Basava Dhanne : **basu.dhanne@gmail.com**

 **Asst.prof CMREC Hyderabad**

 **EMBEDDED SYSTEMS**

**(B.Tech IV–ISemester)**

**UNIT-I**

**Embedded Computer Systems** : An embedded computersystemis an electronic system,which

includes a microcomputer like the Stellaris LM3S1968 .It is configured to perform a specific dedicated application .Software is programmed into ROM .This software is not accessible to the user of the device , and software solves only a limited range of problems .Here the microcomputer is embedded or hidden inside the system.

Each **embedded microcomputer** system , accepts inputs, performs calculations, and generates outputs and runs in “real time.”

For Example a typical automobile now a days contains an average of ten microcontrollers. In fact, modern houses may contain as many as 150 microcontrollers and on average a consumer now interacts with microcontrollers up to 300 times a day. General areas that employ embedded microcomputers encompass every field of engineering namely : Communications, automotive, military, medical, consumer, machine control etc...

Characteristics of an Embedded systems:

* Speed (bytes/sec)
* Power (watts)
* Size (cm3) and weight (g)
* Accuracy (% error)
* Adaptibility.

So,an embedded system must perform the operations at a high speed so that it can be readily used for real time applications and its power consumption must be very low and the size of the system should be as for as possible small and the readings must be accurate with minimum error .The system must be easily adaptable for different situations.

**Software Issues :** The important software issues related to the embedded system are mentioned below.

* Software maintenance is extremely important.
* Verification of proper operation,
* Updates for the software in periodic intervals is very important.
* Fixing the bugs in the software improves its efficiency and also a very important factor.
* Adding features, New features must be added to the software when ever they are available
* Extending to new applications, the software must be upgraded such that its applicability increases for new application areas.
* Change user configurations .This is an important factor to improve the popularity of the software.

**APPLICATIONS :** Embedded systems find wide variety of applications in various fields.They are given below

* Automobile
* Aeronautics
* Space
* Rail Transport
* Mobile communications
* Industrial processing
* Remote sensing ,Radio and Networking
* Robotics
* Consumer electronics,music players, Computer applications
* Security (e-commerce, smart cards)
* Medical electronics (hospital equipment, and mobile monitoring) and
* Defense application

**Memory-mapped Architecture:** A memory mapped architecture is one whereI/O devices are connected like memory and I/O devices are assigned addresses, and the software accesses the I/O devices using these addresses. Software inputs from an input device same instructions as a memory read and software outputs from an output device same instructions as a memory write.

The memory mapped architecture is shown below in the case of ARM processor.



The system bus contains three busses. A bus is nothing but group of (8 or 16) lines which carry address (input, output, RAM or ROM), or data, and control information. Based on this they are known as address bus, data bus and control buses. Here the data bus is always a bidirectional bus.The address specifies which slave module will communicate with the processor one address per memory access cycle The data contains information that is being transferred. Control signals specify the direction of the transfer. One complete data transfer is known as a **bus cycle**. The processor always controls the address (where to access), the direction (read or write), and the control (when to access.)

**MC68HCII Architecture :**

 Motorola Inc ,one of the pioneers in microcontroller manufacturing has introduced this 8-bit microcontroller M68HC11 in the year 1985 and it is descended from the Motorola 6800 microprocessor. Now it is produced by Freescale Semiconductors. It is a CISC microcontroller , optimized for low power consumption and high-performance operation at bus frequencies up to 4 MHz . The 68HC11 chip has built-in EEPROM/OTPROM, RAM, digital I/O, timers, A/D converter, PWM generator, and synchronous and asynchronous communications channels (RS232 and SPI). Typical current draw is less than 10mA. The 68HC11 devices are more powerful and more expensive than the 68HC08 microcontrollers, and are used in barcode readers, hotel card key writers, amateur robotics, and various other embedded systems. The MC68HC11A8 was the first MCU to include CMOS EEPROM.

**OPERATING MODES:**

The 6811 can operate in any on of the four modes . They are Single-chip mode: uses internal memory for program & data. Expanded mode : allows for use of external memory. Bootstrap mode: used to load programs into RAM. Test mode : used by Motorola to test the chip is operation. These modes are selected by two pins MODE A & MODE B

 **Single chip (MODA=0, MODB=1)**

 No external address and data bus functions CPU can only access on-chip memory ii.Ports B and C are general purpose parallel I/O iii.All software needed to control MCU must be in internal memory iv. On reset, execution begins at address #E000

**Expanded multiplexed (MODA=MODB=1)**

 External memory and peripheral devices can be accessed by time-multiplexed address-data bus ii.Port B used for high byte of address (output) iii.Port C provides low byte of address (output) and 8- bit data (bi-directional) iv. External address latch is required v. Execution begins at address #E000

 **Bootstrap (MODA=MODB=0)**

 On power up or reset, the program in the bootstrap ROM is executed ii.CPU waits for a 256-byte program segment to be downloaded through the serial link and stored starting at address #0000 iii.Execution then begins at address $0000 iv.Permits wide variety of programs to be downloaded

**Test Mode (MODA=1, MODB=0)**

 Primarily used to test the chip by the manufacturer ii. Overrides some automatic protection mechanisms

**SALIENT FEATURES :**

 The MC68HC11 is HCMOS based advanced 8-bit MCU with numerous on-chip peripheral capabilities. Up to 10MIPS Throughput at 10MHz 256 Bytes of RAM , 512 Bytes of In-System Programmable EEPROM and Programming Lock. Eight channel 8-bit Analog to Digital Convertor One serial peripheral interface, with a speed up to 1M.The MC68HC11 is available in two packages . One is 48-pin dual inline package (DIP) and the other is the 52 Pin Plastic Leaded Chip Carrier(PLCC) known as Lead quad pack. In the 48 pin DIP package 38 pins are available for I/O functions.(34 I/O lines+ 2 interrupt lines + 2 hand shake control lines). Similarly in a 52 PLCC pack 42 pins are meant for different I/O functions, and the remaining are used for interrupt and handshake signals. MC68HC11 has one universal Asynchronous Serial Communications Interface (UART) One Watchdog Timer One 16-Bit free running timer, with 3 capture functions and 5 compare functions One Pulse Accumulator and Powerful bit-manipulation instructions. Six powerful addressing modes (Immediate, Extended, Indexed, Inherent and Relative) Power saving STOP and WAIT modes Memory mapped I/O and special functions.

**ARCHITECTURE**

It is based on HCMOS Technology and has a common internal bus for the address and data of 8-bits. It has an MCU clock whose frequency can be educed to zero. As the MCU is completely MOSFET based the power dissipation is negligible in stop or start states. So,this is optimized for lowpower consumption and high performance operation.



 **Block Diagram Of MC68HC11**

 **CPU FEATURES** :

 An 8M.Hz XTAL(external clock) with 2 M.Hz clock related operations. A 16-bit program counter that loads a powerup value from a reset vector address 0xFFFE – 0xFFFF Two 8- bit Accumulators A and B work as general purpose registers. They can be concatenated as a 16-bit double accumulator [D]. Two 16-bit Index registers Ix and Iy can be used as pointers to memory locations and hold the 16 bit addresses of memory locations. It has a multiplexed address and data bus. One 16 –bit stack pointer ,which decreases by 1 after the push of each byte. Two external interrupts IRQ , XIRQ .One of the is can be configured as non- maskable external interrupts like NMI in 80196. Although this is an 8-bit processor ,it has some 16-bit instructions.( ADD, Sub, shift and rotate) .

 **REGISTER ORGANISATION**

 The MC68HC11 microcontroller has a rich set of registers and they are classified into two categories : CPU registers and I/O registers. The CPU Registers are shown in the next slide . A and B are the two 8-bit registers called general purpose accumulators which are used to perform most of the arithmetic operations. These two accumulators can be concatenated to form a 16-bit accumulator which is known as double accumulator “D”. This accumulator is used for 16 bit operations . Index registers (IX and IY).Two 16-bit registers used mainly in addressing memory operands. They normally used to hold addresses of 16-bit memory locations.These registers are also , some times used for them for 16-bit computation also. Stack Pointer (SP):Stack is a first in first out data structure.This 16-bit stack pointer register hold the address of the stack top.

**Program Counter(PC):** It is a 16-bit register which stores the address of the next instruction to be executed. The 68HC11 fetches the instruction one byte at a time and increments the PC by 1 after after fetching each instruction byte. After the execution of an instruction the PC is incremented by the number of bytes of the executed instruction.

**Condition Code Register (CCR**) : This is an 8-bit register used to keep track of the program execution status , control the execution of conditional branch instructions and enable/disable the interrupt handling . This register contains five status indicators, two interrupt masking bits, and a S TOP disable bit. The register is named for the five status bits since that is the major use of the

register . These status flags reflect the results of arithmetic and other operations of the CPU as it performs instructions. The five flags are half carry (H),negative (N), zero (Z), overflow (V), and carry/borrow (C). The half-carry flag, which is used only for BCD arithmetic operations is only affected by the add accumulators A and B (ABA), ADD, and add with carry (ADC) addition instructions.



**Fig: REGISTER ORGANISATION**

The N, Z, V, and C status bits allow for branching based on the results of a previous operation. Simple branches are included for either state of any of these four bits. The H bit indicates a carry from bit 3 during an addition operation. This status indicator allows the CPU to adjust the result of an 8-bit BCD addition so it is in correct BCD format, even though the add was a binary operation. This H bit, which is only updated by the ABA, ADD, and ADC instructions, is used by the DAA instruction to compensate the result in accumulator A to correct BCD format . The N bit reflects the state of the most significant bit (MSB) of a result. For twos complement, a number is negative when the MSB is set and positive when the MSB is 0. The N bit has uses other than in twos-complement operations. By assigning an often tested flag bit to the MSB of a

register or memory location, the user can test this bit by loading an accumulator.The Z bit is set when all bits of the result are 0s. Compare instructions do an internal implied subtraction, and the condition codes, including Z, reflect the results of that subtraction. A few operations (INX, DEX, INY, and DEY) affect the Z bit and no other condition flags.The C bit is normally used to indicate if a carry from an addition or a borrow has occurred as a result of a subtraction. The C bit also acts as an error flag for multiply and divide operations. Shift and rotate instructions operate with and through the carry bit to facilitate multiple-word shift operations . The STOP disable (S) bit is used to allow or disallow the STOP instruction. Some users consider the STOP instruction dangerous because it causes the oscillator to stop; however, the user can set the S bit in the CCR to disallow the STOP instruction. If the STOP instruction is encountered by the CPU while the S bit is set, it will be treated like a no-operation (NOP) instruction, and processing continues to the next instruction. The interrupt request (IRQ) mask (I bit) is a global mask that disables all maskable interrupt sources. While the I bit is set, interrupts can become pending and are remembered, but CPU operation continues uninterrupted until the I bit is cleared. After any reset, the I bit is set by default and can be cleared only by a software instruction. When any interrupt occurs, the I bit is automatically set after the registers are stacked but before the interrupt vector is fetched. After the interrupt has been serviced, an RTI instruction is normally executed, restoring the registers to the values that were present before the interrupt occurred. The XIRQ mask (X bit) is used to disable interrupts from the XIRQ pin. After any reset, X is set by default and can be cleared only by a software instruction.

**Addressing Modes:**

Addressing modes are used to specify the operands needed in an instruction. The M68HC11 CPU supports SIX addressing modes. They are Immediate addressing mode Direct addressing Extended addressing Indexed (with either of two 16-bit index registers and an 8-bit offset) Inherent and Relative addressing mode.Each of the addressing modes (except inherent) results in an internally generated, double-byte value referred to as the effective address. This value appears on the address bus during the external memory reference portion of the instruction All bit-manipulation instructions use immediate addressing to fetch a bit mask, and branch variations use relative addressing mode to determine a branch destination

**Immediate Addressing (IMM):**

Immediate (IMM) In the immediate addressing mode, the actual argument is contained in the byte(s) immediately following the instruction in which the number of bytes matches the size of the register. These instructions are two, three, or four (if pre byte is required) bytes. In this case, the effective address of the instruction is specified by the character # sign and implicitly points to the byte following the opcode . The immediate value is limited to either one or two bytes, depending on the size of the register involved in the instruction.

**Examples :**

 LDAA #22 ; loads the decimal value 22 into the accumulator A.

ADDA #@32 ; adds the octal value 32 to accumulator A.

LDAB #$17 ; loads the hex value 17 into accumulator B.

LDX #$1000 ; loads the hex value 1000 into the index register X, where the upper byte of X receives the value of $10 and the lower byte of X gets the value of $00.

 **Character prefixes:**

|  |  |  |
| --- | --- | --- |
| S.No | Prefix | Definition |
| 1 | None | Decimal |
| 2 | $ | Hexadecimal |
| 3 | @ | Octal |
| 4 | % | Binary |
| 5 | **’** | Single ASCII character |

**Direct Mode (DIR):**

 In the direct addressing mode, the least significant byte of the effective address of the instruction operand appears in the byte following the opcode The high-order byte of the effective address is assumed to be $00 and is not included in the instruction. This limits use of the direct mode to operands in the $0000-$00FF area of the memory.

**Examples:**

ADDA $00 ; adds the value stored at the memory location with the effective address $0000 to accumulator A.

SUBA $ 20 ; subtracts the value stored at the memory location whose address is $0020 from accumulator A.

 LDD $10 ; loads the contents of the memory locations at $0010 and $0011 into double accumulator D, where the contents of the memory location at $0010 are loaded into accumulator A and those of the memory location at $0011 are loaded into accumulator B.

**Extended Mode (EXT):**

 In the extended addressing mode, the effective address of the operand appears explicitly in the two bytes following the op code.

 EX: LDAA $ 1003 ; loads the 8-bit value stored at the memory location with effective address $1003 into accumulator A.

LDX $ 1000 ; loads the 16-bit value stored at the memory locations with the effective addresses $1000 and $1001 into the index register X. The byte at $1000 will be loaded into the upper byte of X and the byte at $1001 will be loaded into the lower byte of X.

 ADDD $1030 ; adds the 16-bit value stored at the memory locations with the effective addresses $1030 and $1031 to double accumulator D.

**Indexed Mode (INDX, INDY)**

 In the indexed addressing mode, one of the index registers (X or Y) is used in calculating the effective address. So, the effective address is variable and depends on the current contents of the index register X (or Y) and a fixed, 8-bit unsigned offset contained in the instruction. Because the offset byte is unsigned, only positive offsets in the range from 0 to 255 can be represented. If no offset is specified, the machine code will contain $00 in the offset byte.

**Examples:**

1. ADDA 10,X ; adds the value stored at the memory location pointed to by the sum of 10 and the contents of the index register X to accumulator A. Each of the following instructions subtracts the value stored at the memory location pointed to by the contents of index register X from accumulator A .

Ex.2. SUBA 0,X

Ex.3. SUBA ,X

**Inherent Mode (INH):**

In the inherent mode, everything needed to execute the instruction is encoded in the opcode . The operands are CPU registers and thus are not fetched from memory. These instructions are usually one or two bytes.

Exs : ABA ; adds the contents of accumulator B to accumulator A.

INCB ; increments the value of accumulator B by 1.

 INX ; increments the value of the index register X by 1.

**Relative Mode (REL) :**

Relative Mode (REL) The relative addressing mode is used only for branch instructions. Branch instructions, other than the branching versions of the bit- manipulation instructions, generate two machine-code bytes, one for the opcode and one for the branch offset. The branch offset is the distance relative to the first byte of the instruction immediately following the branch instruction. The branch offset has a range of 128 to 127 bytes.

**Example:**

 BEQ $e164

 $e100 ADDA #10

 ……………..

 $e164 DECB

 …………..

The 68HC11 will branch to execute the instruction DECB if the Z bit in the CCR register is 1, when the instruction BEQ $e164 is executed.

**UNIT II**

**8051 MICRCONTROLLER**

The 8051 microcontroller is a very popular 8-bit microcontroller introduced by Intel in the year 1981 and it has become almost the academic standard now a days. The 8051 is based on an 8-bit CISC core with Harvard architecture. Its 8-bit architecture is optimized for control applications with extensive Boolean processing. It is available as a 40-pin DIP chip and works at +5 Volts DC. The salient features of 8051 controller are given below.

**SALIANT FEATURES:** The salient features of 8051 Microcontroller are

i. 4 KB on chip program memory (ROM or EPROM)).

ii. 128 bytes on chip data memory(RAM).

iii. 8-bit data bus

iv. 16-bit address bus

v. 32 general purpose registers each of 8 bits

vi. Two -16 bit timers T0 and T1

vii. Five Interrupts (3 internal and 2 external).

ix. Four Parallel ports each of 8-bits (PORT0, PORT1,PORT2,PORT3) with a total of 32 I/O

 lines.

x. One 16-bit program counter and One 16-bit DPTR ( data pointer)

xi. One 8-bit stack pointer

xii. One Microsecond instruction cycle with 12 MHz Crystal.

xiii. One full duplex serial communication port.

**ARCHITECTURE & BLOCK DIAGRAM OF 8051 MICROCONTROLLER:**

The architecture of the 8051 microcontroller can be understood from the block diagram. It has Harward architecture with RISC (Reduced Instruction Set Computer) concept. The block diagram of 8051 microcontroller is shown in fig below1.It consists of an 8-bit ALU, one 8-bit PSW(Program Status Register), A and B registers , one 16-bit Program counter , one 16-bit Data pointer register(DPTR),128 bytes of RAM and 4kB of ROM and four parallel I/O ports each of 8-bit width.

****

**Fig.1. Block Diagram of 8051 Microcontroller**

8051 has 8-bit ALU which can perform all the 8-bit arithmetic and logical operations in one machine cycle. The ALU is associated with two registers A & B

**A and B Registers** : The A and B registers are special function registers which hold the results of many arithmetic and logical operations of 8051.The A register is also called the **Accumulator** and as it’s name suggests, is used as a general register to accumulate the results of a large number of instructions. By default it is used for all mathematical operations and also data transfer operations between CPU and any external memory.

 The B register is mainly used for multiplication and division operations along with A register.

 MUL AB : DIV AB.

It has no other function other than as a location where data may be stored.

**The R registers**: The "R" registers are a set of eight registers that are named R0, R1, etc. up to

 and including R7. These registers are used as auxillary registers in many operations. The "R" registers are also used to temporarily store values.

**Program Counter(PC) :** 8051 has a 16-bit program counter .The program counter always points to the address of the next instruction to be executed. After execution of one instruction the program counter is incremented to point to the address of the next instruction to be executed.It is the contents of the PC that are placed on the address bus to find and fetch the desired instruction.Since the PC is 16-bit width ,8051 can access program addresses from 0000H to FFFFH ,a total of 6kB of code.

**Stack Pointer Register (SP) :** It is an 8-bit register which stores the address of the stack top. i.e the Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value is pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location. Similarly when a value is popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP. Since the SP is only 8-bit wide it is incremented or decremented by two . SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered.

**STACK in 8051 Microcontroller :** The stack is apart of RAM used by the CPUto store information temporarily. This information may be either data or an address .The CPU needs this storage area as there are only limited number of registers. The register used to access the stack is called the Stack pointer which is an 8-bit register..So,it can take values of 00 to FF H.When the 8051 is powered up ,the SP register contains the value 07.i.e the RAM location value 08 is the first location being used for the stack by the 8051 controller

 There are two important instructions to handle this stack.One is the PUSH and the Other is the POP. The loading of data from CPU registers to the stack is done by PUSH and the loading of the contents of the stack back into aCPU register is done by POP.

EX : MOV R6 ,#35 H

 MOV R1 ,#21 H

 PUSH 6

 PUSH 1

 In the above instructions the contents of the Registers R6 and R1 are moved to stack and they occupy the 08 and 09 locations of the stack.Now the contents of the SP are incremented by two and it is 0A

Similarly POP 3 instruction pops the contents of stack into R3 register.Now the contents of the SP is decremented by 1

In 8051 the RAM locations 08 to 1F (24 bytes) can be used for the Stack.In any program if we need more than 24 bytes of stack ,we can change the SP point to RAM locations 30-7F H.this can be done with the instruction MOV SP,# XX.

**Data Pointer Register(DPTR) :** It is a 16-bit register which is the only user-accessible. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR. This DPTR can also be used as two 8-registers DPH and DPL.

**Program Status Register (PSW) :** The 8051 has a 8-bit PSW registerwhich is alsoknown as Flag register.In the 8-bit register only 6-bits are used by 8051.The two unused bits are user definable bits.In the 6-bits four of them are conditional flags .They are Carry –CY,Auxiliary Carry-AC, Parity-P,and Overflow-OV .These flag bits indicate some conditions that resulted after an instruction was executed.



The bits PSW3 and PSW4 are denoted as RS0 and RS1 and these bits are used th select the bank registers of the RAM location. The meaning of various bits of PSW register is shown below.

CY PSW.7 Carry Flag

AC PSW.6 Auxiliary Carry Flag

FO PSW.5 Flag 0 available for general purpose .

RS1 PSW.4 Register Bank select bit 1

RS0 PSW.3 Register bank select bit 0

OV PSW.2 Overflow flag

--- PSW.1 User difinable flag

P PSW.0 Parity flag .set/cleared by hardware.

The selection of the register Banks and their addresses are given below. 

**Memory organization :** The 8051 microcontroller has 128 bytes of Internal RAM and 4kB of on chip ROM .The RAM is also known as Data memory and the ROM is known as program memory. The program memory is also known as Code memory .This Code memory holds the actual 8051 program that is to be executed. In 8051 this memory is limited to 64K .Code memory may be found on-chip, as ROM or EPROM. It may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. The 8051 has only 128 bytes of Internal RAM but it supports 64kB of external RAM. As the name suggests, external RAM is any random access memory which is off-chip. Since the memory is off-chip it is not as flexible interms of accessing, and is also slower. For example, to increment an Internal RAM location by 1,it requires only 1 instruction and 1 instruction cycle but to increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. So, here the external memory is 7 times slower.

**Internal RAM OF 8051 :**

This Internal RAM is found on-chip on the 8051 .So it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it’s contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared. The 128 bytes of internal RAM is organized as below.

(i) Four register banks (Bank0,Bank1, Bank2 and Bank3) each of 8-bits (total 32 bytes). The default bank register is Bank0. The remaining Banks are selected with the help of RS0 and RS1 bits of PSW Register.

(ii) 16 bytes of bit addressable area and

(iii) 80 bytes of general purpose area (Scratch pad memory) as shown in the diagram below. This area is also utilized by the microcontroller as a storage area for the operating stack.



The 32 bytes of RAM from address 00 H to 1FH are used as working registers organized as four banks of eight registers each.The registers are named as R0-R7 .Each register can be addressed by its name or by its RAM address.

 For EX : MOV A, R7 or MOV R7,#05H

**Internal ROM (On –chip ROM):** The 8051 microcontroller has 4kB of on chip ROM but it can be extended up to 64kB.This ROM is also called program memory or code memory. The CODE segment is accessed using the program counter (PC) for opcode fetches and by DPTR for data. The external ROM is accessed when the EA(active low) pin is connected to ground or the contents of program counter exceeds 0FFFH.When the Internal ROM address is exceeded the 8051 automatically fetches the code bytes from the external program memory.



**SPECIAL FUNCTION REGISTERS (SFRs) :** In 8051 microcontroller there certainregisters which uses the RAM addresses from 80h to FFh and they are meant for certain specific operations .These registers are called Special function registers (SFRs).Some of these registers are bit addressable also. The list of SFRs and their functional names are given below. In these SFRs some of them are related to I/O ports (P0,P1,P2 and P3) and some of them are meant for control operations (TCON,SCON, PCON..) and remaining are the auxillary SFRs, in the sense that they don't directly configure the 8051.

|  |  |  |  |
| --- | --- | --- | --- |
|  **S.No** | **Symbol** | **Name of SFR** | **Address (Hex)** |
| 1 | ACC\* | Accumulator | **0E0** |
| 2 | B\* | B-Register | **0F0** |
| 3 | PSW\* | Program Status word register | **0DO** |
| 4 | SP | Stack Pointer Register | **81** |
| 5 | DPTR | DPL  | Data pointer low byte | **82** |
| DPH | Data pointer high byte | **83** |
| 6 | P0\* | Port 0 | **80** |
|  | P1\* | Port 1 | **90** |
| 8 | P2\* | Port 2 | **0A** |
| 9 | P3\* | Port 3 | **0B** |
| 10 | IP\* | Interrupt Priority control | **0B8** |
| 11 | IE\* | Interrupt Enable control | **0A8** |
| 12 | TMOD | Tmier mode register | **89** |
| 13 | TCON\* | Timer control register | **88** |
| 14 | TH0 | Timer 0 Higher byte | **8C** |
| 15 | TL0 | Timer 0 Lower byte | **8A** |
| 16 | TH1 | Timer 1Higher byte | **8D** |
| 17 | TL1 | Timer 1 lower byte | **8B** |
| 18 | SCON\* | Serial control register | **98** |
| 19 | SBUF | Serial buffer register | **99** |
| 20 | PCON | Power control register | **87** |

**The** \* **indicates the bit addressable SFRs**

 **Table:SFRs of 8051 Microcontroller**

 **General Parallel I/O Ports :**

The 8051 microcontroller has four parallel I/O ports , each of 8-bits .So, it provides the user 32 I/O lines for connecting the microcontroller to the peripherals. The four ports are P0 (Port 0), P1(Port1) ,P2(Port 2) and P3 (Port3). Upon reset all the ports are output ports. In order to make them input, all the ports must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

Ex: MOV A,#0FFH ; A = FF

 MOV P0,A ; make P0 an input port

**PORT 0:**

Port 0 is an 8-bit I/O port with dual purpose. If external memory is used, these port pins are used for the lower address byte address/data (AD0-AD7), otherwise all bits of the port are either input or output.. Unlike other ports, Port 0 is not provided with pull-up resistors internally ,so for PORT0 pull-up resistors of nearly 10k are to be connected externally as shown in the fig.2.

**Dual role of port 0**: Port 0 can also be used as address/data bus(AD0-AD7), allowing it to be used for both address and data. When connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save the pins. ALE indicates whether P0 has address or data. When ALE = 0, it provides data D0-D7, and when ALE =1 it provides address and data with the help of a 74LS373 latch.



**Port 1:** Port 1 occupies a total of 8 pins (pins 1 through 8). It has no dual application and acts only as input or output port. In contrast to port 0, this port does not need any pull-up resistors since pull-up resistors connected internally. Upon reset, Port 1 is configured as an output port. To configure it as an input port , port bits must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”. For Ex :

 MOV A, #0FFH ; A=FF HEX

 MOV P1,A ; make P1 an input port by writing 1’s to all of its pins

**Port 2 :** Port 2 is also an eight bit parallel port. (pins 21- 28). It can be used as input or output port. As this port is provided with internal pull-up resistors it does not need any external pull-up resistors. Upon reset, Port 2 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A=FF hex

 MOV P2, A ; make P2 an input port by writing all 1’s to it

**Dual role of port 2** : Port2 lines are also associated with the higher order address lines A8-A15. In systems based on the 8751, 8951, and DS5000, Port2 is used as simple I/O port.. But, in 8031-based systems, port 2 is used along with P0 to provide the 16-bit address for the external memory. Since an 8031 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8031 is connected to external memory, Port 2 is used for the upper 8 bits of the 16 bit address, and it cannot be used for I/O operations.

**PORT 3** : Port3 is also an 8-bit parallel port with dual function.( pins 10 to 17). The port pins can be used for I/O operations as well as for control operations. The details of these additional operations are given below in the table. Port 3 also do not need any external pull-up resistors as they are provided internally similar to the case of Port2 & Port 1. Upon reset port 3 is configured as an output port . If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A= FF hex

 MOV P3, A ; make P3 an input port by writing all 1’s to it

 **Alternate Functions of Port 3 :** P3.0 and P3.1 are used for the RxD (Receive Data) and TxD (Transmit Data) serial communications signals. Bits P3.2 and P3.3 are meant for external interrupts. Bits P3.4 and P3.5 are used for Timers 0 and 1 and P3.6 and P3.7 are used to provide the write and read signals of external memories connected in 8031 based systems

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** |  **Port 3 bit** |  **Pin No** |  **Function** |
| 1 | P3.0 | 10 | RxD |
| 2 | P3.1 | 11 | TxD |
| 3 | P3.2 | 12 |  |
| 4 | P3.3 | 13 |  |
| 5 | P3.4 | 14 | T0 |
| 6 | P3.5 | 15 | T1 |
| 7 | P3.6 | 16 |  |
| 8 | P3.7 | 17 |  |

 **Table: PORT 3 alternate functions**

**Interrupt Structure** : An interrupt is an external or internal event that disturbs the microcontroller to inform it that a device needs its service. The program which is associated with the interrupt is called the **interrupt service routine** (ISR) or **interrupt handler**. Upon receiving the interrupt signal the Microcontroller , finish current instruction and saves the PC on stack. Jumps to a fixed location in memory depending on type of interrupt Starts to execute the interrupt service routine until RETI (return from interrupt)Upon executing the RETI the microcontroller returns to the place where it was interrupted. Get pop PC from stack

The 8051 microcontroller has **FIVE** interrupts in addition to Reset. They are

* Timer 0 overflow Interrupt
* Timer 1 overflow Interrupt
* External Interrupt 0(INT0)

* External Interrupt 1(INT1)
* Serial Port events (buffer full, buffer empty, etc) Interrupt

Each interrupt has a specific place in code memory where program execution (interrupt service routine) begins.

* External Interrupt 0: 0003 H
* Timer 0 overflow: 000B H
* External Interrupt 1: 0013 H
* Timer 1 overflow: 001B H
* Serial Interrupt : 0023 H

Upon reset all Interrupts are disabled & do not respond to the Microcontroller. These interrupts must be enabled by software in order for the Microcontroller to respond to them. This is done by an 8-bit register called Interrupt Enable Register (IE).

**Interrupt Enable Register :**



* EA : Global enable/disable. To enable the interrupts this bit must be set High.
* --- : Undefined-reserved for future use.
* ET2 : Enable /disable Timer 2 overflow interrupt.
* ES : Enable/disable Serial port interrupt.
* ET1 : Enable /disable Timer 1 overflow interrupt.
* EX1 : Enable/disable External interrupt1.
* ET0 : Enable /disable Timer 0 overflow interrupt.
* EX0 : Enable/disable External interrupt0

Upon reset the interrupts have the following priority.(Top to down). The interrupt with the highest PRIORITY gets serviced first.

* 1. External interrupt 0 (INT0)
	2. Timer interrupt0 (TF0)
	3. External interrupt 1 (INT1)
	4. Timer interrupt1 (TF1)
	5. Serial communication (RI+TI)

Priority can also be set to “high” or “low” by 8-bit IP register.- Interrupt priority register



IP.7: reserved

IP.6: reserved

IP.5: Timer 2 interrupt priority bit (8052 only)

IP.4: Serial port interrupt priority bit

IP.3: Timer 1 interrupt priority bit

IP.2: External interrupt 1 priority bit

IP.1: Timser 0 interrupt priority bit

IP.0: External interrupt 0 priority bit

 **TIMERS in 8051 Microcontrollers :** The 8051 microcontroller has two 16-bit timers Timer 0 (T0) and Timer 1(T1) which can be used either to generate accurate time delays or as event counters. These timers are accessed as two 8-bit registers TLO, THO & TL1 ,TH1 because the 8051 microcontroller has 8-bit architecture.

**TIMER 0 :** The Timer 0 is a 16-bit register and can be treated as two 8-bit registers (TL0 & TH0) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

Ex : The instruction Mov TL0,#07 moves the value 07 into lower byte of Timer0.

Similarly Mov R5,TH0 saves the contents of TH0 in the R5 register.

 

**TIMER 1 :** The Timer 1 is also a 16-bit register and can be treated as two 8-bit registers (TL1 & TH1) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

Ex : The instruction MOV TL1,#05 moves the value 05 into lower byte of Timer1.

Similarly MOV R0,TH1 saves the contents of TH1 in the R0 register



**TMOD Register :** The various operating modes of both the timers T0 and T1 are set by an 8-bit register called TMOD register**.** In this TMOD register the lower 4-bits are meant for Timer 0 and the higher 4-bits are meant for Timer1.



**GATE**: This bit is used to start or stop the timers by hardware .When GATE= 1 ,the timers can be started / stopped by the external sources. When GATE= 0, the timers can be started or stopped by software instructions like SETB TR0 or SETB TR1

**C/T (clock/Timer) :** This bit decides whether the timer is used as delay generator or event counter. When **C/T = 0 ,**the Timer is used as delay generator and if C/T=1 the timer is used as an event counter. The clock source for the time delay is the crystal frequency of 8051.

**M1,M0 (Mode) :** These two bits are thetimer mode bits. The timers of the 8051 can be configured in three modes.Mode0, Mode1 and Mode2.The selection and operation of the modes is shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** |  **M0** |  **M1** |  **Mode** | **Operation** |
| 1 | 0 | 0 | 0 | 13-bit Timer mode8-bit Timer/counter. THx with TLx as 5-bit prescalar  |
| 2 | 0 | 1 | 1 | 16-bit Timer mode.16-bit timer /counter without pre-scalar |
| 3 | 1 | 0 | 2 | 8-bit auto reload. THx contains a value that is to be loaded into TLx each time it overflows |
| 4 | 1 | 1 | 3 | Split timer mode |

**PIN Diagram of 8051 Microcontroller :** The 8051 microcontroller is available as a 40 pin DIP chip and it works at +5 volts DC. Among the 40 pins , a total of 32 pins are allotted for the four parallel ports P0,P1,P2 and P3 i.e each port occupies 8-pins .The remaining pins are VCC, GND, XTAL1, XTAL2, RST, EA ,PSEN.

**XTAL1,XTAL2**: These two pins are connected to Quartz crystal oscillator which runs the on-chip oscillator. The quartz crystal oscillator is connected to the two pins along with a capacitor of 30pF as shown in the circuit. If we use a source other than the crystal oscillator, it will be connected to XTAL1 and XTAL2 is left unconnected.





**RST**: The RESET pin is an input pin and it is an active high pin. When a high pulse is applied to this pin the microcontroller will reset and terminate all activities. Upon reset all the registers except PC will reset to 0000 Value and PC register will reset to 0007 value.

** (External Access):** This pin isan active low pin. This pin is connected to ground when microcontroller is accessing the program code stored in the external memory and connected to Vcc when it is accessing the program code in the on chip memory. This pin should not be left unconnected.

**(Program Store Enable) :** This is an output pin which is active low. When the microcontroller is accessing the program code stored in the external ROM ,this pin is connected to the OE (Output Enable) pin of the ROM.

**ALE (Address latch enable):** This is an output pin, which is active high**.** When connected to external memory , port 0 provides both address and data i.e address and data are multiplexed through port 0 .This ALE pin will demultiplex the address and data bus .When the pin is High , the AD bus will act as address bus otherwise the AD bus will act as Data bus.

**P0.0- P0.7(AD0-AD7) :** The port 0 pins multiplexed with Address/data pins .If the microcontroller is accessing external memory these pins will act as address/data pins otherwise they are used for Port 0 pins.

**P2.0- P2.7(A8-A15) :** The port2 pins are multiplexed with the higher order address pins **.**When the microcontroller is accessing external memory these pins provide the higher order address byte otherwise they act as Port 2 pins.

**P1.0- P1.7 :**These 8-pins are dedicated for Port1 to perform input or output port operations.

**P3.0- P3.7 :**These 8-pins are meant for Port3 operations and also for some control operations like Read,Write,Timer0,Timer1 ,INT0,INT1 ,RxD and TxD

**ADDRESSING MODES OF 8051 :**

The way in which the data operands are accessed by different instructions is known as the addressing modes. There are various methods of denoting the data operands in the instruction. The 8051 microcontroller supports mainly **5** addressing modes. They are

1.Immediate addressing mode

2.Direct Addressing mode

3.Register addressing mode

4. Register Indirect addressing mode

5.Indexed addressing mode

**Immediate addressing mode :** The addressing mode in which the data operand is a constant and it is a part of the instruction itself is known as Immediate addressing mode. Normally the data must be preceded by a # sign. This addressing mode can be used to transfer the data into any of the registers including DPTR.

 Ex: MOV A , # 27 H : The data (constant) 27 is moved to the accumulator register

 ADD R1 ,#45 H : Add the constant 45 to the contents of the accumulator

 MOV DPTR ,# 8245H :Move the data 8245 into the data pointer register.

 MOV P1,#21 H

**Direct addressing mode**: The addressing mode in which the data operand is in the RAM location (00 -7FH) and the address of the data operand is given in the instruction is known as Direct addressing mode. The direct addressing mode uses the lower 128 bytes of Internal RAM and the SFRs

MOV R1, 42H : Move the contents of RAM location 42 into R1 register

MOV 49H,A : Move the contents of the accumulator into the RAM location 49.

ADD A, 56H : Add the contents of the RAM location 56 to the accumulator

**Register addressing mode** :The addressing mode in which the data operand to be manipulated lies in one of the registers is known as register addressing mode.

MOV A,R0 : Move the contents of the register R0 to the accumulator

ADD A,R6 :Add the contents of R6 register to the accumulator

MOV P1, R2 : Move the contents of the R2 register into port 1

MOV R5, R2 : This is invalid .The data transfer between the registers is not allowed.

**Register Indirect addressing mode :**The addressing mode in which a register is used as a pointer to the data memory block is known as Register indirect addressing mode.

MOV A,@ R0 :Move the contents of RAM location whose address is in R0 into **A** (accumulator)

MOV @ R1 , B : Move the contents of B into RAM location whose address is held by R1

When R0 and R1 are used as pointers, they must be preceded by @ sign

**One of the advantages of register indirect addressing mode is that it makes accessing the data more dynamic than static as in the case of direct addressing mode.**

**Indexed addressing mode :** This addressing mode is usedin accessing the data elements of lookup table entries located in program ROM space of 8051.

Ex : MOVC A,@ A+DPTR

The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Here C denotes code .In this instruction the contents of A are added to the 16-bit DPTR register to form the 16-bit address of the data operand.

**Interfacing of ADC 0804 to 8051 Microcontroller :**

ADC 0804 is a single channel analog to digital converter i.e., it can take only one analog signal. ADC 0804 has 8 bit resolution. The higher resolution ADC gives smaller step size. Step size is smallest change that can be measured by an ADC. For an ADC with resolution of 8 bits, the step size is 19.53mV (5V/255). The time taken by the ADC to convert analog data into digital form depends on the frequency of clock source. The conversion time of ADC 0804 is around 110us. To use the internal clock a capacitor and resistor are used as shown in the circuit. The input to the ADC is given from a regulated power supply and a 10K potentiometer

The 8051 Microcontroller is used to provide the control signals to the ADC. CS(chip select) pin of ADC is directly connected to ground. The pin P1.1, P1.0 and P1.2 are connected to the pin WR, RD and INTR of the ADC respectively. When the input voltage from the preset is varied the output of ADC varies also varies.

From the circuit it is clear that the ADC interfaced directly to the microcontroller. The Port1 is used as an input port which receives the digital data from the ADC.Port pins P2.5 and P2.6 are used for SOC and EOC operation.When the conversion is over the ADC will send an interrupt signal to the microcontroller through the pin P2.7 .Now the Microcontroller receives digital data through the Port1.This data after conversion to decimal data is displayed on the LCD module .



The assembly language program for ADC is given below .

 MOV P1 , 0FF H ; Make the port1 high and configure port1 as Input port

**BACK**: CLR P2.6 ; Generation of SOC pulse

 SETB P2.5 ;

 **LOOP** JB P2.7 , LOOP ; Wait for conversion, Is conversion over?

 CLR P2.5 ; Enable Read the digital data

 MOV A ,P1 ; Read digital data through Port1

 SETB P2.5 ; Disable read after read operation

 CALL DISPLAY ; Display the data on LCD module

 SJMP BACK ; Continue the conversion process

 **Stepper motor Interfacing**

  A stepper motor is a device that translates electrical pulses into mechanical movement. The stepper motor rotates in steps in response to the applied signals. It is used in applications such as disk drives, dot matrix printers, plotters and robotics.It is mainly used for position control. Stepper motors have a permanent magnet called rotor (also called the shaft) surrounded by a stator . There are also steppers called variable reluctance stepper motors that do not have a PM rotor. The most common stepper motors have four stator windings that are paired with a center-tapped. This type of stepper motor is commonly referred to as a. four-phase or unipolar stepper motor. The center tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator.



**Assembly Language Program**

mov stepper, #0CH

acall delay

mov stepper, #06H

acall delay

mov stepper, #03H

acall delay

mov stepper, #09H

acall delay

sjmp main

delay:

mov r7,#4

wait2:

mov r6,#0FFH

wait1:

mov r5,#0FFH

wait:

djnz r5,wait

djnz r6,wait1

djnz r7,wait2

ret

end

**References:**

1. Jonathan.W.Valvano, “Embedded Microcomputer system”, Brooks/COLE Thomson learning series.
2. Muhammed Ali Mazidi, Janice Gillies Pie Mazidi, “The 8051 Microcontroller and Embedded

 Systems”— Pearson EducationAsia.

**Unit III**

**8051 Programming**

**ADDRESSING MODES OF 8051 :**

The way in which the data operands are accessed by different instructions is known as the addressing mode. There are various methods of denoting the data operands in the instruction. The 8051 microcontroller provides five distinct addressing modes. They are

1. Immediate addressing mode

2. Register addressing mode

3. Direct Addressing mode

4. Register Indirect addressing mode

5. Indexed addressing mode

**Immediate addressing mode :** The addressing mode in which the data operand is a constant and it is a part of the instruction itself is known as Immediate addressing mode. Normally the data must be preceded by a # sign. This addressing mode can be used to transfer the data into any of the registers including DPTR.

MOV A, # 27 H ; The data (constant) 27 is moved to the accumulator register

ADD R1, # 45 H ; Add the constant 45 to the contents of the accumulator

MOV DPTR, # 8245H ; Move the data 8245 into the data pointer register.

MOV P1, # 21 H ; Move the data 21H to port1

**Direct addressing mode**: The addressing mode in which the data operand is in the RAM location (00 -7FH) and the address of the data operand is given in the instruction is known as Direct addressing mode. The direct addressing mode uses the lower 128 bytes of Internal RAM and the SFRs.

MOV R1, 42H ; Move the contents of RAM location 42 into R1 register

MOV 49H, A ; Move the contents of the accumulator into the RAM location 49

ADD A, 56H ; Add the contents of the RAM location 56 to the accumulator

**Register addressing mode** : The addressing mode in which the data operand to be manipulated lies in one of the registers is known as register addressing mode.

MOV A, R0 ; Move the contents of the register R0 to the accumulator

ADD A, R6 ; Add the contents of R6 register to the accumulator

MOV P1, R2 ; Move the contents of the R2 register into port 1

MOV R5, R2 ; This is invalid. The data transfer between the registers is not

 allowed.

**Register Indirect addressing mode :** The addressing mode in which a register is used as a pointer to the data memory block is known as Register indirect addressing mode.

MOV A, @ R0 ; Move the contents of RAM location whose address is in R0 into

A (accumulator)

MOV @ R1, B ; Move the contents of B into RAM location whose address is

 held by R1

When R0 and R1 are used as pointers, they must be preceded by @ sign

**One of the advantages of register indirect addressing mode is that it makes accessing the data more dynamic than static as in the case of direct addressing mode.**

**Indexed addressing mode :** This addressing mode is usedin accessing the data elements of lookup table entries located in program ROM space of 8051.

MOVC A, @ A+DPTR

The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Here C denotes code. In this instruction the contents of A are added to the 16-bit DPTR register to form the 16-bit address of the data operand.

**Instruction Set of 8051:**

The 8051 microcontroller provides the following groups of instructions.

1. Arithmetic instructions
2. Logical and Compare instructions
3. Single bit instructions
4. Loop, Jump and Call instructions
5. Data transfer instructions

**Arithmetic instructions :**

This group of operators perform arithmetic operations such as addition, subtraction, multiplication and division etc. Arithmetic operations affect the flags, such as Carry Flag (CY), Overflow Flag (OV) etc, in the PSW register.

**ADD (Addition of two 8 bit numbers) :**

 ADD A, Source ; A = A + Source

 ADD A, #data ; A= A + immediate data

 ADD A, Rn ; A = A + [ Rn ]

 ADD A, direct ; A = A + [Direct memory]

 ADD A, @Rn ; A = A + [Memory pointed to by Rn]

The instruction ADD is used to add two operands. The destination operand is always in register A while the source operand can be a register, immediate data or in memory.

Ex: ADD A, R1

 ADD A,#23

**ADDC (Addition of two 16 bit numbers with carry) :**

 ADDC A, Source ; A = A + Source + 1

 ADDC A, #data ; A= A + immediate data + CY

 ADDC A, Rn ; A = A + [ Rn ] + CY

 ADDC A, direct ; A = A + [Direct memory] + CY

 ADDC A, @Ri ; A = A + [Memory pointed to by Ri] + CY

The instruction ADDC (add with carry) is used to add two 16-bit numbers. It is used when a carry is propagated from lower byte to higher byte.

Ex: ADDC A, R6

 ADDC A, #45

Note : @Ri implies contents of memory location pointed to by R0 or R1 and Rn refers to registers R0-R7 of the currently selected register bank.

**DA (Decimal adjust for addition) :**

 DA A ; Adjust for BCD addition

This is a decimal adjust instruction. It adjusts the 8-bit value in ACC resulting from operations

like ADD or ADDC and produces two 4-bit digits (in packed Binary Coded Decimal (BCD) format). Effectively, this instruction performs the decimal conversion by adding 00H, 06H, 60H or 66H to the accumulator, depending on the initial value of ACC and PSW.

Ex: DA A

 **SUBB (Subtract with borrow) :**

 SUBB A, source ; A = A – source – CY

 SUBB A, #data ; A= A - immediate data - CY

 SUBB A, Rn ; A = A - [ Rn ] - CY

 SUBB A, direct ; A = A - [Direct memory] - CY

 SUBB A, @Ri ; A = A - [Memory pointed to by Ri] - CY

The SUBB subtracts the specified data byte and the carry flag together from the accumulator, leaving the result in the accumulator.

Ex: SUBB A, R4

 SUBB A, #65

**MUL (Multiplication):**

The 8051 microcontroller supports byte by byte multiplication only.

 MUL AB ; A x B, place 16-bit result in A and B

In byte by byte multiplication, one of the operands must be in register A, and the second operand must be in register B. After multiplication, the result is in the A and B registers, the lower byte is in A and the upper byte is in B.

Ex: MUL AB ; A = lower byte, B = higher byte

**DIV (Division) :**

In the division of unsigned numbers, the 8051 supports byte over byte only.

 DIV AB ; divide A by B

In dividing a byte by a byte, the numerator must be in register A and the denominator must be in B. After the DIV instruction is performed, the quotient is in A and the remainder is in B.

Ex: DIV AB ; A = quotient and B = remainder

**INC (Increment) :**

 INC A ; Increment A by 1 (A = A + 1)

 INC Rn ; [Rn] = [Rn] + 1

 INC @Ri ; [@Ri] = [@Ri] + 1

 INC direct ; direct = direct + 1

This is used to increment the contents of the register by 1.

Ex: INC R1

 INC DPTR

**DEC (Decrement) :**

 DEC A ; Decrement A by 1 (A = A - 1)

 DEC Rn ; [Rn] = [Rn] - 1

 DEC @Ri ; [@Ri] = [@Ri] - 1

 DEC direct ; direct = direct – 1

This is used to decrement the contents of the register by 1.

Ex: DEC R3

 DEC DPTR

**Logical instructions :**

Logical instructions perform standard Boolean operations such as AND, OR, XOR, NOT

(compliment). Other logical operations are clear accumulator, rotate accumulator left and

right, and swap nibbles in accumulator.

**AND (Logical AND):**

 ANL destination, source ; dest = dest AND source

 ANL A, Rn ; A & [Rn]

 ANL A, direct ; A & [direct memory]

 ANL A, @Ri ; A & [memory pointed to by Ri]

 ANL A, #data ; A & immediate data

 ANL direct, A ; [direct] = [direct] & A

 ANL direct, #data ; [direct] = [direct] & immediate data

This instruction will perform a logical AND on the two operands and place the result in the destination. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate. The ANL instruction is often used to mask (set to 0) certain bits of an operand.

Ex: ANL A, R1

 ANL A, #45

**OR (Logical OR):**

 ORL destination, source ; dest = dest OR source

 ORL A, Rn ; A | [Rn]

 ORL A, direct ; A | [direct memory]

 ORL A, @Ri ; A | [memory pointed to by Ri]

 ORL A, #data ; A | immediate data

 ORL direct, A ; [direct] = [direct] | A

 ORL direct, #data ; [direct] = [direct] | immediate data

This instruction will perform the OR operation on the two operands, and place the result in the destination. The ORL instruction can be used to set certain bits of an operand to 1. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate.

Ex : ORL A, R5

 ORL A, #76

**XOR (Logical XOR) :**

 XRL destination, source ; dest = dest XOR source

 XRL A, Rn ; A ^| [Rn]

 XRL A, direct ; A ^ [direct memory]

 XRL A, @Ri ; A ^ [memory pointed to by Ri]

 XRL A, #data ; A ^ immediate data

 XRL direct, A ; [direct] = [direct] ^ A

 XRL direct, #data ; [direct] = [direct] ^ immediate data

This instruction will perform the XOR operation on the two operands, and place the result in the destination. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate.

Ex: XRL A, R3

 XRL A, #32

**CPL (Complement) :**

 CPL A ; Complement A

This instruction complements the contents of register A. The complement action changes the 0s to 1s and 1s to 0s. This is also called 1’s complement.

Ex: CPL A

**Rotate and Swap instructions :**

 In many applications there is a need to perform a bitwise rotation of an operand. In the 8051 the rotation instructions RL, RR, RLC, and RRC are designed specifically for that purpose. They allow a program to rotate the accumulator right or left. In the 8051, to rotate a byte the operand must be in register A. There are two types of rotations. One is simple rotation of the bits of A, and the other is a rotation through the carry.

**Rotating the bits of A right and left :**

 RR A ; rotate right A

In rotate right, the 8 bits of the accumulator are rotated right one bit, and bit D0 exits from the least significant bit and enters into D7 (most significant bit).



 RL A ; rotate left A

In rotate left, the 8 bits of the accumulator are rotated left one bit, and bit D7 exits from the (most significant bit) and enters into D0 (least significant bit).



**Rotating through the carry :**

 RRC A ; rotate right through carry

In RRC A, as bits are rotated from left to right, they exit the LSB to the carry flag, and the carry flag enters the MSB. In other words, in RRC A, the LSB is moved to CY and CY is moved to the MSB. In reality, the carry flag acts as if it is part of register A, making it a 9-bit register.



 RLC A ; rotate left through carry

In RLC A, as bits are shifted from right to left they exit the MSB and enter the carry flag, and the carry flag enters the LSB. In other words, in RCL the MSB is moved to CY and CY is moved to the LSB.



**SWAP :**

 SWAP A

Another useful instruction is the SWAP instruction. It works only on the accumulator (A). It swaps the lower nibble and the higher nibble. In other words, the lower 4 bits are put into the higher 4 bits and higher 4 bits are put into the lower 4 bits.



Ex: SWAP A

**Single Bit instructions (Boolean Variable instructions) :**

One unique and powerful feature of the 8051 is single bit operation. Single bit instructions allow the programmer to set, clear, move and complement individual bits of a port, memory or register.

**CLR (Clear) :**

 CLR bit ; Clear the bit

 CLR C ; Clear the carry

This instruction clears (set to 0) the specified bit indicated in the instruction. CLR instruction can operate on the carry flag or any directly addressable bit.

Ex : CLR P2.1

 CLR C

**SETB (Set bit ) :**

 SETB bit ; Set the bit

 SETB C ; Set the carry

This operation sets the specified bit to 1. SETB instruction can operate on the carry flag or any directly-addressable bit.

Ex : SETB P1.3

 SETB C

**CPL (Complement ) :**

 CPL bit ; Complement the bit

 CPL C ; Complement the carry

This operation complements the bit indicated by the operand. CPL instruction can operate on the carry flag or any directly addressable bit.

Ex: CPL P2.1

CPL P2.2

**JB (Jump on bit) :**

 JB bit, target ; Jump to target if bit = 1

This instruction jumps to the address indicated if the destination bit is 1, otherwise the program continues to the next instruction. The bit tested is not modified.

Ex: JB ACC.7, target1

JB P1.2, target2

**JNB (Jump on no bit) :**

 JNB bit, target ; Jump to target if bit = 0

This instruction jumps to the address indicated if the destination bit is 0, otherwise the program continues to the next instruction. The bit tested is not modified.

Ex: JNB ACC.6, target1

JNB P1.3, target2

**JBC (Jump on bit and clear the bit) :**

 JBC bit, target ; Jump to target if bit = 1, then clear the bit

If the source bit is 1, this instruction clears it and branches to the address indicated; else it proceeds with the next instruction.

Ex: JBC P1.3, target1

JBC P1.2, target2

**JC (Jump on carry) :**

 JC target ; Jump to target if CY = 1

This instruction branches to the address, indicated by the label, if the carry flag is set, otherwise the program continues to the next instruction.

Ex: JC target1

**JNC (Jump on no carry) :**

 JNC target ; Jump to target if CY = 0

This instruction branches to the address, indicated by the label, if the carry flag is **not** set, otherwise the program continues to the next instruction.

Ex: JNC target1

**ANL (AND Logic) :**

 ANL C,bit ; AND CY with bit and save it on CY

 ANL C,/bit ; AND CY with inverted bit and save it on CY

This instruction ANDs the bit addressed with the carry bit and stores the result in the carry bit itself. If the source bit is a logical 0, then the instruction clears the carry flag; else the carry flag is left in its original value. If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, but the source bit itself is not affected.

Ex : ANL C,P2.7 ; AND carry flag with bit 7 of P2

 ANL C,/OV ; AND with inverse of OV flag

**ORL ( OR Logic) :**

 ORL C,bit ; OR CY with bit and save it on CY

 ORL C,/bit ; OR CY with inverted bit and save it on CY

This instruction ORs the bit addressed with the carry bit and stores the result in the carry bit itself. It sets the carry flag if the source bit is a logical 1; else the carry is left in its original value. If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, but the source bit itself is not affected.

Ex: ORL C,P2.7 ; OR carry flag with bit 7 of P2

 ORL C,/OV ; OR with inverse of OV flag

**MOV ( Copy ) :**

 MOV b, C ; Copy carry status to bit location (CY = b)

 MOV C, b ; copy bit location status to carry (b = CY)

The instruction loads the value of source operand bit into the destination operand bit. One of the operands **must** be the carry flag; the other may be any directly-addressable bit.

Ex : MOV P2.3, C

MOV C, P3.3

**Program branching instructions (LOOP, JUMP and CALL) :**

**Looping instructions :**

Repeating a sequence of instructions a certain number of times is called a loop. In 8051 the loop action is performed by the instruction “DJNZ reg, label”. In this instruction, the register is decremented, if it is not zero, it jumps to the target address referred to by the label. Prior to the start of the loop the register is loaded with the counter for the number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.

Ex: DJNZ R2, label

**Jump instructions:**

In 8051, the jump instructions are divided into two categories as conditional and unconditional.

**Conditional jump instructions :**

The conditional jump is a jump in which the control is transferred conditionally to the target location. Some of the conditional jump instructions are tabulated below.

|  |  |
| --- | --- |
| Instruction | Action |
| JZ | Jump if A = 0 |
| JNZ | Jump if A ≠ 0 |
| DJNZ | Decrement and jump if A ≠ 0  |
| CJNE A,byte | Jump if A ≠ byte |
| CJNE reg, #data | Jump if byte ≠ #data |
| JC | Jump if CY = 1 |
| JNC | Jump if CY = 0 |
| JB | Jump if bit = 1 |
| JNB | Jump if bit = 0 |
| JBC | Jump if bit = 1 and clear bit |

**JZ (Jump on zero) :**

In this instruction the content of register A is checked. If it is zero, it jumps to the target address.

Ex: JZ target

**JNZ (Jump on no zero) :**

In this instruction the content of register A is checked. If it is not zero, it jumps to the target address.

Ex: JNZ target

**CJNE (Compare and jump if not equal ) :**

The 8051 has an instruction for the comparision.

 CJNE destination, source, relative address

In the 8051, the actions of comparing and jumping are combined into a single instruction called CJNE ( compare and jump if not equal ). The CJNE instruction compares two operands, and jumps if they are not equal. In CJNE, the destination operand can be in the accumulator or in one of the Rn registers. The source operand can be in a register, in memory, or immediate.

Ex : CJNE A, #0FF, target

 CJNE R1, #00, target

All conditional jumps are short jumps, meaning that the address of the target must be within -128 to +127 bytes of the contents of the PC.

**Unconditional jump instructions :**

The unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051, there are two unconditional jumps : LJMP (long jump) and SJMP (short jump).

**LJMP (Long jump) :**

LJMP is an unconditional long jump. It is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH.

 LJMP target ; Jump to target unconditionally

**SJMP (Short jump) :**

SJMP is an unconditional short jump. It is a 2-byte instruction in which the first byte is the op- code, and the second byte represents the relative address of the target location. The relative address ranges from 00 to FFH.

 SJMP target ; Jump to target unconditionally

**CALL instructions :**

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the 8051 there are two instructions for call : LCALL (long call) and ACALL (absolute call).

**LCALL (Long call) :**

It is a 3-byte instruction. In which, the first byte is the op-code and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K byte address space of the 8051.

 LCALL target ; Jump to subroutine located at target address

**ACALL (Absolute call) :**

It is a 2-byte instruction. In which, the first byte is the op-code and the second byte is used for the address of the target subroutine. ACALL can be used to call subroutines located within 2K bytes address.

 ACALL target ; Jump to subroutine located at target address

**RET (Return) :**

When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instruction from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction.

 RET ; control returns to main program

**Data transfer instructions :**

Data transfer instructions are used to transfer data between an internal RAM location and

SFR location, without going through the accumulator. Data can also be transferred between

the internal and external RAM by using indirect addressing. The upper 128 bytes of data RAM are accessed only by indirect addressing and the SFRs are accessed only by direct addressing.

**MOV (Move):**

MOV destination, source ; destination = source

This instruction moves the source byte into the destination location. The source byte is not affected.

Ex: MOV Ri, direct ; Ri = [direct]

MOV Ri, #data ; Ri = 8-bit immediate data

MOV DPTR, #data 16 ; DPTR = 16-bit immediate data

**MOVC :**

MOVC A,@A+<base register> ; A = Code byte from [@A+<base register>]

This instruction moves a code byte from program memory into ACC. The effective address of the byte fetched is formed by adding the original 8-bit accumulator contents and the contents of the base register, which is either the data pointer (DPTR) or program counter (PC). 16-bit addition is performed and no flags are affected. The instruction is useful in reading the look-up tables in the program memory. If the PC is used, it is incremented to the address of the following instruction before being added to the ACC.

Ex: MOVC A,@A + PC ; A = Code byte from [@A+PC]

MOVC A,@A + DPTR ; A = Code byte from [@A+DPTR]

**MOVX :**

 MOVX destination, source ; destination = source

This instruction transfers data between ACC and a byte of external data memory. There are two forms of this instruction, the only difference between them is whether to use an 8-bit or 16-bit indirect addressing mode to access the external data RAM.

Ex : MOVX @DPTR, A ; External[@DPTR] = A

MOVX @Ri, A ; External[@Ri] = A

MOVX A, @DPTR ; A = Data byte from external ram [@DPTR]

MOVX A, @Ri ; A = Data byte from external ram [@Ri]

**PUSH :**

 PUSH direct ; PUSH into stack

This instruction increments the stack pointer (SP) by 1. The contents of *Direct*, which is an internal memory location or a SFR, are copied into the internal RAM location addressed by the stack pointer.

Ex: PUSH 22h

 PUSH 56h

**POP :**

POP direct ; POP form stack

This instruction reads the contents of the internal RAM location addressed by the stack pointer (SP) and decrements the stack pointer by 1. The data read is then transferred to the *Direct* address which is an internal memory or a SFR.

Ex: POP DPH

 POP DPL

**XCH :**

 XCH A, byte ; A = byte, byte = A

This instruction swaps the contents of ACC with the contents of the indicated data byte.

Ex : XCH A, Ri ; A = Ri, Ri = A

 XCH A, direct ; A = direct, direct = A

 XCH A, @Ri ; A = @Ri, @Ri = A

**XCHD :**

 XCHD A, @Ri ; Exchange lower order digits

This instruction exchanges the low order nibble of ACC (bits 0-3), with that of the internal RAM location pointed to by Ri register. The high order nibbles (bits 7-4) of both the registers remain

the same.

Ex: XCHD A,@R1

**Time delay generation and calculation :**

For the CPU to execute an instruction takes a certain number of clock cycles. In the 8051 family, these clock cycles are referred to as machine cycles. To calculate the time delay, we use this machine cycles. In the 8051 family, the length of the machine cycle depends on the frequency of the crystal oscillator connected to the 8051 system. The crystal oscillator along with on-chip circuitry, provide the clock source for the 8051 CPU.

In the 8051, one machine cycle lasts 12 oscillator periods. Therefore, to calculate the machine cycle, we take 1/12 of the crystal frequency, then take its inverse.

If the crystal frequency is 11.0592MHz, then

Frequency of the machine cycle = 11.0592MHz/12 = 921.6 KHz

Time period of the machine cycle = 1/921.6KHz = 1.085 $µ$s

**Delay calculation :**

A delay subroutine consists of two parts 1) setting a counter and 2) a loop. Most of the time delay is performed by the body of the loop.

 MOV A, #55 ; Move the data 55h to A

AGAIN: MOV P1, A ; Move the contents of A to Port1

 LCALL DELAY ; Call delay subroutine

 CPL A ; Complement the contents of A

SJMP AGAIN ; Go to label AGAIN

 DELAY: MOV R3, #0FF ; Move the data FFh to register R3

 HERE: DJNZ R3, HERE ; Decrement R3, if R3≠0, then go to label HERE

 RET ; Return to main program

We have the following machine cycles for each instruction of the DELAY subroutine.

 Machine cycles No. of times the

 Instruction is executed

DELAY: MOV R3, #0FF 1 1

 HERE: DJNZ R3, HERE 2 255

 RET 1 1

Therefore, we have a time delay of [(1x1) + (2x255) + (1x1)] x 1.085$ µ$s = 555.52$ µ$s

Another way to get a large delay is to use a loop inside a loop, which is also called a nested loop.

 Machine cycles No. of times the

 Instruction is executed

 DELAY: MOV R5, #0FF 1 1

LOOP: MOV R3, #0FF 1 255

 HERE: DJNZ R3, HERE 2 255

 DJNZ R5, LOOP 2 255

 RET 1 1

Therefore, we have a time delay of [(1x1) + (1x255) + (2x255) + (2x255) + (1x1) ] x 1.085$ µ$s = 1277 x 1.085$ µ$s = 1385.545$ µ$s

**Assembly language programs :**

**Addition :**

 MOV A, #DATA1 ; DATA1 is moved to A

 MOV R1, #DATA2 ; DATA2 is moved to R1

ADD A, R1 ; Add DATA1 and DATA2 and place the result in A

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A in 8500h

LJMP 03 ; User break

**INPUT :**

DATA1 = 12h

DATA2 = 15h

**RESULT :**

8500h – 27h

**Subtraction :**

 CLR C

 MOV A, #DATA1 ; DATA1 is moved to A

 MOV R1, #DATA2 ; DATA2 is moved to R1

SUBB A, R1 ; Subtract DATA1 - DATA2 and place the result in A

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A in 8500h

LJMP 03 ; User break

**INPUT :**

DATA1 = 25h

DATA2 = 15h

**RESULT :**

8500h – 10h

**Multiplication :**

 MOV A, #DATA1 ; DATA1 is moved to register A

 MOV B, #DATA2 ; DATA2 is moved to register B

MUL AB ; Multiply the contents of A and B

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A (lower byte) in 8500h

INC DPTR ; Increment the DPTR by 1 (i.e., 8501h)

MOV A,B ; Move the contents of B to A

MOVX @DPTR, A ; Store the content of B (higher byte) in 8501h

LJMP 03 ; User break

**INPUT :**

DATA1 = 02h

DATA2 = 15h

**RESULT :**

8500h – 30h (Lower byte)

8501h – 00h (Higher byte)

**Division :**

 MOV A, #DATA1 ; DATA1 is moved to register A

 MOV B, #DATA2 ; DATA2 is moved to register B

DIV AB ; Divide the contents of A by B

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A (Quotient) in 8500h

INC DPTR ; Increment the DPTR by 1 (i.e., 8501h)

MOV A,B ; Move the contents of B to A

MOVX @DPTR, A ; Store the content of B (Remainder) in 8501h

LJMP 03 ; User break

**INPUT :**

DATA1 = 95h

DATA2 = 10h

**RESULT :**

8500h – 09h (Quotient)

8501h – 05h (Remainder)

**Largest Number :**

MOV DPTR, #8500 ; Load the data pointer with 8500h

 MOV R1, #05 ; Move 05h (count) to register R1

 MOV B, #00 ; Move 00h to register B

LOOP3: MOVX A, @DPTR ; Move the first data byte to A (i.e.,8500h)

 CJNE A, B, LOOP1 ; If A≠B, then go to label LOOP1

LOOP1: JC LOOP2 ; If A<B (CY = 1), then go to label LOOP2

 MOV B, A ; Move the contents of register A to B

 INC DPTR ; DPTR is incremented by 1

 DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

 SJMP LOOP4 ; Jump to label LOOP4

LOOP2: INC DPTR ; DPTR is incremented by 1

 DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

LOOP4: MOV A, B ; Move the contents of register B to A

 MOV DPTR, #8600 ; Load the DPTR with 8600h

 MOVX @DPTR, A ; Move the contents of register A to 8600h

 LJMP 03 ; User break

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8600h – 56h

**Smallest Number :**

MOV DPTR, #8500 ; Load the data pointer with 8500h

 MOV R1, #05 ; Move 05h (count) to register R1

 MOV B, #0FF ; Move 00h to register B

LOOP3: MOVX A, @DPTR ; Move the first data byte to A (i.e.,8500h)

 CJNE A, B, LOOP1 ; If A≠B, then go to label LOOP1

LOOP1: JNC LOOP2 ; If A>B (CY = 0), then go to label LOOP2

 MOV B, A ; Move the contents of register A to B

 INC DPTR ; DPTR is incremented by 1

 DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

 SJMP LOOP4 ; Jump to label LOOP4

LOOP2: INC DPTR ; DPTR is incremented by 1

 DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

LOOP4: MOV A, B ; Move the contents of register B to A

 MOV DPTR, #8600 ; Load the DPTR with 8600h

 MOVX @DPTR, A ; Move the contents of register A to 8600h

 LJMP 03 ; User break

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8600h – 12h

**Ascending order :**

 MOV R0, #05 ; Move the count 05h to register R0

LOOP3: MOV DPTR, #8500 ; Load the DPTR with 8500h

 MOV A, R0 ; Move the contents of R0 to register A

 MOV R1, A ; Move the contents of A to register R1

LOOP2: MOVX A, @DPTR ; Move the contents of DPTR to register A

 MOV B, A ; Move the contents of A to register B

 INC DPTR ; Increment the DPTR by 1

 MOVX A, @DPTR ; Move the contents of DPTR to register A

 CLR C ; Clear the Carry (CY = 0)

 MOV R2, A ; Move the contents of A to register R2

 SUBB A, B ; Subtract A – B, if A>B then CY=0 else CY=1

 JNC LOOP1 ; If CY=0, then go to label LOOP1

 MOV A, B ; Move the contents of A to register B

MOVX @DPTR, A ; Move the contents of A to DPTR

DEC DPTR ; Decrement the DPTR by 1

MOV A, R2 ; Move the contents of R2 to register A

MOVX @DPTR, A ; Move the contents of A to DPTR

LOOP1: DJNZ R1, LOOP2 ; Decrement R1 by1, if R1≠0, then go to LOOP2

 DJNZ R0, LOOP3 ; Decrement R0 by 1, if R0≠0, then go to LOOP3

 LJMP 03 ; Stop the execution

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8500h – 12h

8501h – 23h

8502h – 34h

8503h – 43h

8504h – 56h

**Descending order :**

MOV R0, #05 ; Move the count 05h to register R0

LOOP3: MOV DPTR, #8500 ; Load the DPTR with 8500h

 MOV A, R0 ; Move the contents of R0 to register A

 MOV R1, A ; Move the contents of A to register R1

LOOP2: MOVX A, @DPTR ; Move the contents of DPTR to register A

 MOV B, A ; Move the contents of A to register B

 INC DPTR ; Increment the DPTR by 1

 MOVX A, @DPTR ; Move the contents of DPTR to register A

 CLR C ; Clear the Carry (CY = 0)

 MOV R2, A ; Move the contents of A to register R2

 SUBB A, B ; Subtract A – B, if A>B then CY=0 else CY=1

 JC LOOP1 ; If CY=1, then go to label LOOP1

 MOV A, B ; Move the contents of A to register B

MOVX @DPTR, A ; Move the contents of A to DPTR

DEC DPTR ; Decrement the DPTR by 1

MOV A, R2 ; Move the contents of R2 to register A

MOVX @DPTR, A ; Move the contents of A to DPTR

LOOP1: DJNZ R1, LOOP2 ; Decrement R1 by1, if R1≠0, then go to LOOP2

 DJNZ R0, LOOP3 ; Decrement R0 by 1, if R0≠0, then go to LOOP3

 LJMP 03 ; Stop the execution

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8500h – 56h

8501h – 43h

8502h – 34h

8503h – 23h

8504h – 12h

**Generation of a square wave :**

HERE: SETB P1.0 ; Make P1.0 = 1

 LACLL DELAY ; Call delay subroutine

 CLR P1.0 ; Make P1.0 = 0

 LCALL DELAY ; Call delay subroutine

 SJMP HERE ; Go to label HERE

DELAY: MOV R3, #0FF ; Load R3 with FFh

LOOP: DJNZ R3, LOOP ; Decrement R3 by1 and if R3≠0 go to label LOOP

 RET ; Return to main program



**Generation of a rectangular wave :**

HERE: SETB P1.0 ; Make P1.0 = 1

 LACLL DELAY1 ; Call delay subroutine

 CLR P1.0 ; Make P1.0 = 0

 LCALL DELAY2 ; Call delay subroutine

 SJMP HERE ; Go to label HERE

DELAY1: MOV R5, #0FF ; Load R5 with FFh

LOOP: MOV R6, #0FF ; Load R6 with FFh

HERE: DJNZ R6, HERE ; Decrement R6 by1 and if R6≠0 go to label HERE DJNZ R5, LOOP ; Decrement R5 by1 and if R5≠0 go to label LOOP RET ; Return to main program

DELAY2: MOV R3, #0FF ; Load R3 with FFh

LOOP: DJNZ R3, LOOP ; Decrement R3 by1 and if R3≠0 go to label LOOP

 RET ; Return to main program

****

**@@@**

**UNIT-VI:**

**INTRODUCTION TO REALTIME OPERATING**

**Introduction -Operating system (OS) :** An Operating system (OS) is a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create ,print ,copy , delete and display files , read data from files ,write data to files ,control the I/O operations , allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time ,share the other system resources and provide inter task communication ,Timers , clocks , memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

So, the Operating system can also be defined as a collection of system calls or functions which provide an interface between hardware and application program.

It manages the hardware resources of a computer and hosting applications that run on the computer. Hence it is also called a resource Manager.

An OS typically provides multitasking, synchronization, Interrupt and Event Handling, Input/Output, Inter-task Communication, Timers and Clocks and Memory Management.

The core of the OS is the Kernel which is typically a small, highly optimised set of libraries.The Kernel is a program that constitutes the central core of an operating system. It has complete control over everything that occurs in the system.

The Kernel is the first part of the operating system to load into memory during booting (i.e., system startup), and it remains there for the entire duration of the session because its services are required continuously.

The kernel provides basic services for all other parts of the operating system, typically including memory management, process management, file management and I/O (input/output) management (i.e., accessing the peripheral devices). These services are requested by other parts of the operating system or by application programs through a specified set of program interfaces referred to as system calls.

**Popular Operating Systems**: Windows (from Microsoft), MacOS, MS-Dos, Linux(Open source),Unix (Multi user-Bell Labs) , Xenix (Microsoft),Android (Mobile) ..

**REAL TIME SYSTEMS** : Real-time systems are those systems in which the correctness of the system depends not only on the Output , but also on the time at which the results are produced(Time constraints must be strictly followed).

Real time systems are two types. (i) Soft real time systems and (ii) Hard real time systems. A Soft real time system is one in which the performance of the system is only degraded but, not destroyed if the timing deadlines are not met .

**For Ex:** Air conditioner, TV remote or music player, Bus reservation ,automated teller machine in a bank , A Lift etc.

A hard Real time system is one in which the failure to meet the time dead lines may lead to a complete catastrophe or damage to the system.

For Ex: Air navigation system, Nuclear power plant , Failure of car brakes , Gas leakage system ,RADAR operation ,Air traffic control system etc.

**Typical Real Time Applications:** Real Time systems find applications invarious fields of science and technology. The prominent applications are (i) Digital Control (ii) command and control, (iii) Signal processing (iv) Telecommunication systems and (v) Defense etc .

**Examples:**

* In automobile engineering, the real time systems control the engine and brakes of the vehicle and regulate traffic lights for smooth travel.
* In air craft monitoring, the real time systems schedule and monitor the takeoff and landing of the planes, make it fly, maintain the flight path, and avoid accidents.
* The real time patient care system monitor and regulate the blood pressure and heart beats of the patient and also , they can entertain people with electronic games ,TV and music.
* The real time systems are found in Air Traffic Control system also. The Air Traffic Control (ATC) system regulates the flow of flights to each destination airport. It does so by assigning to each aircraft an arrival time and en route to the destination
* The real time systems are important in industries also.For example a system of robots perform assembly tasks and repairs in a factory or chemical industries where human beings cannot enter.
* An avionics system for a military aircraft ,the real time systems perform the tracking and ballistic computations and coordinates the RADAR and weapon control systems.
* Dgital filtering, video and voice compressing/decompression, and radar signal processing are the major applications of real time systems in signal processing.
* Another interesting application is the real-time database systems that refers to a diverse spectrum of information systems, ranging from stock price quotation systems, to track records databases, to real-time file systems.
* Real time systems are also found in Supervisory Control and Data Acquisition (SCADA). In SCADA systems the sensors are placed at different geographical points to collect the raw data and this data are processed and stored in a Real time data base.
* Robots used in nuclear power stations ,to handle the radioactive material and other dangerous materials .
* Real time system applications are also found in office automation where LASER printers and FAX machines are used .

**REAL TIME OPERATING SYSTEM (RTOS)** : It is an operating system that supports real-time applications by providing logically correct result within the deadline set by the user. A real time operating system makes the embedded system into a real time embedded system.

The basic structure of RTOS is similar to regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks.

Though the real-time operating systems may or may not increase the speed of execution, but they provide more precise and predictable timing characteristics than general-purpose OS.

The figure below shows the embedded system with RTOS

****

All the embedded systems are not designed with RTOS. Low end application systems do not require the RTOS but only High end application oriented embedded systems which require scheduling alone need the RTOS.

For example an embedded system which measures Temperature or Humidity etc. do not require any operating system.

Where as a Mobile phone , RADAR or Satellite system used for high end applications require an operating system.

**Popular Real-Time Operating Systems:**

|  |  |
| --- | --- |
| **RTOS** | **Applications/Features** |
| Windows CE(Microsoft Widows) | Used small foot print mobile and connected devicesSupported by ARM,MIPS, SH4 & x86 architectures |
| LynxOS | Complex, hard real-time applications·POSIX- compatible, multiprocess, multithreaded OS.·Supported by x86, ARM, PowerPC architectures |
| VxWorks(Wind river) | **·** Most widely adopted RTOS in the embedded industry.· Used in famous NASA rover robots Spirit and Opportunity·Certified by several agencies and international standards forreal time systems, reliability and security-critical applications. |
| Micrium µC/OS-II | · Ported to more than a hundred architectures including x86,mainly used in microcontrollers with low resources.· Certified by rigorous standards, such as RTCADO-178B |
| QNX | · Most traditional RTOS in the market.· Microkernel architecture; completely compatible with thePOSIX· Certified by FAADO-278 and MIL-STD-1553 standards. |
| Symbian | Designed for SmartphonesSupported by ARM, x86 architecture |
| VRTX | **·**Suitable for traditional board based embedded systems and SoC architectures·Supported by ARM, MIPS, PowerPC & other RISCarchitectures |
| RTLINUX | Open source |
| [Neutrino](http://en.wikipedia.org/wiki/QNX) |  |

**DIFFERENCES BETWEEN RTOS and GENERAL PURPOSE OS :**

The basic differences are Determinism, Task scheduling and preempting

**Determinism :** The key difference between general-computing operating systems and real-time operating systems is the “deterministic " timing behavior in the real-time operating systems. "Deterministic" timing means that OS consume only known and expected amounts of time. RTOS have their worst case latency defined. Latency is not of a concern for General Purpose OS.

**Task Scheduling :** General purpose operating systems are optimized to run a variety of applications and processes simultaneously, thereby ensuring that all tasks receive at least some processing time. As a consequence, low-priority tasks may have their priority boosted above other higher priority tasks, which the designer may not want. However, RTOS uses priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently. All system calls are deterministic, implying time bounded operation for all operations and ISRs. This is important for embedded systems where delay could cause a safety hazard. The scheduling in RTOS is time based. In case of General purpose OS, like Windows/Linux, scheduling is process based.

* **Preemptive kernel** - In RTOS, all kernel operations are pre-emptiable
* **Priority Inversion** - RTOS have mechanisms to prevent priority inversion
* **Usage** - RTOS are typically used for embedded applications, while General Purpose OS are used for Desktop PCs or other generally purpose PCs.

**Note** : **Jitter**: The Timing error of a task over subsequent iterations of a program or loop is referred to as jitter. RTOS are optimized to minimize jitter.

There are four broad categories of kernels.

i.**Monolithic** kernels: provide rich and powerful abstractions of the underlying hardware.

ii.**Microkernels** provide a small set of simple hardware abstractions and use applications called servers to provide more functionality

iii. **Hybrid** (modified Micro kernels) Kernels are much like pure Microkernels , except that they include some additional code in kernel space to increase performance.

iv. **Exo-kernels** provide minimal abstractions, allowing low-level hardware access. In Exokernel systems,library operating systems provide the abstractions typically present in monolithic kernels.

**Pre-Emptive and Non-Pre-Emptive :** In a normal operating system ,if a task is running ,it will continue to run until its completion .It cannot be stopped by the OS in the middle due to any reason .Such concept is known as non-preemptive.

In real time OS ,a running task can be stopped due to a high priority task at any time with-out the willing of present running task.This is known as pre-emptiveness.

So, Preemptive scheduling involves scheduling based on the highest priority. The highest priority will always be given chance. Non-preemptive scheduling is a process is not interrupted once started until it is finished.

**Initialization of RTOS:**

RTOS is initialized using the following code.

Void main(void)

{

Init RTOS( ); /\*Initialize the RTOS\*/

Start task (v respond to Button, High \_priority);

Start task (v calculate tasklevels , low\_priority);

Start\_RTOS ( ); /\*start RTOS\*/

}

**Architecture of the RTOS :**

The heart or nucleus of any RTOS is the kernel. Inside the kernel is the scheduler. It is basically a set of algorithms which manage the task running order. Multitasking definition comes from the ability of the kernel to control multiple tasks that must run within time deadlines. Multitasking may give the impression that multiple threads are running concurrently, as a matter of fact the processer runs task by task, according to the task scheduling.



**General Architecture of RTOS**

**Architecture of the Kernel :**

The kernel is the core of an operating system. It is a piece of software responsible for providing secure access to the system’s hardware and to running the programs. Kernel is common to every operating system either a real time or non-real time .The major difference lies in its architecture .Since there are many programs, and hardware access is limited, the kernel also decides when and how long a program should run. This is called scheduling. Kernels has various functions such as file management, data transfer between the file system ,hardware management ,memory management and also the control of CPU time. The kernel also handles the Interrupts.

**Kernel Objects :** The various kernel objects are Tasks, Task Scheduler, Interrupt Service Routines, Semaphores, Mutexes, Mailboxes, Message Queues, Pipes, Event Registers, Signals and Timers

**(i).Task:**

A task is a basic unit or atomic unit of execution that can be scheduled by an RTOS to use the system resources like CPU , Memory, I/O devices etc. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state. The control signal that initiates the execution of a task is provided by the operating system.

There are two types of tasks. (i)Simple Task(S-Task) and (ii) Complex Task(C-Task).

****

**Simple Task (S-task):** A simple task is one which has no synchronization point i.e., whenever an S -task is started, it continues until its termination point is reached. Because an S-task cannot be blocked within the body of the task the execution time of an S-task is not directly dependent on the progress of the other tasks in the node. S-task is mainly used for single user systems.

**Complex Task (C-Task):** A task is called a complex task (C-Task) if it contains a blocking synchronization statement (e.g., a semaphore operation "wait") within the task body. Such a "wait" operation may be required because the task must wait until a condition outside the task is satisfied, e.g., until another task has finished updating a common data structure, or until input from a terminal has arrived.

**Task States:**

At any instant of time a task can be in one of the following states :

**(i) Dormant**

**(ii).Ready**

**(iii).Running and**

**(iv).Blocked.**

When a task is first created , it is in the dormant task .When it is added to RTOS for scheduling ,it is a ready task. If the input or a resource is not available ,the task gets blocked.



If no task is ready to run and all of the tasks are blocked, the RTOS will usually run the Idle Task .An Idle Task does nothing .The idle task has the lowest priority.

void Idle task(void)

{

While(1);

}

**Creation of a Task:**

A task is characterized by the parameters like task name , its priority , stack size and operating system options .To create a task these parameters must be specified .A simple program to create a task is given below.

result = task-create(“Tx Task”, 100,0x4000,OS\_Pre-emptiable); /\*task create\*/

if (result = = os\_success)

{ /\*task successfully created\*/

}

**Task Scheduler:**

Task scheduler is one of the important component of the Kernel .Basically it mis a set of algorithms that manage the multiple tasks in an embedded system. The various tasks are handled by the scheduler in an orderly manner. This produces the effect of simple multitasking with a single processor. The advantage of using a scheduler is the ease of implementing the sleep mode in microcontrollers which will reduce the power consumption considerably (from mA to µA). This is important in battery operated embedded systems.

The task scheduler establishes task time slots. Time slot width and activation depends on the available resources and priorities.

A scheduler decides which task will run next in a multitasking system.

Every RTOS provides three specific functions.

(i).Scheduling (ii) Dispatching and (iii). Inter-process communication and synchronization.

The scheduling determines ,which task ,will run next in a multitasking system and the dispatches perform the necessary book keeping to start the task and Inter-process communication and synchronization assumes that each task cooperate with others.

**Scheduling Algorithms:** In Multitasking systemto schedule the various tasks ,different scheduling algorithms are used..They are

(a).First in First out

(b).Round Robin algorithm

©.Round Robin with priority

(d)Non-preemptive

(e)Pre-emptive.

In FIFO scheduling algorithm, the tasks which are ready-to-run are kept in a queue and the CPU serves the tasks on first-come-first served basis.

In Round-Robin Algorithm the kernel allocates a certain amount of time for each task waiting in the queue. For example, if three tasks 1, 2 and 3 are waiting in the queue, the CPU first executes task1 then task2 then task3 and then again task1.

The round-robin algorithm can be slightly modified by assigning priority levels to the tasks. A high priority task can interrupt the CPU so that it can be executed. This scheduling algorithm can meet the desired response time for a high priority task..This is the Round Robin with priority.

In Shortest-Job First scheduling algorithm, the task that will take minimum time to be executed will be given priority. The disadvantage of this is that as this approach satisfies the maximum number of tasks, some tasks may have to wait forever.

In preemptive multitasking, the highest priority task is always executed by the CPU, by preempting the lower priority task. All real-time operating systems implement this scheduling algorithm.

The various function calls provided by the OS API for task management are given below.

• Create a task

• Delete a task

• Suspend a task

• Resume a task

• Change priority of a task

• Query a task

**Interrupt Service Routines**:

An interrupt service routine (ISR), also known as an interrupt handler, is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt.

In a real-time embedded system

,there are two possible interrupts. one is the Hardware Interrupt and the other is the software Interrupt.

Hardware Interrupts are asynchronous interrupts which are triggered by an electric pulse ,where as software interrupts are synchronous interrupts and these are triggered by a command or instruction.

In hardware driven scheduling, mostly timers,keyboard devices ,I/O ports will take part.

ISR is a small program, which is executed to develop an interface between the user and the hardware. The CPU will execute the ISR subroutine when it receives either a hardware or software interrupt.

The synchronization mechanism cannot be used in an ISR , because it is not possible in an ISR to wait indefinitely for a resource to be available.

The faster the ISR can do its job ,the better the real time performance of the RTOS .Hence the ISR should be always as small as possible.

When the CPU receives either a software or hardware interrupts, it will try to execute the corresponding ISR. Before that all the other interrupt sources are disabled and the interrupts are enabled only after the completion of the ISR .Hence the CPU must execute the ISR as fast as possible and also the ISR must be always as small as possible.

In real-time operating systems, the interrupt latency, interrupt response time and the interrupt recovery time are very important.

**Interrupt Latency**: It is the time between the generation of an interrupt by a device and the servicing of the device which generated the interrupt.

For many operating systems, devices are serviced as soon as the device's interrupt handler is executed. Interrupt latency may be affected by interrupt controllers, interrupt masking, and the operating system's (OS) interrupt handling methods.

**Interrupt Response Time** : Time between receipt of interrupt signal and starting the code that handles the interrupt is called interrupt response time.

**Interrupt Recovery Time**: Time required for CPU to return to the interrupted code/highest priority task is called interrupt recovery time.

**Semaphores:**

A semaphore is nothing but a value or variable or data which can control the allocation of a resource among different tasks in a parallel programming environment.

The concept of semaphore was first proposed by the Dutch computer scientist Edsger Dijkstra in the year 1965.

So, Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems.

Semaphores which allow an arbitrary resource count are called counting semaphores, whilst semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

The operation of a semaphore can be understood from the following diagram.



**Types of Semaphores:** There are three types of semaphores (i).Binary Semaphores, (ii) Counting Semaphores and (iii). Mutexes.

A binary semaphore is a synchronization object that can have only two states 0 or 1. i.e not taken and taken.

**Take :** Taking a binary semaphore brings it in the “taken” state, trying to take a semaphore that is already taken enters the invoking thread into a waiting queue.

**Release:** Releasing a binary semaphore brings it in the “not taken” state if there are not queued threads. If there are queued threads then a thread is removed from the queue and resumed, the binary semaphore remains in the “taken” state. Releasing a semaphore that is already in its “not taken” state has no effect.



Binary semaphores have no ownership attribute and can be released by any thread or interrupt handler regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize threads with external events implemented as ISRs, for example waiting for a packet from a network or waiting that a button is pressed. Because there is no ownership concept a binary semaphore object can be created to be either in the “taken” or “not taken” state initially.

**Counting Semaphores:**

A counting semaphore is a synchronization object that can have an arbitrarily large number of states. The internal state is defined by a signed integer variable, the counter.

The counter value (N) has a precise meaning: The Negative value indicates that , there are exactly -N threads queued on the semaphore.

The Zero value indicates that no waiting threads, a wait operation would put in queue the invoking thread.

The Positive value indicates that no waiting threads, a wait operation would not put in queue the invoking thread.

Two operations are defined for counting the semaphores.

**Wait:** This operation decreases the semaphore counter .If the result is negative then the invoking thread is queued.

**Signal:** This operation increases the semaphore counter .If the result is nonnegative then a waiting thread is removed from the queue and resumed.

Counting semaphores have no ownership attribute and can be signaled by any thread or interrupt handler regardless of who performed the last wait operation .Because there is no ownership concept a counting semaphore object can be created with any initial counter value as long it is non-negative.

The counting semaphores are usually used as guards of resources available in a discrete quantity. For example the counter may represent the number of used slots into a circular queue, producer threads would “signal” the semaphores when inserting items in the queue, consumer threads would “wait” for an item to appear in queue, this would ensure that no consumer would be able to fetch an item from the queue if there are no items available.

The OS function calls provided for Semaphore management are

• Create a semaphore

• Delete a semaphore

• Acquire a semaphore

• Release a semaphore

• Query a semaphore

**Mutexes :**

Mutex means mutual exclusion A mutex is a synchronization object that can have only two states. They are not-owned and owned. Two operations are defined for mutexes

**Lock:** This operation attempts to take ownership of a mutex, if the mutex is already owned by another thread then the invoking thread is queued.

**Unlock:** This operation relinquishes ownership of a mutex. If there are queued threads then a thread is removed from the queue and resumed, ownership is implicitly assigned to the thread.

Mutex is basically a locking mechanism where a process locks a resource using mutex. As long as the process has mutex, no other process can use the same resource. (Mutual exclusion ). Once process is done with resource, it releases the mutex. Here comes the concept of ownership. Mutex is locked and released by the same process/thread. It cannot happen that mutex is acquired by one process and released by other.

So, Unlike semaphores, mutexes have owners. A mutex can be unlocked only by the thread that owns it . Most RTOSs implement this protocol in order to address the Priority Inversion problem.

Semaphores can also handle mutual exclusion problems but are best used as a communication mechanism between threads or between ISRs and threads.

The OS functions calls provided for mutex management are

• Create a mutex

• Delete a mutex

• Acquire a mutex

• Release a mutex

• Query a mutex

• Wait on a mutex

**Difference between Mutex & Semaphore:** Mutexes are typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

**Mailboxes:**

One of the important Kernel services used to sent the Messages to a task is the message mailbox. A Mailbox is basically a pointer size variable. Tasks or ISRs can deposit and receive messages (the pointer) through the mailbox.

A task looking for a message from an empty mailbox is blocked and placed on waiting list for a time (time out specified by the task) or until a message is received. When a message is sent to the mail box, the highest priority task waiting for the message is given the message in priority-based mailbox or the first task to request the message is given the message in FIFO based mailbox.

The operation of a mailbox object is similar to our postal mailbox. When someone posts a message in our mailbox , we take out the message.



A task can have a mailbox into which others can post a mail. A task or ISR sends the message to the mailbox.

To manage the mailbox object, the following function calls are provided in the OS API:

• Create a mailbox

• Delete a mailbox

• Query a mailbox

• Post a message in a mailbox

• Read a message form a mailbox.

**Message Queues**:

The Message Queues ,are used to send one or more messages to a task i.e the message queues are used to establish the Inter task communication. Basically Queue is an array of mailboxes. Tasks and ISRs can send and receive messages to the Queue through services provided by the kernel. Extraction of messages from a queue follow FIFO or LIFO structure.

Applications of message queue are

• Taking the input from a keyboard

• To display output

• Reading voltages from sensors or transducers

• Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on our application, the highest priority task or the first task waiting in the queue can take the message. At the time of creating a queue, the queue is given a name or ID, queue length, sending task waiting list and receiving task waiting list.

To use a message queue ,first it must be created.The creation of a Queue return a queue ID .So,if any task wish to post some message to a task ,it should use its queue ID.

qid = queue\_create( “MyQueue” , Queue\_options) ; //\*Queue name and OS

specification options\*//

Each queue can be configured as a fixed size/variable size.

The following function calls are provided to manage message queues

• Create a queue

• Delete a queue

• Flush a queue

• Post a message in queue

• Post a message in front of queue

• Read message from queue

• Broadcast a message

• Show queue information

• Show queue waiting list.

**Event Registers:**

Some kernels provide a special register as part of each tasks control block .This register, called an event register. It consists of a group of binary event flags used to track the occurrence of specific events. Depending on a given kernel’s implementation of this mechanism, an event register can be 8 or 16 or 32 bits wide, may be even more.



Each bit in the event register treated like a binary flag and can be either set or cleared. Through the event register, a task can check for the presence of particular events that can control its execution. An external source, such as a task or an ISR, can set bits in the event register to inform the task that a particular event has occurred.

For managing the event registers, the following function calls are provided:

• Create an event register

• Delete an event register

• Query an event register

• Set an event register

• Clear an event flag

**Pipes:**

Pipes are kernel objects that are used to exchange unstructured data and facilitate synchronization among tasks. In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown in below Figure.



Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created. Data is written via one descriptor and read via the other. The data remains in the pipe as an unstructured byte stream.

Data is read from the pipe in FIFO order. A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full. Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task, as shown in the below

Figure. It is also permissible to have several writers for the pipe with multiple readers on it.

****

The function calls in the OS API to manage the pipes are:

• Create a pipe

• Open a pipe

• Close a pipe

• Read from the pipe

• Write to the pipe

**Signals-Signal Handler**

A signal is an event indicator. It is a software interrupt that is generated when an event occurs. It diverts the signal receiver from its normal execution path and triggers the associated asynchronous processing. Mainly the , signals notify tasks of events that occurred during the execution of other tasks or ISRs. The difference between a signal and a normal interrupt is that signals are so-called software interrupts, which are generated via the execution of some software within the system. By contrast, normal interrupts are usually generated by the arrival of an interrupt signal on one of the CPU’s external pins. They are not generated by software within the system but by external devices.

The number and type of signals defined is both system-dependent and RTOS-dependent. An easy way to understand signals is to remember that each signal is associated with an event. The event can be either unintentional, such as an illegal instruction encountered during program execution, or the event may be intentional, such as a notification to one task from another that it is about to terminate. While a task can specify the particular actions to undertake when a signal arrives, the task has no control over when it receives signals. Consequently, the signal arrivals often appear quite random,

****

When a signal arrives, the task is diverted from its normal execution path, and the corresponding signal routine is invoked. The terms signal routine, signal handler, asynchronous event handler, and asynchronous signal routine are inter-changeable. Each signal is identified by an integer value, which is the signal number or vector number.

The function calls to manage a signal are

• Install a signal handler

• Remove an installed signal handler

• Send a signal to another task

• Block a signal from being delivered

• Unblock a blocked signal

• Ignore a signal.

**Timers:**

A timer is the scheduling of an event according to a predefined time value in the future, like setting an alarm clock. For instance, the kernel has to keep track of different times

• A particular task may need to be executed periodically, say, every 10ms. A timer is

used to keep track of this periodicity.

• A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time, it has to take appropriate action.

• A task may be waiting in a queue for a shared resource. If the resource is not available for a specified time, an appropriate action has to be taken.

The following function calls are provided to manage the timer:

• Get time

• Set time

• Time delay (in system clock ticks)

• Time delay (in seconds)

• Reset timer

**Memory Management**

It is a service provided by a kernel which allots the memory needed ,either static or dynamic for various processes. The manager optimizes the memory needs and memory utilization. The memory manager allocates memory to the processes and manages it with appropriate protection. There may be static and dynamic allocations of memory. The manager optimizes the memory needs and memory utilization. An RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.

Hence, the two instructions “Malloc” and “free” ,although available in C language , are not used by the embedded engineer ,because of the latency problem.

So, an RTOS may or may not support memory protection in order to reduce the latency and memory needs of the processes.

The API provides the following function calls to manage memory

• Create a memory block

• Get data from memory

• Post data in the memory

• Query a memory block

• Free the memory block.

**Priority Inversion Problem** In any real time embedded system ,if a high priority task is blocked or waiting and a low priority task is running or under execution ,this situation is called Priority Inversion. This priority Inversion is shown in the diagram below.

****

In Scheduling, priority inversion is the scenario where a low priority Task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task releases the resource,

effectively “inverting” the relative priorities of the two tasks.

Suppose some other medium priority task, one that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both the low priority task and the high priority task.

The consequences of the priority Inversion are

(i) Reduce the performance of the system (ii) May reduce the system responsiveness

which leads to the violation of response time guarantees (iii) Create problems in real-time systems.

There are two types of priority inversions.(i) Bounded and (ii).Unbounded.

For example let us consider two tasks TA and TB in a real time system. Task A has higher priority than Task B. Initially Task A is under execution. But Task A is blocked after some time due to interruption and Task B is scheduled next. The Task B acquires a mutex corresponding to a resource common to both Task A and Task B. After some time Task A acquire mutex before the completion of Task B. But Task A cannot acquire mutex and it is blocked because already Task B has acquired the mutex. So, the Task A ,though has the higher priority ,is blocked until Task B releases the mutex for the resource. This is called the bounded Priority inversion.

If the time over which the higher priority is blocked is unknown ,then it is called unbounded priority inversion.

The Priority Inversion is avoided by using two protocolas.,namely

(i).Priority Inheritance Protocol (PIP)

(ii) Priority Ceiling Protocol(PCP).

**The Priority Inheritance Protocol** is a resource access control protocol that raises the priority of a task, if that task holds a resource being requested by a higher priority task, to the same priority level as the higher priority task.

The **priority ceiling protocol** is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections .In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource.

**Types of operating systems :**

An Operating system (OS) is nothing but a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create ,print ,copy , delete and display files , read data from files ,write data to files ,control the I/O operations , allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time ,share the other system resources and provide inter task communication ,Timers , clocks , memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

There are three important types of operating systems .They are

(i).Embedded Operating System

(ii). Real time operating system and

(iii).Handheld operating system.

**(i).Embedded Operating System**

The operating system used for embedded computer systems is known as embedded operating system. These operating systems are designed to be compact, efficient, and reliable.

The embedded operating system uses a preemptive priority based kernel.But this kernel do not meet the strict dead lines.By removing the unnecessary components from the kernel of desktop operating system ,the embedded operating can be obtained. This OS occupies less memory space.The popularly known embedded operating systems are

(a).Embedded NT (b) Windows XP Embedded (c) Embedded Linux

The Embedded NT for its minimal operation without any network support occupies nearly 9MB of RAM and 8 MB of Flash .It is a preemptive ,multitasking operating system. Generally Embedded NT is preferred to other OSs because of its ease in developing the applications.It is suitable for embedded systems built around single board computers for applications,like Internet Kiosks,Automatic Teller Machines (ATM) etc..

Microsoft Windows XP Embedded is the successor to Embedded NT.It is also pre-emptive multitasking operating system like Embedded NT.This OS is widely used in set top boxes, point of sale terminals and Internet Kiosks etc.

Embedded Linux is a open source software and it is covered by GNU General Public License(GPL) and hence the complete source code is available at free of cost. The important features of Embedded Linux are POSIX support and availability of large software resources.

Embedded Linux is used in embedded computer systems such as mobile phones, personal digital assistants, media players, set-top boxes, and other consumer electronics devices, networking equipment, machine control, industrial automation, navigation equipment and medical instruments.

**Real-Time Operating System:**

A real-time operating system (RTOS) is an operating system (OS) intended to serve te real time application requests .A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task .

A hard real-time operating system has less jitter than a soft real-time operating system. The main objective is not the high throughput, but a guarantee of meeting the deadlines. An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.

A real-time OS has an advanced algorithm for scheduling Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real-time OS is valued more for how quickly or how predictably it can respond.

There are various Real-Time operating systems both commercial and open source in the market

(i).QNX Neutrino (ii)VxWorks (iii) microC/OS-II (iv).RTLinux.

**QNX Neutrino**  is a real time operating system from QNX Software systems limited . It is supported by ARM, MIPS, Power PC, Strong ARM,X86 and Pentium.

This OS supports multiple scheduling algorithms and upto 65535 tasks and can create embedded data base applications.

**microC/OS-II** is a real time operating system used mainly in academic institutions.It is available in source code form for non-commercial applications.This do not support the Round Robin scheduling algorithm.

**RT Linux** is a hard real time RTOS microkernel that runs the entire Linux operating system as a fully preemptive process. It was developed by Victor Yodaiken and Michael Barabanov at the New Mexico Institute of Mining and Technology. It was commercialized at FSM Labs.

RT Linux runs underneath the Linux OS. The Linux is an idle task for RT Linux. The real-time software running under RT Linux is given priority as compared to non-real-time threads running under Linux. This OS is an excellent choice for 32-bitprocessor based embedded systems.

**Handheld Operating Systems:**

Handheld computers are becoming very popular now a days due to their increasing applications. A handheld operating system, also known as a mobile OS, a mobile platform, is the operating system that controls a mobile device.

Typical examples of devices running a mobile operating system are smart phones, personal digital assistants (PDAs), tablet computers and information appliances, or what are sometimes referred to as smart devices, which may also include embedded systems, or other mobile devices and wireless devices.

The important requirements for a mobile operating system are:

• To keep the cost of the handheld computer low,small footprint of the OS is required.

• The OS should have support for soft real time performance.

• TCP/IP stack needs to be integrated along with the OS.

• Communication protocol stacks for Infrared, Bluetooth, IEEE 802.11interfaces need

to be integrated.

• There should be support for data synchronization.

The popular handheld operating systems available are (a) Palm OS (b)Symbian OS (iii)Windows CE (iv) Windows CE.NET.

**Palm OS** is a popular Handheld OS ,where the application development can be done using C,C++ or Java. Most of the Sony based handheld devices use this OS.

The **Windows CE** is another popular handheld operating system widely used in Pocket PCs and PDAs. In addition to this the handheld devices from Thoshiba , Hitachi ,Alpha use this OS. Application software can be developed in Visual Basic,VC++using the embedded visual tools.

**----------------xxxxxxx-----------**

**References:**

1.Embedded/Real Time Systems : Concepts,Design &Programming

 –Dr.K.V.K.K Prasad : Dreamtech Publs

2. Embedded & Real Time Systems Notes - Mr. Suman Kalyan Oduri