PIC device, see "Special Features of CPU section" in PIC device datasheets for details about configuration bytes.

MPLAB X

The latest MPLAB X deals with Configuration Bytes differently in C files. With MPLAB X assembler files, __CONFIG and CONFIG directives still work but the C compiler requires a different format. It requires the use of #pragma config WDTE = ON syntax.

The easiest way to generate the required #pragma directives is with a built-in Configuration Bytes Tool.A window opens with a list of the possible configuration bytes available on device that is set in the project. The available Configuration bytes are different for PIC18F chips but the method is the same. Set them how you want and click "Generate Source Code to Output button. This creates the code you need to cut and paste into your main source file, or put in a separate C file and use #includedirective in your main file.

#pragma config IESO = OFF // Internal/External Oscillator Switchover bit

HEX FILE FORMAT FOR PIC16F DEVICES

The PIC16F file has a similar format to the PIC18F file but does not use extended addressing as the PIC16F devices are smaller. **This section is not correct for the latest PIC16F1xxx chips – see separate section below**.

Code: Code is always at the top of the .hex file. The layout varies with different compilers and assemblers e.g. amount of data per line, whether blank lines are included etc. Note that the PIC16F devices use 14-bit instructions, so code is stored as Words with low byte first. Therefore, an unused location appears as FF3F. Addressing is in bytes though.

EEPROM Data: If the device has EEPROM, the data is stored at address 0x4200 upwards in the HEX file. It is stored in word format but only the lower byte contains data – the high byte is always 0 and is discarded.

Configuration Word: There is only one 14-bit configuration word on most PIC16F, stored at address 0x400E. It is stored high byte first. Some devices have up to 3 words.

User Data: Up to 8 bytes stored at 0x4000.

End of File: The End Of File marker for all Intel Hex files is :00000001FF

Example

```
:100000005D38A23BB437B11731090A0F1202E92358
:100010007B3E20286F335609A104A12BB00DE92D9A
:100020008D22260D260A931C951D510C6711131065
:10420000FF00FF00FF00FF00FF00FF00FF00FF00B6
:06400E00FF3FFF3FFF3FF2
:08400000FF3FFF3FFF3FFF3FC0
:00000001FF
```

Programming tool

A programming tool or software development tool is a computer program that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. The ability to use a variety of tools productively is one hallmark of a skilled software engineer.

List of Tools

- Binary compatibility analysis tools
- Bug databases: Comparison of issue tracking systems - Including bug tracking systems
- Build tools: Build automation, List of build automation software
- Call graph
- Code coverage: Code coverage#Software code coverage tools.
- Code review: List of tools for code review
- Code sharing sites: Freshmeat, Krugle, Sourceforge, GitHub. See also Code search engines.
- Compilation and linking tools: GNU toolchain, gcc, Microsoft Visual Studio, CodeWarrior, Xcode, ICC
- Debuggers: Debugger#List of debuggers. See also Debugging.
- Disassemblers: Generally reverse-engineering tools.
- Documentation generators: Comparison of documentation generators, help2man, Plain Old Documentation, asciidoc
- Formal methods: Mathematical techniques for specification, development and verification
- GUI interface generators
- Library interface generators: SWIG
- Integration Tools
- Memory debuggers are frequently used in programming languages (such as C and C++) that allow manual memory management and thus the possibility of memory leaks and other problems. They are also useful to optimize efficiency of memory usage.
  Examples: dmalloc, Electric Fence, Insure++, Valgrind
- Parser generators: Parsing#Parser development software

- Performance analysis or profiling: List of performance analysis tool
- Revision control: List of revision control software, Comparison of revision control software
- Scripting languages: PHP, Awk, Perl, Python, REXX, Ruby, Shell, Tcl
- Search: grep, find
- Source code Clones/Duplications Finding: Duplicate code#Tools
- Source code editor
  - Text editors: List of text editors, Comparison of text editors
- Source code formatting: indent
- Source code generation tools: Automatic programming#Implementations
- Static code analysis: lint, List of tools for static code analysis
- Unit testing: List of unit testing frameworks

IDE

Integrated development environments combine the features of many tools into one package. They for example make it easier to do specific tasks, such as searching for content only in files in a particular project. IDEs may for example be used for development of enterprise-level applications.

Different aspects of IDEs for specific programming languages can be found in this comparison of integrated development environments.

UNIT-IV

CASE STUDIES OF PIC CONTROLLER

Multiplexed displays are electronic display devices where the entire display is not driven at one time.

Instead, sub-units of the display (typically, rows or columns for a dot matrix display or individual characters for a character oriented display, occasionally individual display elements) are multiplexed, that is, driven one at a time, but the electronics and the persistence of visioncombine to make the viewer believe the entire display is continuously active.

A multiplexed display has several advantages compared to a non-multiplexed display:

- fewer wires (often, far fewer wires) are needed
- simpler driving electronics can be used
- both lead to reduced cost
- reduced power consumption

Multiplexed displays can be divided into two broad categories:

1. character-oriented displays
2. pixel-oriented displays

Character-oriented displays

Most character-oriented displays (such as seven-segment displays, fourteen-segment displays, and sixteen-segment displays) display an entire character at one time. The various segments of each character are connected in a two-dimensional diode matrix and will only illuminate if both the "row" and "column" lines of the matrix are at the correct electrical potential. The light-emitting element normally takes the form of a light-emitting diode (LED) so electricity will only flow in one direction, keeping the individual "row" and "column" lines of the matrix electrically isolated from each other. For liquid crystal displays, the intersection of the row and column is not conductive at all.

In the example of the VCR display shown above, the illuminated elements are the plates of many individual triode vacuum tubes sharing the same vacuum enclosure. The grids of the triodes are

arranged so that only one digit is illuminated at a time. All of the similar plates in all of the digits (for example, all of the lower-left plates in all of the digits) are connected in parallel. One by one, the microprocessor driving the display enables a digit by placing a positive voltage on that digit's grid and then placing a positive voltage on the appropriate plates. Electrons flow through that digit's grid and strike those plates that are at a positive potential.

If the display had been built with every segment being individually connected, the display would have required 49 wires just for the digits, with more wires being needed for all of the other indicators that can be illuminated. By multiplexing the display, only seven "digit selector" lines and seven "segment selector" lines are needed. The extra indicators (in our example, "VCR", "Hi-Fi", "STEREO", "SAP", etc.) are arranged as if they were segments of an additional digit or two or extra segments of existing digits and are scanned using the same multiplexed strategy as the real digits.

Most character-oriented displays drive all the appropriate segments of an entire digit simultaneously. A few character-oriented displays drive only one segment at a time. The display on the Hewlett-Packard HP-35 was an example of this. The calculator took advantage of an effect of pulsed LED operation where very brief pulses of light are perceived as brighter than a longer pulse of light with the same time-integral of intensity.

A keyboard matrix circuit has a very similar arrangement as a multiplexed display, and has many of the same advantages. In order to reduce the number of wires even further, some people "share" wires between a multiplexed display and a keyboard matrix, reducing the number of wires even further.[1]

Pixel-oriented displays

By comparison, in dot matrix displays, individual pixels are located at the intersections of the matrix's "row" and "column" lines and each pixel can be individually controlled.

Here, the savings in wiring becomes far more dramatic. For a typical 1024x768 (XGA) computer screen, 2,359,296 wires would be needed for non-multiplexed control. That many wires would be completely impractical. But by arranging the pixels into a multiplexed matrix, only 1792 wires are needed; a completely practical situation.

Pixel-oriented displays may drive a single pixel at a time or an entire row or column of pixels simultaneously. Active-matrix liquid crystal displays provide a storage element at each pixel so that the pixel continues to display the correct state even when not being actively driven.

Electronic controller for the washing machine

It's 2007. My great old washing machine (an Aurora T-5502) got stuck while in the spin cycle and spun for hours. When I found out it was too late and there was already some damage in the machine.Cause? The electromechanical timer. It was worn-out and the contacts couldn't rotate as originally designed. The old timer couldn't be repaired and has to be replaced.The timer was like an old fashioned clock, where the hand touches different contacts along the way, activating different functions.

The machine provides 4 different full programs, 2 short ones, and configuration:

- Ropa de algodón (cotton)
- Ropa delicada (delicate)
- Ropa muy sucia (very dirty)
- Modo agresivo (aggressive mode)
- Drain
- Spin only
- Configuration

For each program following parameters can also be selected:

- Prewash
- Temperature
- Disable spin cycle
- Initial delay

Washing Machine Internals

The system is divided in three parts:

- The control panel
- The motherboard
- The interface with the washing machine

The control panel

It's the interface with the user. Contains:

- LCD screen (16x2)
- 3 buttons (Left, Right, Enter)
- Red power led (to show the power stage is working)
- Yellow led (show established link to pc, if connected)
- Piezoelectric sounder (to generate tones)

The LCD has his own driver (microcontroller). Mine was the HD44780, and the protocol can be found on the datasheet. I used a library which spared me the headache. Sadly the driver is very sensitive to EMI, which caused a LOT of trouble.

A special consideration is required with the buttons, specially if you buy cheap ones: when you press the button the metallic pieces bounce at an almost microscopic level. In many cases the bounce interrupts the electric flow up to many times for some microseconds before completely close the circuit as expected to. As the microcontroller is fast enough it could wrongly sense many presses of the button even if you only pressed it once.

There is two solutions:

- Hardware filters (low pass)
- Filter it by software

The sounder is connected directly to the microcontroller, being able to generate a wide range of tones.

The motherboard

It's the central part of the design. It hosts the microcontroller and connects it with the peripherals.

The design was made with Eagle CAD. You can find it on Github.

The PCB it's made of cheap Pertinax (FR-2) to save some money. No high frequencies to justify a higher quality.

The first thing is the power source. The input voltage is 9VAC 1A (a 220v/9v transformer with fuse). The motherboard contains a diode bridge to obtain 12VDC for the relays and the servo and voltage regulator (LM7805) to power the LCD and the micro at 5VDC.

Peripherals are connected to the mainboard trough housing connectors, making removal of the motherboard very easy.

The microcontroller used is the PIC 16F877PI with at 4Mhz. There was no need to run at a higher frequency. It also worked better at this frequency with the tone generator library for the piezo sounder.

To program it there is a ICSP socket, to allow direct firmware updates without having to extract the micro.

There is optical isolation with the power stage using optocouplers (HPCL-817).

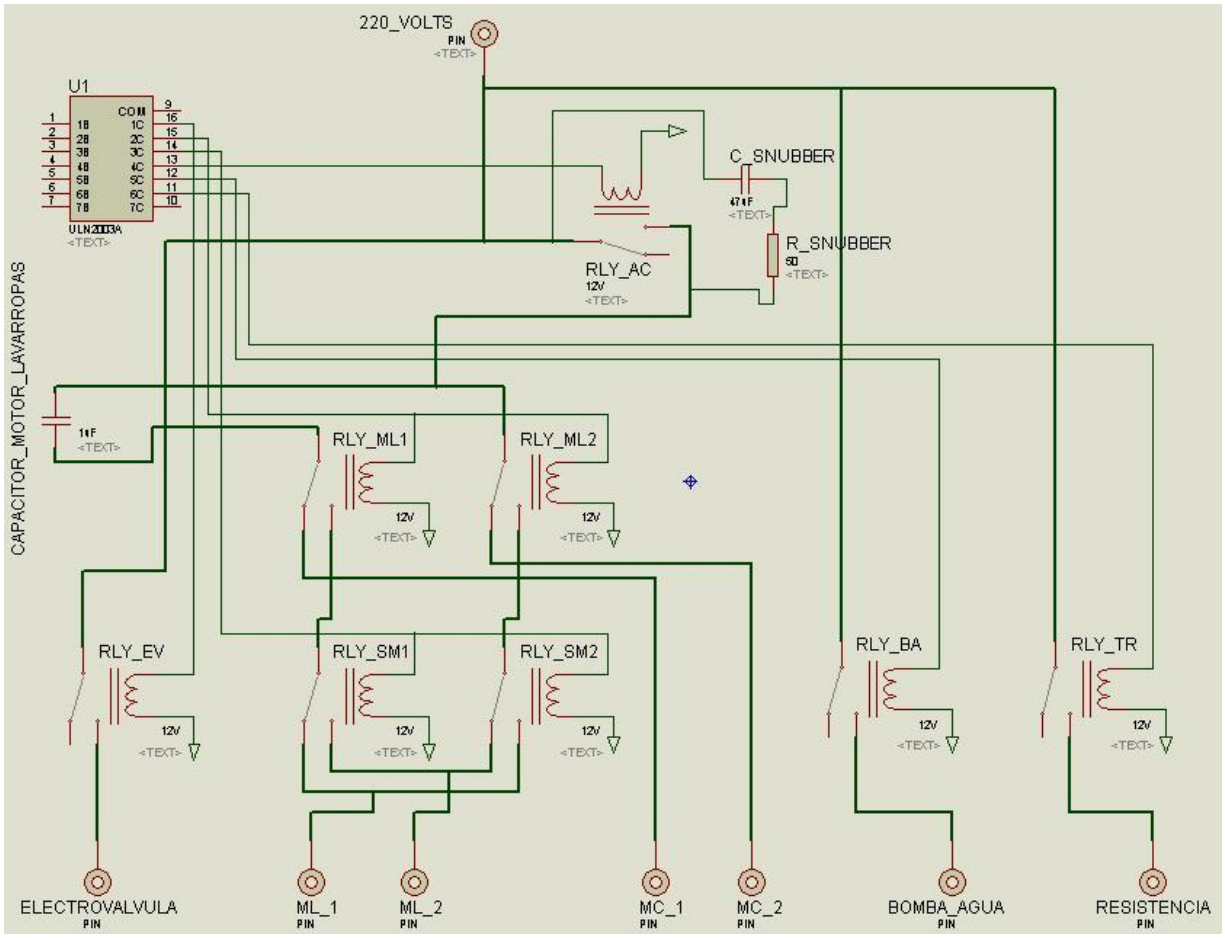
The firmware is C code compiled with the PCM compiler from CCS, a very good PIC compilers of that time. The firmware uses about 85% of the ROM and required some nasty tricks to fit them in.

Interface with the washing machine

Power stage

It consists in a 'panel' of relays, acting as switches to control the electric flow to the different components of the machine:

The panel diagram:



The actual panel:

- Electrovalve(RLY_EV): an electrically controlled water inlet. It let water flow when an electrical current is applied. The water goes directly to the dispenser system
- The heater(RLY_TR): a resistance to heat the water
- Drain pump(RLY_BA): to extract the water from the wash drum

- The motor(RLY_AC, RLY_ML1, RLY_ML2, RLY_SM1, RLY_SM2): provides the rotation of the drum. My machine has one motor with 2 different coils. One for slow speed and one for the spin cycle.

The connections to Electrovalve, Heater and Pump were relatively straightforward. So I will omit the explanation.

To interface the motor I had to do some experimentation to find out how to make it work. These are the results:

Lets assign numbers to the connector pins (from left to right). The upper row 1, 2, 3 and 4, 5, 6 to the bottom row.

Pin number

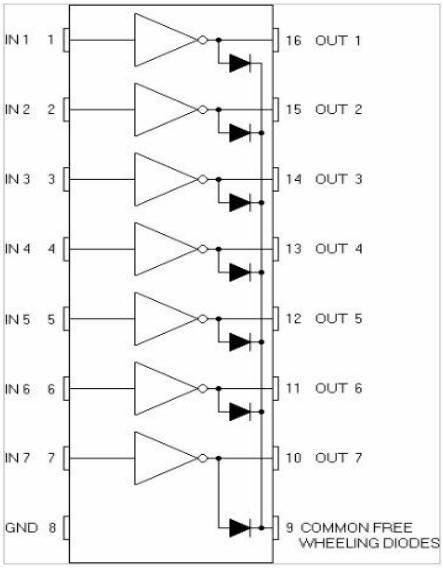
1	GND
2	High Speed Coil (MC_1)
3	Low Speed Coil (ML_1)
4	GND
5	High Speed Coil (MC_2)
6	Low Speed Coil (ML_2)

The relays RLY_MLx makes the commutation between low speed coil (for normal washing) ML_x and high speed one (for spinning) MC_x.

When the low speed coil is active, the relays RLY_SM control the direction of rotation (see panel diagram). When the high speed coil is active they have no effect.

The relay RLY_AC switch the motor on and off. The snubber reduces the EMI of the commutation, which is horrible for big inductive loads like the motor.

To control the relays I used an array of Darlington transistors (ULN2003AN). Each Darlington comes with a freewheeling diode, to protect the transistors from coil auto induction, when driving inductive loads.



Sensors

Pressure switch: allows to know when the water reached a predefined level by measuring the air pressure exert by the water in a special air chamber. It's used as a normal switch.

Door switch: actually a main switch, which is closed only by the door. When the door closes the machine is turned on. I keep using it as originally designed. The door switch doesn't contains any lock mechanism.

Temperature sensor: originally was a bimetalic thermostat to measure water temperature and control the heater. I completely replaced it with a LM35 temperature sensor reusing the original metallic housing of the thermostat. The LM35 was attached to the housing using super glue.

Dispenser system

This machine has only one water inlet and a rotatory arm to divert the water flow into the different compartments. The arm was originally controlled by the electromechanical timer. To

achieve the same function I used a low cost servo for model airplanes (I took the idea from Pablo Canello)

The servo was attached to the rotatory arm (green) using a little bit of bricolage ingenuity.

The servo has 3 lines, VCC, GND and signal. To control the servo position the PIC will generate a PWM signal at 50Hz. Varying the pulse within 1ms to aprox 2.5ms will be linearly translated to 0 degrees and 180 degrees respectively.

As long the PWM signal is present, the servo will do all it can to get to the desired position. If the signal is interrupted the servo remains in its last position and will not try to correct externally applied forces

Every time before loading water, the servo will be positioned in the corresponding compartment.

To correctly move the rotatory arm, the servo position corresponding to each compartment of the dispenser will be recorded with help of a calibration routine (a once time operation). Each position is a value between 0 and 255 (one byte) and will be stored in the PIC EEPROM.

Operation

Simplifying a little, the operation starts by selecting the washing parameters.

It's followed by many cycles of the following tasks:

1. Select the desired compartiment on the dispenser system
2. Load water until pressure switch closes
3. Wash alternating directions
4. Pump water out

The operation finishes with the spinning cycle (if the spin option wasn't excluded when selecting the parameters).

Interfacing to sensors and actuators in embedded control applications

The use of sensors in embedded electronic devices spans across a wide range of applications, from fundamental scientific and analytical measurements to consumer electronics, and each type of sensors (be it temperature sensors, humidity sensors, strain gauges, or RTDs) presents unique requirements when interfacing with an embedded controller. Modern laboratory and analytical instruments are prolific data generators with obvious needs for communication. They incorporate impressive technology implemented with increasingly complex electronics, yet users demand simple graphical interfaces and intuitive menu-based front panel controls.

Mosaic Industries' embedded controllers and diverse expansion I/O modules called Wildcards are designed to meet these needs by providing computing power to run the instrument's application software, a graphical user interface (GUI) for local operation, mix and match I/O for sensing and control, serial communications for interface to other serial-controlled instruments and sensors, Ethernet and TCP/IP for connection to a local area network (LAN), and web service for communication via remote web browsers.

This set of hardware and software components works together to provide a seamless and transparent interface to sensors and actuators.

Mosaic's embedded controllers promote a high level of software/hardware integration for interfacing to sensors and actuators

We provide a suite of software development tools for our sensor interface boards to simplify your programming. These comprehensive tools include:

- an Integrated Development Environment (IDE) with editor and terminal
- C and Forth compilers, assembler, interactive debugger
- Multitasking real-time operating system, RTOS
- Modular device drivers
- Precoded libraries
- Graphical User Interface (GUI) toolkit
- Graphics Image Converter

All Mosaic sensor interface controllers include pre-coded device drivers providing full high level access to their functions using either the C or Forth programming languages.

Pre-coded I/O drivers facilitate data acquisition, pulse width modulation, motor control, PID control, sensor calibration, frequency measurement, data analysis, data logging, analog control,

and communications.

Extensive documentation with precoded sample programs helps you finish your application quickly.

DC Motor Speed Control using PWM with PIC Microcontroller

You may think that a variable resistor in series with a DC Motor can control its speed. There are three reasons for "Resistor is not a good choice for controlling the speed of a DC Motor".

- The main problem is that the motor is a varying electrical load so a resistor can't do this task. It needs more power during start up than in running state. It draws more current also when a mechanical load is applied to motor shaft.
- The resistor drops excess energy as heat. Thus it is not good for a battery powered device.
- We all know that motor requires more current, so resistors with higher power rating are required to drop excess energy.

PWM can be easily generated using the inbuilt CCP module of a PIC Microcontroller. CCP stands for Capture/Compare/PWM. CCP modules are available with a number of PIC Microcontrollers. Most of them have more than one CCP module. MikroC Pro for PIC Microcontroller provides built in library routines for PWM which makes our task very simple. In this example project DC Motor is interfaced with PIC Microcontroller using L293D Motor Driver. Two Push Button switches are provided to control the speed of the motor. Here we are using 12V DC Motor and average DC value delivered to motor can be varied by varying the duty ratio of the PWM. The average DC Voltage of 0% duty cycle is 0V, 25% duty cycle is 3V, 50% duty cycle is 6V, 75% duty cycle is 9V and for 100% duty cycle 12V.

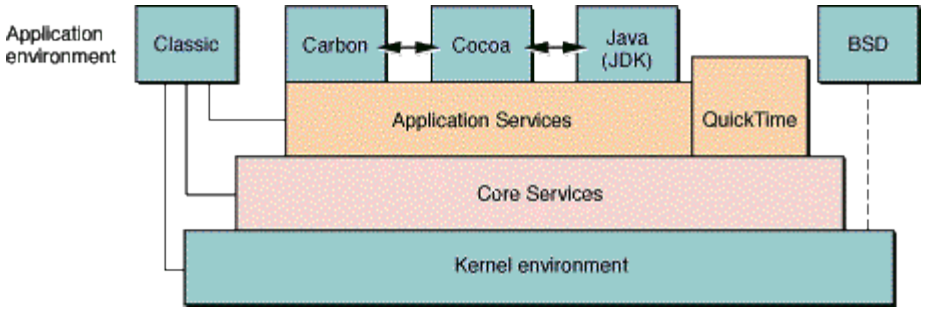
UNIT V –REALTIME OPERATING SYSTEM

Kernel Architecture Overview

OS X provides many benefits to the Macintosh user and developer communities. These benefits include improved reliability and performance, enhanced networking features, an object-based system programming interface, and increased support for industry standards.

In creating OS X, Apple has completely re-engineered the Mac OS core operating system. Forming the foundation of OS X is the kernel. Figure 3-1 illustrates the OS X architecture.

Figure 3-1 OS X architecture



The kernel provides many enhancements for OS X. These include *preemption*, *memory protection*, enhanced performance, improved networking facilities, support for both Macintosh (Extended and Standard) and non-Macintosh (UFS, ISO 9660, and so on) file systems, object-oriented APIs, and more. Two of these features, preemption and memory protection, lead to a more robust environment.

In Mac OS 9, applications cooperate to share processor time. Similarly, all applications share the memory of the computer among them. Mac OS 9 is a *cooperative multitasking* environment. The responsiveness of all processes is compromised if even a single application doesn't cooperate. On the other hand, real-time applications such as multimedia need to be assured of predictable, time-critical, behavior.

In contrast, OS X is a *preemptive multitasking* environment. In OS X, the kernel provides enforcement of cooperation, scheduling processes to share time (preemption). This supports real-time behavior in applications that require it.

In OS X, processes do not normally share memory. Instead, the kernel assigns each *process* its own *address space*, controlling access to these address spaces. This control ensures that no application can inadvertently access or modify another application's memory (protection). Size is not an issue; with the virtual memory system included in OS X, each application has access to its own 4 GB address space.

Viewed together, all applications are said to run in user space, but this does not imply that they share memory. User space is simply a term for the combined address spaces of all user-level applications. The kernel itself has its own address space, called kernel space. In OS X, no application can directly modify the memory of the system software (the kernel).

Although user processes do not share memory by default as in Mac OS 9, communication (and even memory sharing) between applications is still possible. For example, the kernel offers a rich set of primitives to permit some sharing of information among processes. These primitives

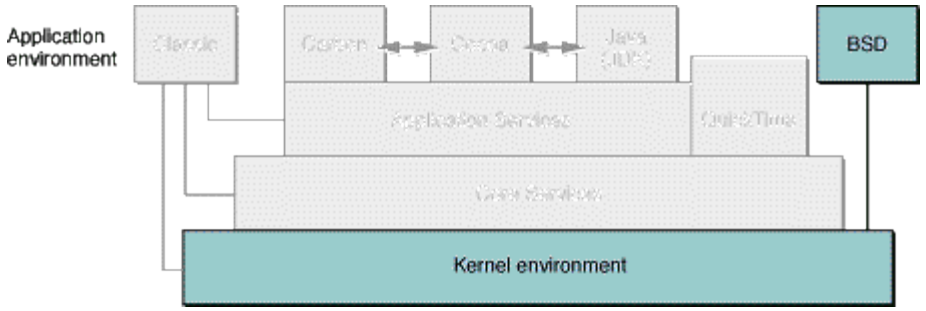
include shared libraries, frameworks, and POSIX shared memory. Mach messaging provides another approach, handing memory from one process to another. Unlike Mac OS 9, however, memory sharing cannot occur without explicit action by the programmer.

Darwin

The OS X kernel is an *Open Source* project. The kernel, along with other core parts of OS X is collectively referred to as *Darwin*. Darwin is a complete operating system based on many of the same technologies that underlie OS X. However, Darwin does not include Apple's proprietary graphics or applications layers, such as Quartz, QuickTime, Cocoa, Carbon, or OpenGL.

Figure 3-2 shows the relationship between Darwin and OS X. Both build upon the same kernel, but OS X adds Core Services, Application Services and QuickTime, as well as the *Classic*, *Carbon*, *Cocoa*, and Java (JDK) application environments. Both Darwin and OS X include the BSD command-line application environment; however, in OS X, use of environment is not required, and thus it is hidden from the user unless they choose to access it.

Figure 3-2 Darwin and OS X



Darwin technology is based on *BSD*, Mach 3.0, and Apple technologies. Best of all, Darwin technology is Open Source technology, which means that developers have full access to the source code. In effect, OS X third-party developers can be part of the Darwin core system software development team. Developers can also see how Apple is doing things in the core operating system and adopt (or adapt) code to use within their own products. Refer to the *Apple Public Source License* (APSL) for details.

Because the same software forms the core of both OS X and Darwin, developers can create low-level software that runs on both OS X and Darwin with few, if any, changes. The only difference is likely to be in the way the software interacts with the application environment.

Darwin is based on proven technology from many sources. A large portion of this technology is derived from FreeBSD, a version of 4.4BSD that offers advanced networking, performance, security, and compatibility features. Other parts of the system software, such as Mach, are based on technology previously used in Apple's MkLinux project, in OS X Server, and in technology acquired from NeXT. Much of the code is platform-independent. All of the core operating-system code is available in source form.

The core technologies have been chosen for several reasons. Mach provides a clean set of abstractions for dealing with memory management, interprocess (and interprocessor) communication (IPC), and other low-level operating-system functions. In today's rapidly

changing hardware environment, this provides a useful layer of insulation between the operating system and the underlying hardware.

BSD is a carefully engineered, mature operating system with many capabilities. In fact, most of today's commercial UNIX and UNIX-like operating systems contain a great deal of BSD code. BSD also provides a set of industry-standard APIs.

New technologies, such as the I/O Kit and Network Kernel Extensions (NKEs), have been designed and engineered by Apple to take advantage of advanced capabilities, such as those provided by an object-oriented programming model. OS X combines these new technologies with time-tested industry standards to create an operating system that is stable, reliable, flexible, and extensible.

Architecture

The foundation layer of Darwin and OS X is composed of several architectural components, as shown in Figure 3-3. Taken together, these components form the *kernel environment*.

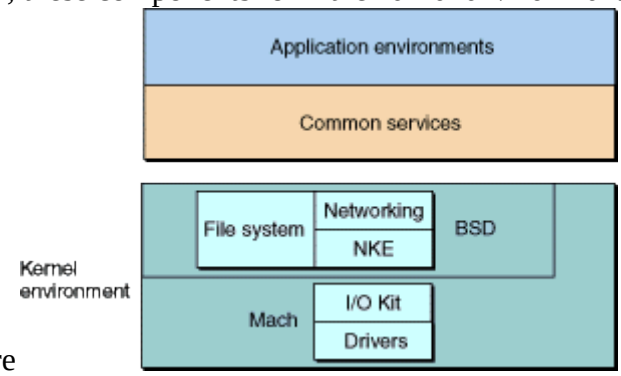


Figure 3-3 OS X kernel architecture

Important: Note that OS X uses the term *kernel* somewhat differently than you might expect. "A kernel, in traditional operating-system terminology, is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating-system services." — from *The Design and Implementation of the 4.4 BSD Operating System*, McKusick, Bostic, Karels, and Quarterman, 1996.

Similarly, in traditional Mach-based operating systems, the kernel refers to the Mach microkernel and ignores additional low-level code without which Mach does very little.

In OS X, however, the kernel environment contains much more than the Mach kernel itself. The OS X kernel environment includes the Mach kernel, BSD, the I/O Kit, file systems, and networking components. These are often referred to collectively as the kernel. Each of these components is described briefly in the following sections. For further details, refer to the specific component chapters or to the reference material listed in the bibliography.

Because OS X contains three basic components (Mach, BSD, and the I/O Kit), there are also frequently as many as three APIs for certain key operations. In general, the API chosen should match the part of the kernel where it is being used, which in turn is dictated by what your code is

attempting to do. The remainder of this chapter describes Mach, BSD, and the I/O Kit and outlines the functionality that is provided by those components.

The process & the operating system OS . These two abstractions lets us switch the state of the process between multiple tasks. The process clearly defines the status of an executing between programs while the OS provides the mechanism for switching execution between the processes.

Operating System Basics

- Bridges between user application/tasks & system resources
- The OS manages the system resources & make them available to the user application/task.

computing system is collection of different I/O sub-system working &

storage memory.

Primary FunctionOf OS

- Make the system convenient to use
- Organizes & manage the system resources efficiently & correctly

The Kernel

The kernel is core of the OS & is responsible for managing the system & the communication among the hardware & the system services. Kernel acts as the abstraction layer between system resources & user application. Kernel contains a set of system libraries of services. For general purpose OS the kernel contains different services for handling the management.

1. Process Management: Managing the

- process/tasks
- Setup the memory space for process
- Load program/code into space(memory)
- Scheduling & managing the execution of the process
- Setting up & managing the PCB

- Inter process communication & system synchronization process termination & deletion

2. Primary Memory Management

- It is volatile memory
- The MMU of kernel is responsible for

- Keeping track of which part of memory area is correctly used by which process

- Allocating &deal locating the memory spaces

3. File System Management

- File is a collection of related information
- A file could be a program, tent files, image file, word documents, audio/video files etc.
- The file system management service of kernel is responsible for

- The creation & deletion of files

- Creation , deletion & alteration directly

- Saving the file in secondary storage memory

- Providing automatic allocation of spaces

- Providing a flexible naming conversion for the files

4. I/O System(Device) Management

- Kernel is responsible for routing the I/O request coming from different user applications
- Direct access of I/O devices are not allowed , we can access the I/O

devices through the API imposed by kernel

- Dynamically update the available devices
- Device manager of the kernel is responsible for handling I/O device related operations
- The kernel talks to the I/O device through the device driver , this is responsible for

- Loading & unloading of device drivers

- Exchanging the information's

5. Secondary Storage Management

- The secondary storage management deals with secondary storage memory device , if any connected to the system
- The main memory is volatile
- The secondary storage management services of kernel deals with

- Disk storage allocation

- Disk scheduling

- Free disk space management

What is RTOS?

1) A real time operating system (RTOS) is an operating system that guarantees a certain **capability** within a **specified time constraint**.

2) An OS is a system program that provides an **interface** between

Application programs and the computer system (**hardware**)

3) The applications where dependability that a certain task will finish before a particular **deadline** is just as obtaining the **correct results**.

4) Besides meeting deadlines RTOS must also be able to respond **predictably** to unpredictable **events** and process **multiple events** concurrently.

A **Real-Time Operating System** (RTOS) is a computing environment that reacts to input within a specific time period. A real-time deadline can be so small that system reaction appears instantaneous. The term real time computing has also been used, however, to describe "slow real- time" output that has a longer, but fixed, time limit.

Learning the difference between real-time and standard operating systems is as easy as imagining yourself in a computer game. Each of the actions you take in the game is like a program running in that environment. A game that has a real-time operating system for its environment can feel like an extension of your body because you can count on a specific "lag time:" the time between your request for action and the computer's noticeable execution of your request. A standard operating system, however, may feel disjointed because the lag time is unreliable. To achieve time reliability, real-time programs and their operating system environment must prioritize deadline actualization before anything else. In the gaming example, this might result in dropped frames or lower visual quality when reaction time and visual effects conflict.

Real-Time Kernel

The heart of a real-time OS (and the heart of every OS, for that matter) is the **kernel**. A kernel is the central core of an operating system, and it takes care of all the OS jobs:

1. Booting
2. Task Scheduling
3. Standard Function Libraries

In an embedded system, frequently the kernel will boot the system, initialize the ports and the global data items. Then, it will start the scheduler and instantiate any hardware timers that need to be started. After all that, the Kernel basically gets dumped out of memory (except for the library functions, if any), and the scheduler will start running the child tasks. Standard function libraries: In an embedded system, there is rarely enough

memory (if any) to maintain a large function library. If functions are going to be included, they must be small, and important.

Basic Kernel Services

Kernel which is a major part of an operating system that provides the most basic services to application software running on a processor. The "kernel" of a real-time operating system ("RTOS") provides an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run.

Scheduling

In typical designs, a task has three states:

1. Running (executing on the CPU);
2. Ready (ready to be executed);
3. Blocked (waiting for an event, I/O for example).

Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can vary greatly, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-pre-emptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state (resource starvation).Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which pre-emption is inhibited, and, in some cases, all interrupts are disabled. But the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires

walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit pre-emption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the fly back time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per ready-queue entry, and

restoration of the highest-priority ready task will take 5 to 30 instructions.

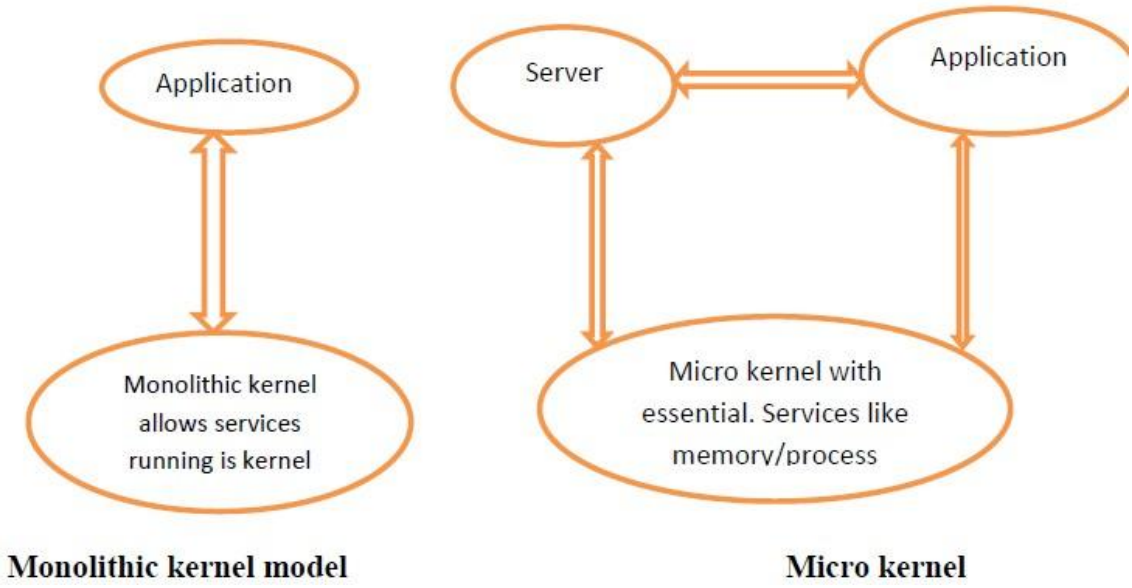
In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Types of Kernel:

Based on kernel design, kernel can be classified into Monolithic & Micro

- In Monolithic kernel architecture all the kernel service run in the kernel space – means all kernel module run with same memory space under single kernel thread

- The drawback of Monolithic kernel is that any error or failure in any of the kernel module i.e. leads to the crashing of entire kernel application. Ex : LINUX,SOLARIS,MS-DOS



Monolithic kernel model **Micro kernel**

Micro kernel: design incorporates only the essential set of OS services into the kernel. The rest of the OS services are implanted in programs known as “servers” which runs in user space.

The essential services of the Micro kernel are

- Memory management
- Process management Timer
- system
- Interrupt handler

Ex. OS , MACH , ONX , MINIX3

Benefits of Micro kernel

1. Robust

2. Configurability

Types of Operating System

Depending on the type of kernel & kernel services purpose & type of in computing system OS are classified into two types

1. General Purpose Operating System[GPOS]

2. Real Time Operating System[RTOS]

1. General Purpose Operating System[GPOS]

The OS which are deployed in general computing systems
are referred as general purpose OS.

The kernel of such OS is more generalized & it contains all kinds of services required for executing generic applications

- GPOS are non-deterministic in behaviour

Ex: PC/Desktop system, windows XP & MS-DOS

2. Real Time Operating System [RTOS]

There is no universal definition available for the term RTOS.

- A RTOS is an OS intended to serve real-time application requests. It must be able to process the data as it comes in typically without buffering delays

- Processing time requirements are measured in tenth of seconds. Hard

RTOS has less jitter than soft RTOS.

[Jitter: Variation in time between packet arriving with time congestion &

time.]

- Real-time implies deterministic timing behaviour – means the OS

services consumes only known & expected amount of time regardless the no of services.

- RTOS decides which application should run in which order & how much time needs to be allocated for each applications.

Ex: Windows CE, UNIX, VxWorks, Micros/OS-II Real Time Kernel:

The RTOS is referred to as Real Time kernel in complement to conventional OS kernel. The kernel is highly specialized & it contains only the minimal set of services required for running the user application/

Tasks.

The Basic Functions of Real Time Kernel:

1. Task/Process Management
2. Task/Process Scheduling
3. Task/Process Synchronization
4. Error/Exception Handling
5. Memory Management
6. Interrupt Handling
7. Time Management

Task/Process Management: Deals with setting up the memory space for the tasks, loading the tasks code into memory space, allocatingsystem resources, setting up a Task Control Block [TCB] for the task &task/process termination/deletion.

TCB: Task Control Block is used for holding the information corresponding to a task. The TCB contains the following set of information

Task ID: Task identification number

- Task State: The current state of the task

- Task Type: Indicates what is the type of this task, the task can be hard real time or soft real time or background task
- Task Priority: Ex task priority=1 for task with priority =1
- Task Context Pointer: context pointer, pointer for saving context
- Task Memory Pointer: Pointers to the code memory, data memory & stack memory for the task
- Task System Resource Pointers: Pointer to system resource used by the task
- Task Pointers: Pointer to other TCB's
- Other Parameters: Other relevant task parameters

The TCB parameters vary across different kernels based on the task management Implementation

- Creates TCB for a task on creating task
- Delete/Remove the TCB when the task is terminated or deleted
- Reads the TCB to get the state of a task
- Updates the TCB with updated parameters on need basics
- Modify the TCB to change the priority of task dynamically

Task/Process Scheduling: Sharing the CPU among various tasks/process. A kernel applications called scheduler, handles the task scheduling. Scheduler is nothing but an algorithm implemented all to , which performs the efficient & optimal scheduling of task to provide a deterministic behaviour.

Task/Process Synchronization: deals with system concurrent accessing a resource which is shared access multiple tasks & communication between various tasks.

Error/Exception Handling: Deals with registering& handling the errors occurred during the execution of tasks. Ex: Insufficient memory, time outs, dead locks, dead line missing, bus error, divide by zero, unknown instruction execution. Errors & exceptions can happen at two levels of services * Kernel Level Service * At Task Level

Memory Management: RTOS makes use of 'Block Based Memory' allocation techniques instead of the usual dynamic memory allocation technique used by GPOS. RTOS kernel uses blocks of fixed size dynamic memory & block is allocated for a task on a need of basis. A few RTOS kernel implements Virtual Memory concepts avoid the garbage collection overhead.

Interrupt Handler: Deals with the handling of several of interrupts. Interrupts provide Real Time behaviour embedded system design process

to systems. Interrupts inform the processor that an external device or an associated task required immediate attention of the CPU. Interrupts can

be either Synchronous Asynchronous. Interrupts which occurs in synch with the currently executing task is known as synchronous interrupts. Usually the system interrupts fall under the synchronous category Ex

Divide by zero, memory segmentation error. A synchronous interrupts are those which occurs at any point of execution of any task & are not in sync with currently executing tasks.

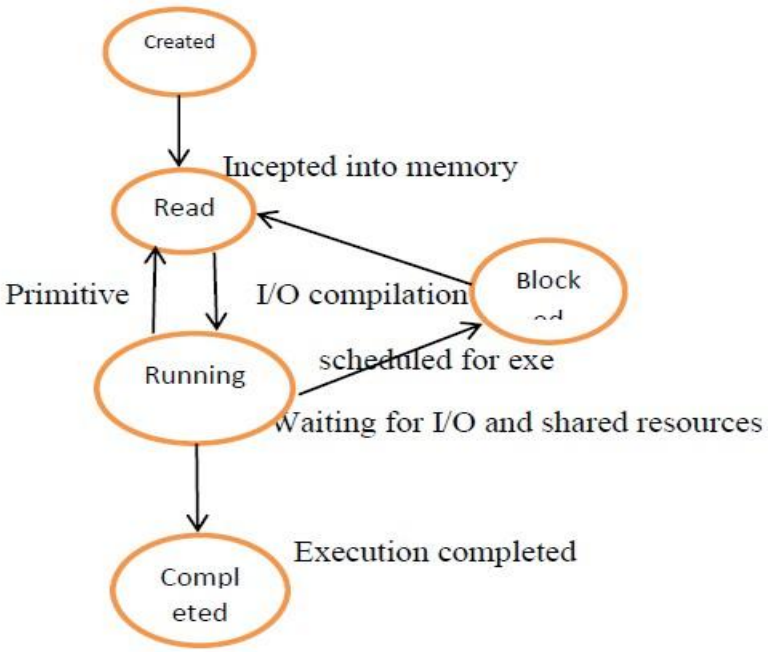
Task, Process & Threads

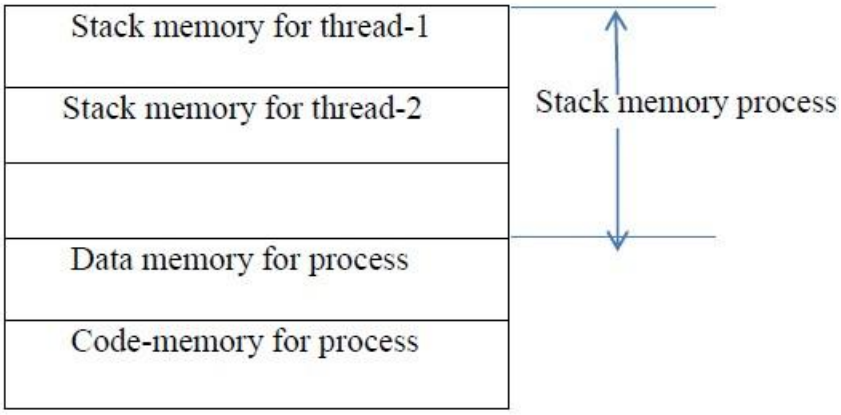
Task: Refers to something that needs to be done. The tasks refers can be one assigned by manages or the one assigned by only one of the processor family needs. In addition we have an order of priority & scheduling timeline for executing these tasks. In OS context, a task is defined as the program in execution & related information maintained by the OS. Task is also known as "JOB" in the OS context.

Process: A process is a program as part of the program, in execution process is also known as instance of the program in execution, multiple instance of the same program can execute simultaneously. Process requires various system resources like CPU for executing the

process, memory for storing the code corresponding to the process, I/O devices for in function exchange etc.

Process States & State Transition





Memory organization of a process and its associated threads Run: A task enters this state as it starts executing on the process Ready state of those tasks that are ready to execute but

cannot be executed because the processor is assigned to another task Blocked a tasks enters this state when it executes synchronization primitive to wait for an event

Ex:-a wait primitive on a semaphore this case the task is inserted in a queue associated with the semaphore.

Created (idle) a periodic job enters this state when it computes its execution and has to wait for the beginning of the next period **Process management:**

Process management deals with the creating a process, setting up the

memory space for the process, loading the process code with the memory space, allocating the system resources.Setting up a process control block (PCB) for the process and process termination/detection. **Task scheduling:**

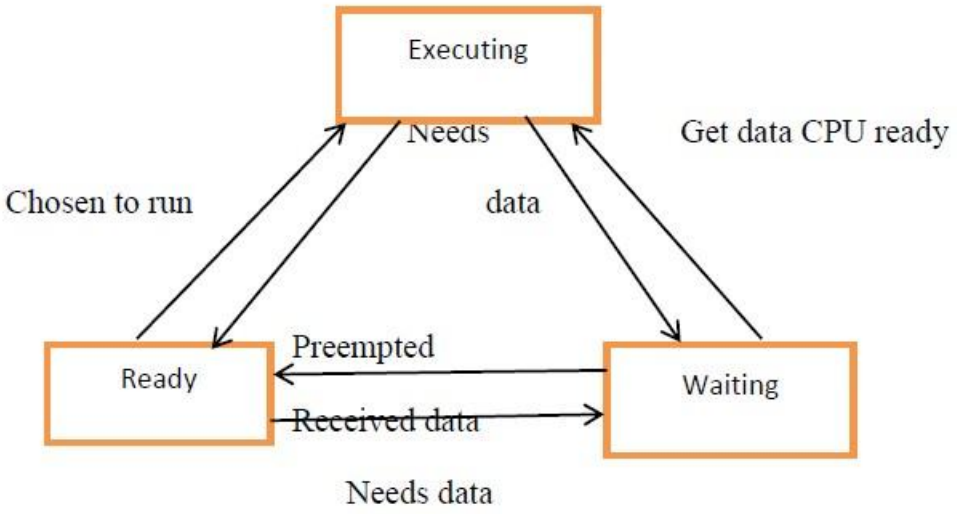
Multitasking evolves the execution switching among the different tasks. There should be same mechanism in place to show the CPU among the

different takes and to decide which process/task is to be executed at a given point of time. Determining which task/process is to be executed at a give point of time is known as task/process scheduling. Task

schedulingfrom the basis of multitasking Scheduling policies from the guidelines for determining which task is to be executed when.

The work of choosing the order of running process is known as scheduling. A scheduling policy defines how processes are selected for

promotion from the ready state to the running sate. The process scheduling decision may take place when a process Switches its states Ready state from Running state Blocked/wait sate from Running state Ready state from Blocked/wait state Completed (executed) state



Scenario 1 is premitive:a process swithches to ready state from the running state

Scenario 2 is scheduling under can be either pre-emptive or non-pre- emptiveWhen a high priority process in the blocked/wait state complete it's I/O and switches to ready state. The scheduler picks it for execution if the scheduling policy used is priority based pre-emptive.

Semaphores

A **semaphore**, in its most basic form, is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment. The two most common kinds of semaphores are **counting semaphores** and **binary semaphores**. Counting semaphores represent multiple resources, while binary semaphores, as the name implies, represents two possible states (generally 0 or 1; locked or unlocked). Semaphores were invented by the late Edsger Dijkstra.

Semaphores can be looked at as a representation of a limited number of resources, like seating capacity at a restaurant. If a restaurant has a capacity of 50 people and nobody is there, the semaphore would be initialized to 50. As each person arrives at the restaurant, they cause the seating capacity to decrease, so the semaphore in turn is decremented. When the maximum capacity is reached, the semaphore will be at zero, and nobody else will be able to enter the restaurant. Instead the hopeful restaurant goers must wait until someone is done with the resource, or in this analogy, done eating. When a patron leaves, the semaphore is incremented and the resource becomes available again.

A semaphore can only be accessed using the following operations: **wait()** and **signal()**. **wait()** is called when a process wants access to a resource. This would be equivalent to the arriving customer trying to get an open table. If there is an open table, or the semaphore is greater than

zero, then he can take that resource and sit at the table. If there is no open table and the semaphore is zero, that process must wait until it becomes available. **signal()** is called when a process is done using a resource, or when the patron is finished with his meal. The following is an implementation of this **counting semaphore** (where the value can be greater than 1):

```
wait(Semaphore s){
   while (s==0);    /* wait until s>0 */
   s=s-1;
}

signal(Semaphore s){
   s=s+1;
}

Init(Semaphore s , Int v){
   s=v;
}
```

Historically, **wait()** was called **P** (for Dutch "Proberen" meaning *to try*) and **signal()** was called **V** (for Dutch "Verhogen" meaning *to increment*). The standard Java library instead uses the name "acquire" for **P** and "release" for **V**.

No other process can access the semaphore when P or V are executing. This is implemented with **atomic** hardware and code. An atomic operation is indivisible, that is, it can be considered to execute as a unit.

If there is only one count of a resource, a **binary semaphore** is used which can only have the values of 0 or 1. They are often used as **mutex locks**. Here is an implementation of mutual-exclusion using binary semaphores:

```
do
{
   wait(s);
   // critical section
   signal(s);
   // remainder section
} while(1);
```

In this implementation, a process wanting to enter its critical section it has to acquire the binary semaphore which will then give it mutual exclusion until it signals that it is done.

For example, we have semaphore s, and two processes, P1 and P2 that want to enter their critical sections at the same time. P1 first calls wait(s). The value of s is decremented to 0 and P1 enters its critical section. While P1 is in its critical section, P2 calls wait(s), but because the value of s is zero, it must wait until P1 finishes its critical section and executes signal(s). When P1 calls signal, the value of s is incremented to 1, and P2 can then proceed to execute in its critical section (after decrementing the semaphore again). Mutual exclusion is achieved because only one process can be in its critical section at any time.

Mutex

In computer programming, a ***mu****tual* ***ex****clusion object (mutex)* is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

Message queues

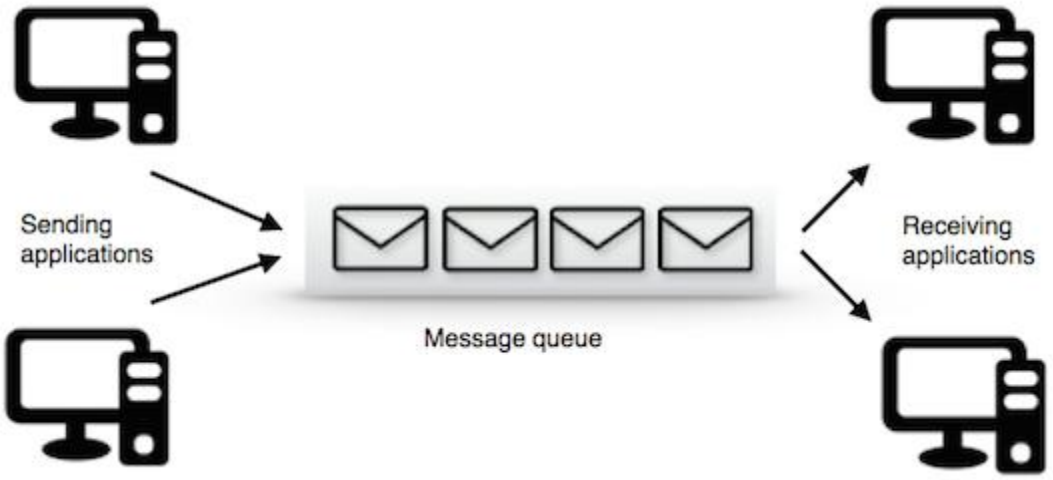
A message queue provide an **asynchronous communications protocol,** a system that puts a message onto a message queue does not require an immediate response to continue processing. Email is probably the best example of asynchronous messaging. When an email is sent can the sender continue processing other things without an immediate response from the receiver? This way of handling messages **decouples** the producer from the consumer. The producer and the consumer of the message do not need to interact with the message queue at the same time.

DECOUPLING AND SCALABILITY

Decoupling is used to describe how much one piece of a system relies on another piece of the system. Decoupling is the process of separating them so that their functionality will be more self contained.

A decoupled system are achieved when two or more systems are able to communicate without being connected. The systems can remain completely autonomous and unaware of each other. Decoupling is often a sign of a computer system that is well structured. It is usually easier to maintain, extend and debug.

If one process in a decoupled system fails processing messages from the queue, other messages can still be added to the queue and be processed when the system has recovered. You can also use a message queue to delay processing; A producer post messages to a queue. At the appointed time, the receivers are started up and process the messages in the queue. Messages in queue can be stored-and-forwarded and the message be redelivered until the message is processed.



Instead of building one large application, is it beneficial to decouple different parts of your application and only communicate between them asynchronously with messages. That way different parts of your application can evolve independently, be written in different languages and/or maintained by separated developer teams.

A message queue will keep the processes in your application separated and independent of each other. The first process will never need to invoke another process, or post notifications to another process, or follow the process flow of the other processes. It can just put the message on the queue and then continue processing. The other processes can also handle their work independently. They can take the messages from the queue when they are able to process them. This way of handling messages creates a system that is easy to maintain and easy to scale.

Message queuing - a simple use case

Imagine that you have a web service that receives many requests every second, where no request is afford to get lost and all requests needs to be processed by a process that is time consuming.

Imagine that your web service always has to be highly available and ready to receive new request instead of being locked by the processing of previous received requests. In this case it is ideal to put a queue between the web service and the processing service. The web service can put the "start processing"-message on a queue and the other process can take and handle messages in order. The two processes will be decoupled from each other and does not need to wait for each other. If you have a lot of requests coming in a short amount of time, the processing system will be able to process them all anyway. The queue will persist requests if their number becomes really huge.

You can then imagine that your business and your workload is growing and you need to scale up your system. All that is needed to be done now is to add more workers, receivers, to work off the queues faster.