Embadded System

A system is a way of working organizing are during one are may cost according to a fixed plan, program or set of rules

A system is also arrangement in which all its unit assembles and work together according to the plan are program

ex: washing Machine, Digital camera, wristlet & DVD players

It is a time display system. Its par are its hardware, needles and battery with the beautiful dial, chassis and strap. this parts organize to show real time every second and continously update the time every second.

Embadded System:

An embadded system is a system that has embadded software and computer hardware which makes its a system dedicated for an applicatio or specific part of a application or product or a part of a larger system

9

DoDD D. MORTEN Auther of embadded micro controller:

Embadded Systems are electroni System that contain a microprocesso or micro controller. But be do not thing at them as computer. the computer is hidden or embaddad syste

David E. Simon

people use the term embadded system to mean any computer system hidden in any of this products.

Embadded

An embadded System is a Syste that has three main components embadded into it.

It embadded hardware Similar to hardware

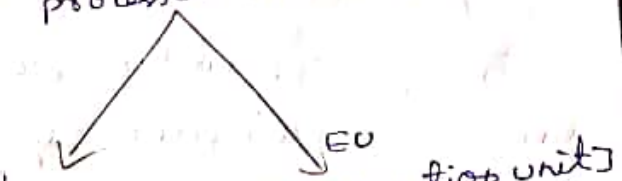
As its Software usually embadde in the ROM or flash memory its usually do not need a secondary hard disk CD, Memory as in a Computer.

It embadded main application software. the application software may concurrently perform a series of tasks or processor or threads.

It embadded a real time operating system the supervisor the application software running on hardware and organizes access to a resource according to the proprioritat taste in the system

Processor Embadded into a system:

Embadded processor in a System



Processor is a chip [IC) modal

Processor core form:

ASI -> Application Specific Integration
SOC -> System On a chip
VLSI -> Very Large Scale Integration

General purpose processor (GPP):

Micro processor
Embedded processor

Application Specific instraction set proces
(ASIP):

Micro controller
Embedded micro controller
Digital signal processor (GSP)
and Media processor
Network processor and I/O proces

Single purpose processor (SPP) as addition
Processors:

CO processor : digital number
Accelerator : eg : Java code
controller : eg : peripheral devic
GPP (or) ASIP cores Integrated
into either an ASIP a VLSI circuit
Application specific system
processor
Multicore processor

1)
Microprocessor:

one and only CPU NOT used in
RAM & ROM
~4 MHz it is yearly

5

Advanced in 4 GHz

Intel 80x86
↳ 32-bit

Use of embadded system

ARM, 68HCxxx, 80x86, SPARC

;)

Microcontroller:

8085, ARM, Intal, philips, samsung

Single purpose processor:-

1. co processor eg: floating point processor

2. Graphics processor

image : 144 x 176 pixel → CIF image

Video graphic : 640 x 480 pixel

Video frame 525 x 625 pixel

3. Pixel coprocessor:

Display high resolution picture formats → 2592 x 1944

4. Encryption engine

5. Decryption engine

6. A discrete cosine transformation (DCT)

Speech and video processing

7. protocol stack processor

8. Network processor

Connection establish

Data send and receive process

9. Accelerator

Real time process

10. CODEC (Coedon and Docodon)

11. JPEG CODEC : Joint photographic Expert Group

12. MPEG : (Motion pictures experts Group)

13. Controllers eg : (peripharal devices

ASIC : Application Specific Integration circuit :-

An application specific integrated

The components that can SOC generally loops, to incorporate with in itself include a control processing unit input and output ports internal memory.

VLSI : Very Large scale Integration:-

VLSI is the process of creating an integrated circuit by combining thousand of transistors into single chip

the micro processor is a VLSI device

Embadded hardware units and device in a system:

i) power source:

Some systems are directly connect with power source

not direct connect

NIC -> Network Interface card

5.0V ± 0.25V

3.3V ± 0.3V

2.0V ± 0.2V

1.5V ± 0.2V

we can also use charge pump

ii) clock oscillator and clocking units:

clock is digital signals

controls time to executing an instruction

iii) System timer:

to timer in a second
Configured for system clock
System supervisor function in the OS at periodical intervals

4. Read time clock:

Microcontroller provides the time circuit for the counting and timing devi

5. Reset circuit, Power-up Reset and watching-Timer Reset:

From the beginning using a switch or signal

Restart of the system from beginning whenever power is switched on in the system.

Restart of system when it is stuck up in certain set of instructions for a period more than a preset time interval.

Embedded Software in a System:

Final machine implement Software for a system.

An embedded system processor executes software that is specific to a given application or host system.

The instruction codes and data in the final space are place in ROM or flash memory. The software also called ROM image.

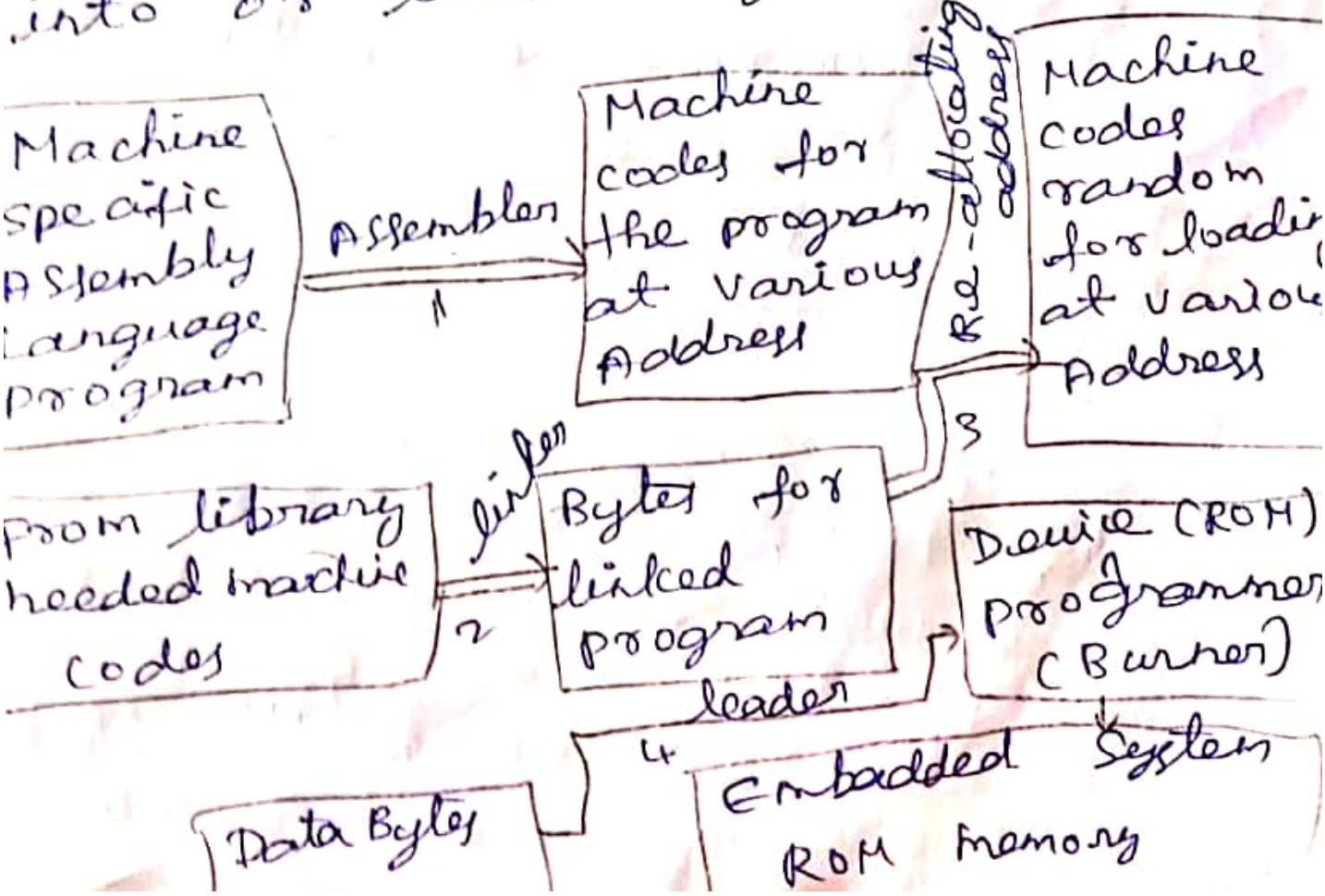
Each code and datum is available only in the bits and byte format. The system requires byte at each ROM address according to the task being executed.

A machine is also a table software

software in a processor specific assembly languages:-

A program are a small specific part can be coded in assembly language using a assembler after understanding the processor and its instruction set assembler the software used for developing code in assembly.

Assembly language coding is extreamly useful for configuring physical device like codes a line display ADC and DAC and reading into or transmitting from a buffer



④ The assembler converts the assembly language into the machine code using a step called assembly.

⑤ In the next step linking is called a linker like the code with the other codes required linking is necessary because of the number of codes to be link for the final binary file. The linked file in binary for run a computer is commonly known executed file or simply an "exe file".

⑥ In the next step the

13. Structural units in a processor:

Structural	units	Functions
MAR	Memory Address Register	It holds the address of the byte or word to be fetched from external memories
MDR	Memory Data Register	It holds byte or word fetched from an external memory or ID Address.
System Buses	Internal Bus	It externally connects all the structural units inside the processor. Its width can be 8, 18, 32, 48 (or) 64 bits.
	Address Bus	An external bus that carries the address from MAR to memory as well as to I/O devices and other units of system
	Data Bus	An external bus that carries during a read or write operation, the data from or to an address

	control bus	the address is determined by HAR. An internal set of signals to carry control signals to processor or memory or double.

BIU - Bus Interface Unit :-

An interface unit between processor internal units and external buses

IR - Instruction Register :-

It sequentially takes instruction codes (OP code) to execution unit of processor

ID - Instruction Dedector :-

It decodes the instruction received at the IR and passes it to processor

CPU dio - Pointers

CU - Control Unit :-

It controls all the bus activities and unit functions needed for processing

ARS - Application Register set :-

(a) A set of on chip register used during processing of instruction of an application program (or) b) A register windows (c) a subset of register with each subset storing variables of a software routing d) A register file associ

15 to a unit such as ALU or FLPU.

ALU - Arithmetic Logical Unit :-

A unit to execute arithmetic or logical instructions according to the current instructions present at IR.

PC - Program Counter :-

It generates an instruction cycle by sending the address defined by it to memory through MAR - It autoincrements as the instruction are fetched regularly and sequentially it is called instruction pointer in 80x86 processors.

SP - Stack Pointer :-

A pointer for an address which corresponds to a stack-top in memory.

IR - Instruction Queue :-

A queue of instruction so that the IR does not have to wait for the next instruction after one has been processed.

I-Cache - Instruction Cache:-

It sequentially stores like an instruction queue, the instructions in FIFO mode.

MMU - Memory Management Unit:-

It manages the memories such that the instruction and data readi available processing.

SRS - System Register Set:-

It is a set of registers used while processing the instructions of the supervisory system program.

FLPU - Floating point processing Unit:-

A unit separate from ALU for floating point processing which is essential in processing functions fast in a microprocessor.

FRS - Floating Point Register Set:-

A register set dedicated for storing floating point numbers in a standard format and used by FLPU for its data and stack.

Allocation of memory to program segments and blocks and memory map of a system:

17 ① Functions. Processor data and stacks at the various segment

⇒ Program routines and process can have different segments

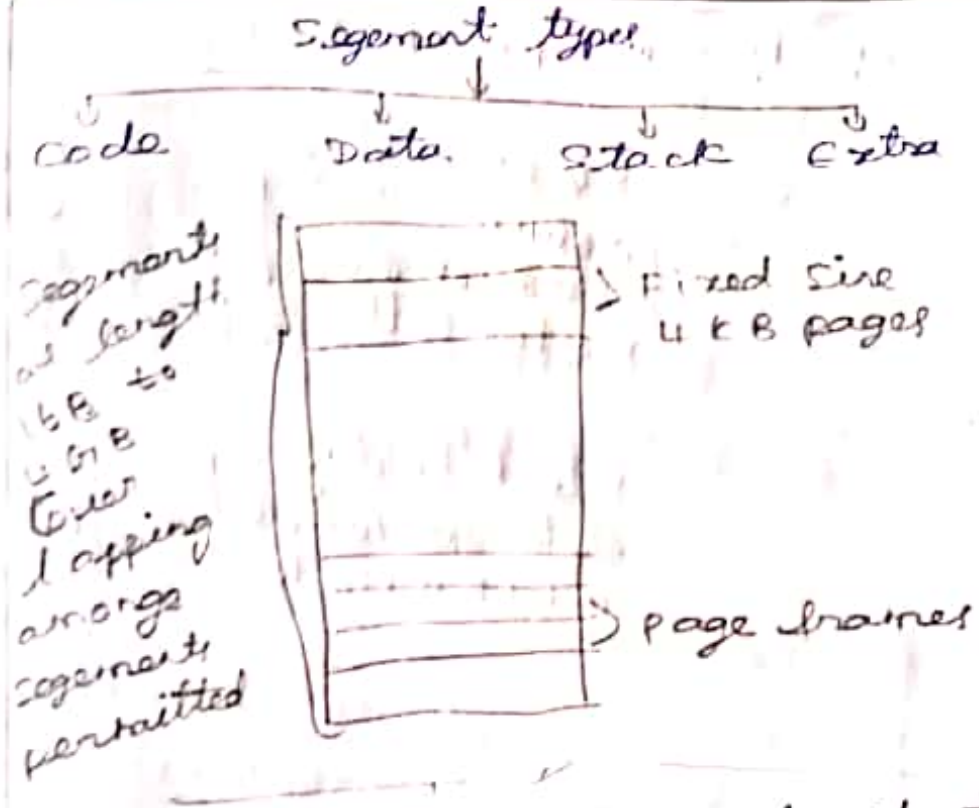
For example.

A program code can be segmented and each segment stored at a different memory block

A pointer address points to the start of the memory block storing a segment and an offset value it used to retrieve for a memory address within that segment

the data have also the segments with each segment at different blocks

A segment can have portions at fixed sizes called pages.



B) Memory blocks for elements of the different data structure and data set.

The software design approach is to use data sets and data structure in a program

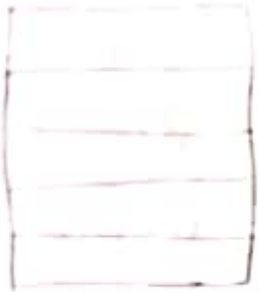
Their can be different sets and different structures of data at the memory.

Following of the data structure and data sets that are commonly used during processing in a system.

A) A Data structure called stack is a special program element:

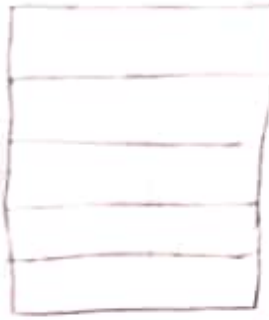
i) A stack means an alloted memory block from which a data element is allows use in a LIFO

used by the processor



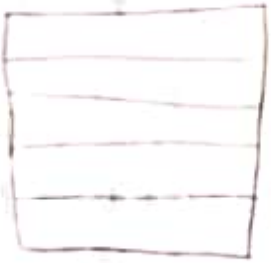
Stack holding return addresses of nested calls

2 byte address that PC requires



Stack holding data retrievable in LIFO mode

A memory block's start and e



Stack 1 holding data Retrievable set content

Stack - N Holding Retrievable Set - N content

Saved contexts of the Task as the stacks

Each procedure has atleast one start pointer so that the instructions start can be pointed on calling of the routines can be facilitated

B) The data structure array is an important programming element M[0], M[1] ... M[N] one dimensional array is a special data structure at the memory. It has a pointer address that always points to the first element of the array

← Mark[i] of a memory Block. Base Address Mark[0] ←

ii.) A data structure called a queue is another important programming element.

iv) Circular queue.
v) queue pipe
vi) table
vii) hash table (pair of key & assigned pointing values)

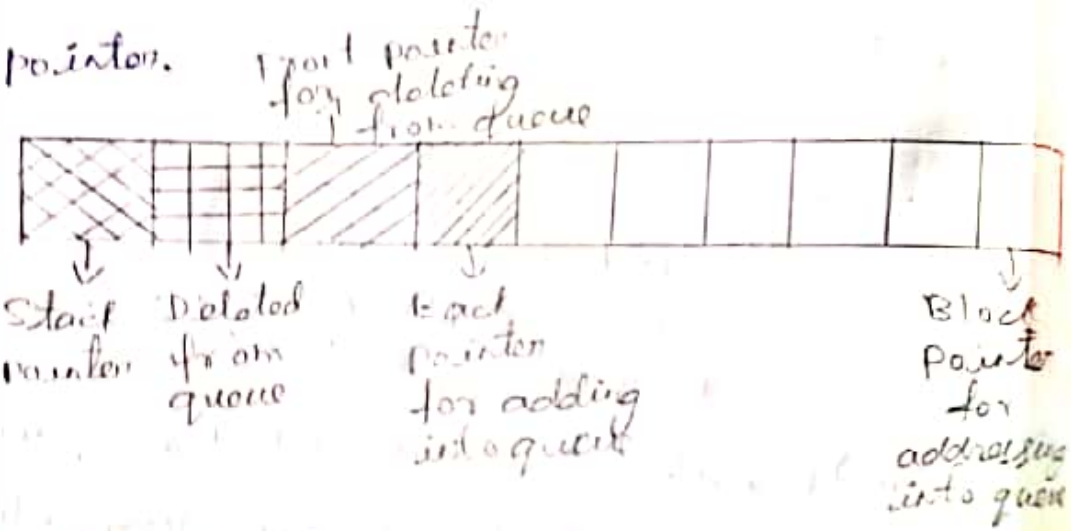
viii) list

It means an allotted memory block from which a data element is always retrieved in FIFO.

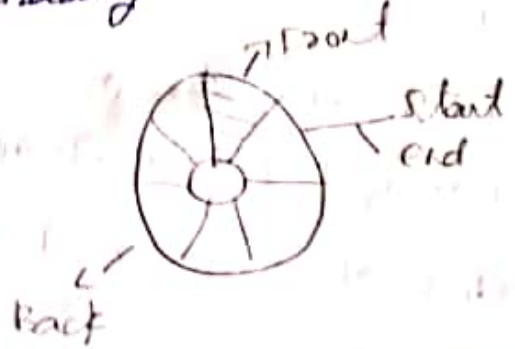
It has two pointers one for its head and the other for its tail.

this pointer should increment a each addition. It is called queue back or tail pointer.

The other pointer is for pointing to an address in a memory block from where an element can be deleted It is called queue front or head pointer.

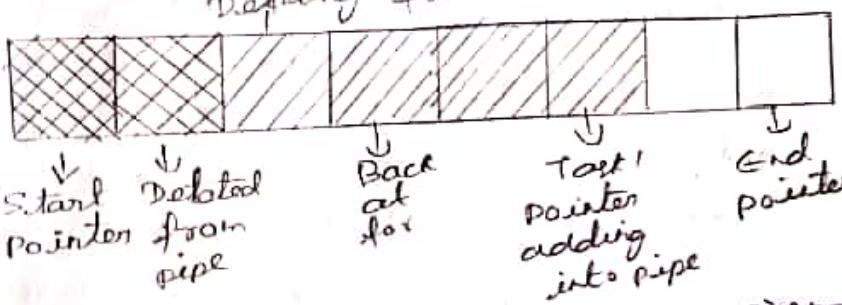


D) A circular queue is a queue in which both pointers cannot increment beyond the memory block and reset to starting value on insertion beyond the boundary



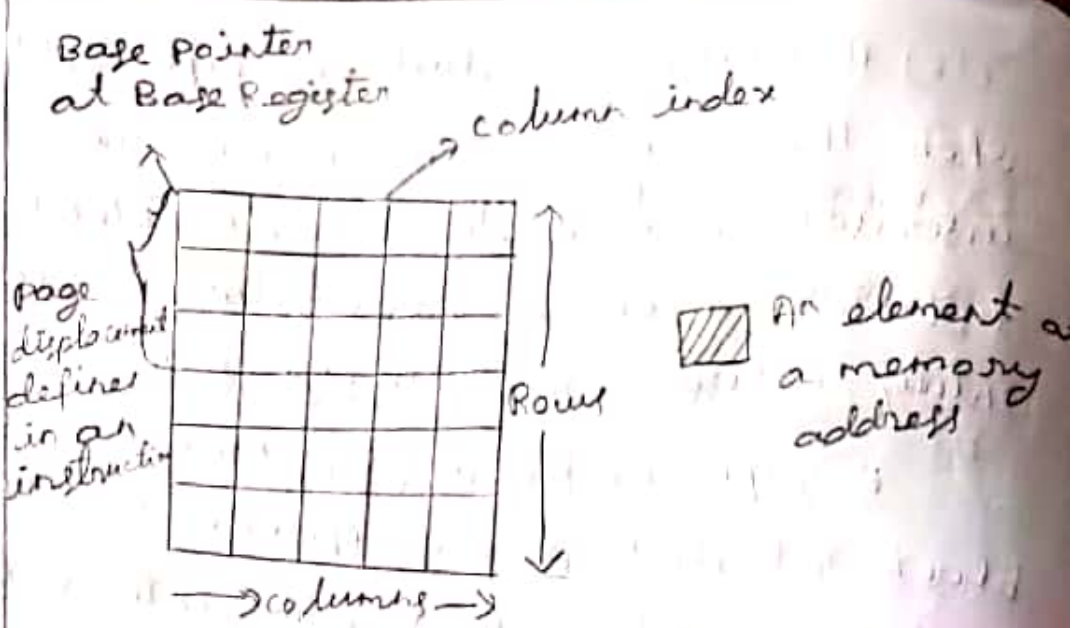
e) A queue is called a pipe, generally when the source from where the insertions are made has an identity distinct from a destination (sink) entity where deletion are made.

A pipe means a common memory block allotted for a queue to two distinct entities that are interconnected in some way.

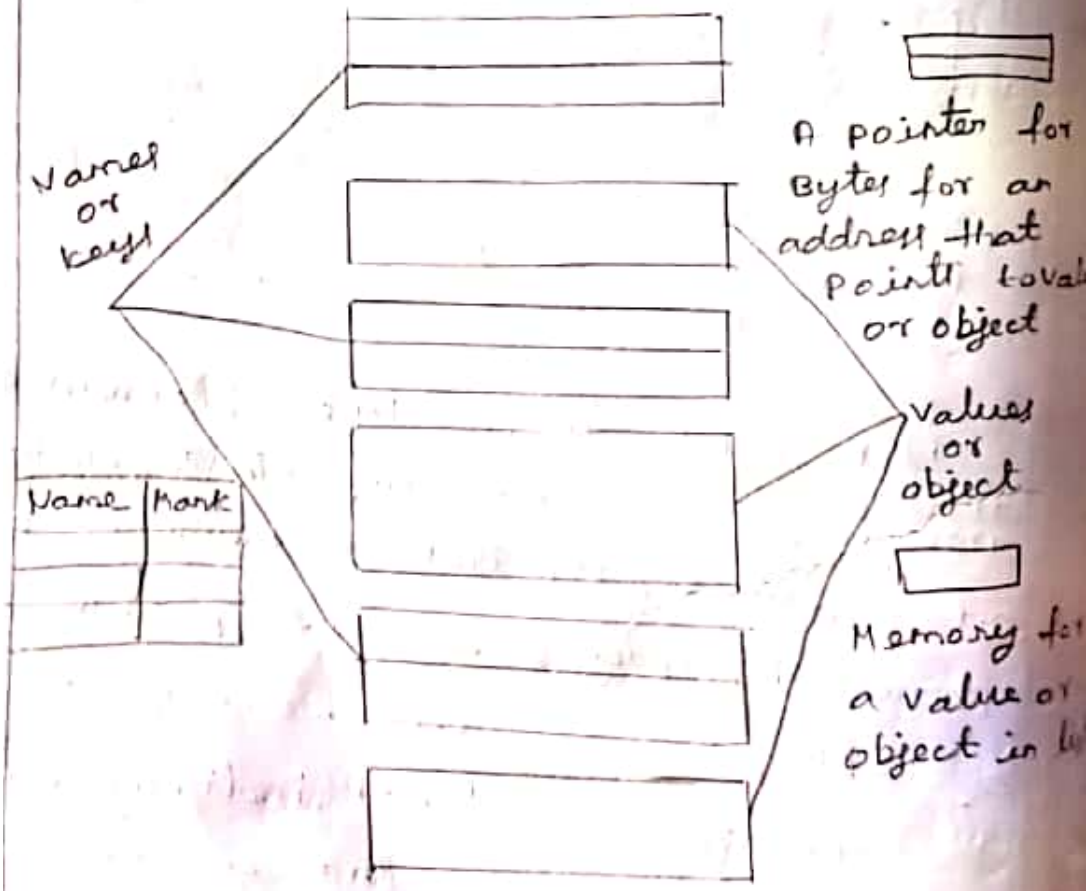


f) A table is a two-dimensional array and is an important data set that is allocated a memory block. There is always a base pointer for a table.

There are two indicies are for a column & other for a row.



G) A hash table is a data set that is a collection of pairs of a key and a corresponding value.

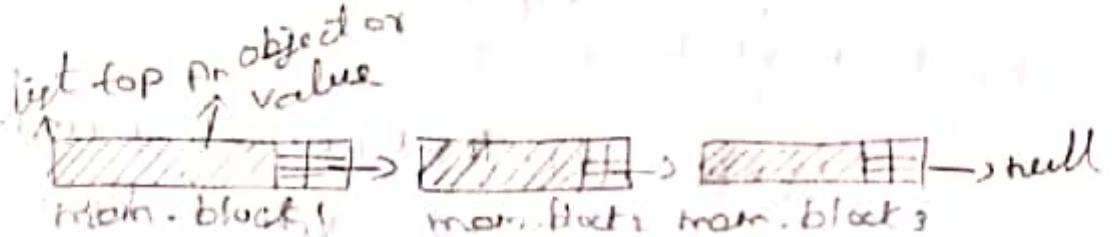


A has table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at non-consecutive memory address.

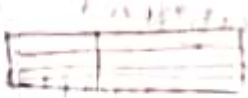
H) A list is a data structure with number of memory blocks, one for each element.

A list has a top pointer for the memory address from where it starts

Each list element at the memory also stores the pointer to the next element. The last element points to null.



(when list is empty)



Next top pointer address for a next list element or object

list f...

(when top is not empty & has one element only

6.7 INTERRUPT-SERVICING MECHANISM

A service routine executes on interrupt.

6.7.1 Preventing Interrupts Overrun

When a source causes an interrupt then execution of corresponding ISR services the request made through the interrupt. When interrupt from the same source reoccurs then that also needs to be serviced. Overrun means that the processor has not completed execution of an ISR for the earlier interrupt before a new one from that source reoccurred.

Example 6.11

1. Assume that the port is receiving bits every 1 μs, 1 byte in 8 μs and the next byte starts after 8 μs. When a new interrupt occurs before the previous byte saves then serial interrupt overrun occurs.
2. Assume that a synchronous serial-port interrupt occurs. If another source interrupted before the service of serial interrupt starts, and ISR for other source could not complete and before second serial interrupt occurs then interrupt overrun occurs.
3. A user inserts the coin in an automatic chocolate-vending machine, and the user inserts another coin before the ISR corresponding to earlier inserted coin finishes then interrupt overrun occurs.

Figure 6.15 shows the interrupt overrun situation for serial port interrupts.

Interrupt overrun can be prevented by several methods. One method is when an ISR is servicing, disable other interrupts.

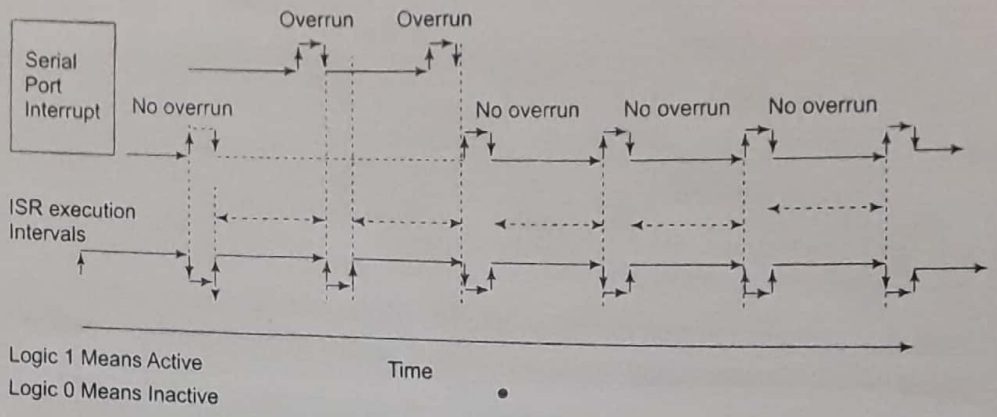


Fig. 6.15 Interrupt overrun of serial port interrupts

6.7.2 Disabling Interrupts

When a routine or ISR is executing in the codes, which must be completed because interrupt from same source is expected soon after, and the situation of interrupt overrun is likely to occur then an instruction DI (Disable Interrupts) is executed at the beginning of the ISR and EI (Enable Interrupts) is executed before return from that ISR.

When a routine or ISR is executing codes in a critical section of codes, which must completed because if another routine starts then situation of interrupt overrun is likely to occur then an instruction DI is executed at the beginning of the critical section of codes and EI at the exit from the section is executed.

Example 6.12

1. A microcontroller, device or system can have interrupt-control bit, one-bit EA (Enable All). EA is called *primary level disabling* or *enabling bit*. Resetting the EA bit disables complete interrupt system. Setting EA enables the interrupt system.
2. A microcontroller, device or system has individual interrupt-control bits for different interrupt sources or different groups with similar sources in one group. The bits are called *secondary level disabling* or *enabling bit*. Resetting one of the bits disables corresponding source or group of similar sources. Setting one of the bits enables corresponding source or group of similar sources.

6.7.3 Nonmaskable Interrupts and Maskable Interrupts Concept

Masking an interrupt means when interrupt takes place, the service to that interrupt-request does not occur. Maskable sources of interrupt provides for masking and unmasking the interrupt service (diversion to the ISR).

Example 6.13

Nonmaskable: Examples are RAM parity error in a PC and error interrupts like division by zero. These must be serviced.

Maskable: Execution of a device-interrupt source or source group can be masked by the corresponding interrupt-control bit. An interrupt request at external signal (at pin) can be masked. An interrupt request execution of a *trap* can be masked. An interrupt request execution of a software interrupt (*exception* function or *signal* function) can be masked. Most interrupt sources are maskable.

A few specific interrupts cannot be masked and they are called nonmaskable interrupts. *Nonmaskable interrupts* are those for which the service is uninterrupted. *Maskable interrupts* are those for which the service may be temporarily interrupted to let other ISRs execute till it is masked.

6.7.4 Interrupt Status Register or Interrupt Pending Register Concept

An identification of source of an interrupt is required, for the processor to interrupt and to initiate steps for servicing of the interrupt. A processing system can identify interrupt by status flag corresponding to an interrupt source in a processor status register. A flag (bit) in *status register* can identify the interrupt when that is set.

A bit in interrupt-pending register that corresponds to an interrupt source enables a processor to interrupt when the processor is not executing any other interrupt whose priority is higher than that.

Example 6.14

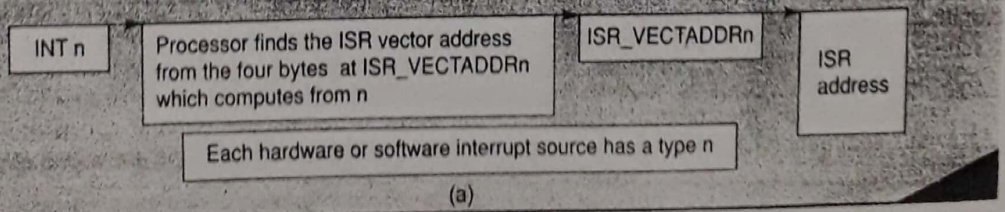
1. The 8051 SCON register status bits are RI and TI to identify the status of serial receive and serial transmit register. A common ISR executes when either TI or RI sets in. An ISR instruction is used to reset the flag.
2. The 8051 TCON register has TF0 and TF1 flag bits for timer 0 overflow and timer 1 overflow interrupt. TF0 auto-resets when ISR corresponding to T0 overflow start execution. TF1 auto-resets when ISR corresponding to T1 overflow start execution.

6.7.5 Interrupt Vector

An interrupt vector is a memory address to which processor vectors [transfers to instruction pointer (program counter) new address] on an interrupt. It then services the interrupt by executing ISR either (i) starting at that address, (ii) at address pointed and generated by bytes at that address, or (iii) starting at that address and then the ISR instruction points to the new ISR address corresponding to the source of interrupt or bits specified in the SWI instruction.

Example 6.15

1. The 8051 services the hardware interrupts by executing ISR starting at ISR_VECTADDR address. The 8051 processor generates distinct ISR_VECTADDR addresses for INT0, T0, INT1, T1, Serial and T2.
2. The 8086 services the interrupt by executing ISR at address pointed and generated by bytes address generated from *n* in INT *n* instruction or hardware interrupt corresponding to a specified *n*. Figure 6.16(a) shows generation of ISR_VECTADDR when instruction INT *n* executes for interrupting the processor.
3. Figure 6.16(b) shows the use of ISR_VECTADDR in APM for the jump to the routine for the interrupt servicing when SWI *i* instruction executes; where *i* is a set of bits. The bits are used in instructions which calculate new starting address of ISR for SWI and calculate pointer to the ISR input parameters. ISR_VECTADDR is common to all values of *i*.



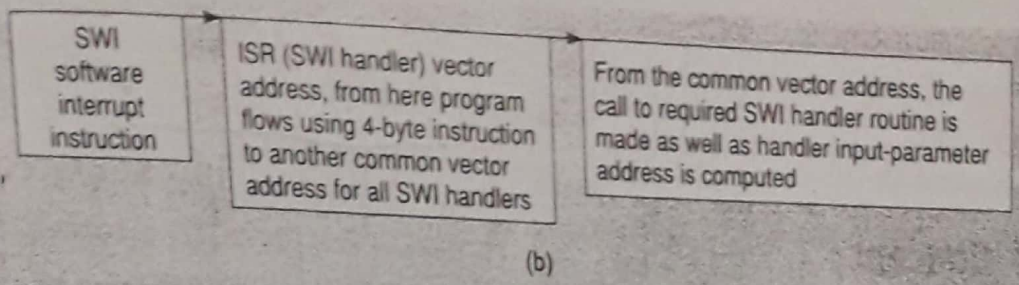


Fig. 6.16 (a) Generation of ISR_ VECTADDR when instruction INT *n* executes for interrupting the processor (b) Use of ISR_ VECTADDR in ARM for the jump to the routine for the interrupt servicing when SWI *I* instruction executes

Interrupt Vector Table

An Interrupt vector table is a concept used for programming for the service routines of each interrupt source. *Interrupt vector table* means a table of ISR_VECTADDR addresses for each interrupt source. The processor interrupts and execute ISR correspond to that address. A table facilitates locating the codes of the ISRs at ISR_VECTADDRs. Figure 6.17 shows concept of a vector table in memory in case of multiple interrupt-sources or source groups.

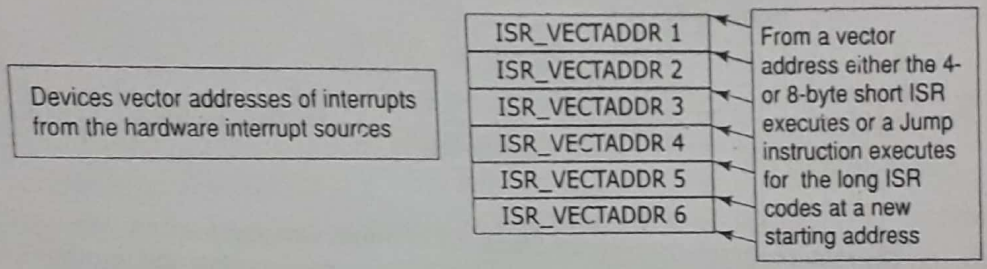


Fig. 6.17 Concept of vector table in memory in case of multiple interrupt-sources or source groups

An interrupt vector is an important part of interrupts-service mechanism, which associates a processing system. The processor first saves instruction pointer (program counter) and/or other registers of CPU on interrupt and then loads a vector address into the program counter. Vector address has (i) ISR, (ii) ISR address for the processor for interrupt source or interrupt type, or a group of sources. Interrupt vector table is an important part of interrupts service mechanism, which provisions for multiple interrupt sources and source groups.

Example 6.16

Consider a touch screen. It generates an interrupt when a screen position is touched. Interrupt activates a request IRQ to the touch-screen controller. A status bit b_t also sets. The b_t resets on start of ISR_{IRQ}. It is a service function (get_touch_position). It reads ADC input. The input gives the touched screen position using a look-up table. The look-up table consists of touched screen positions for each ADC input value.

Controller sends the touched position to the system. System has a table consisting of the previous command(s) and next command for each touched positions. Actions are taken as per the command.

6.8 MULTIPLE INTERRUPTS

6.8.1 Multiple Interrupt Calls

When there are the multiple interrupt sources, an occurrence of each interrupt source (or source group) is identifiable from a bit or bits in the status register and/or in the IPR.

There can be interrupt-service calls in case the higher priority interrupt source activates in succession. A return from any of the ISR is to the lower priority pending ISR.

Let us understand the processor interrupt-service mechanism for the case of multiple interrupts. There can be two types of processor actions to handle multiple interrupts.

1. Certain processors do not provide for in-between routine diversion to higher priority interrupts unless all interrupts or interrupts of priority greater than the presently running routine are masked. Diversion is called *context switching* of processor. Figure 6.18(a) shows diversion to higher priority interrupts only at the end of present interrupt-service routine.
2. Certain processors permit in-between routine diversion to higher priority interrupts. Figure 6.18(b) shows the actions in case the processor provides in diversion in-between the running ISR. There is provision for disabling or masking all interrupts by primary level bit when prevention is needed for in-between diversion. These processors also provide ways to prevent diversion in between the running ISR selectively by provisioning for masking selectively the interrupt service by secondary level bits for the ISR interrupt source groups.

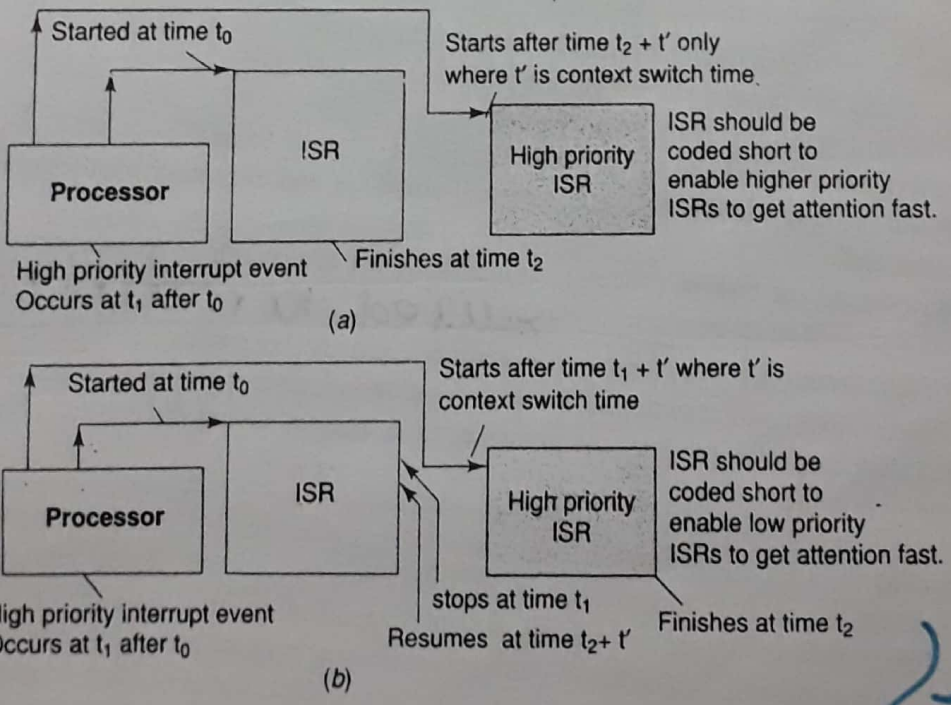


Fig. 6.18 (a) Diversion to higher priority interrupts, only at the end of the present interrupt service routine (b) In-between routine diversion to higher priority interrupts unless all interrupts or interrupts of priority greater than the presently running routine are masked

6.10 CONTEXT AND THE PERIODS FOR CONTEXT SWITCHING

Process of change of running program at the CPU to a new program is as follows:

1. Save the address (instruction pointer) from where the program will begin on return and save processor status word,
2. Save current program's registers, and other program parameters,
3. Find the address (instruction pointer) from where a new program begins,
4. Load the program's address into instruction pointer (program counter),
5. Load the new program's status word, registers, and other program parameters, and
6. Execute instructions of the new program. [Program means foreground program, process, thread, task, routine, ISR, signal handler or exception handler.]

Context of a program means, the address (instruction pointer) from where the program will begin on return, processor-status word, current program's registers, and other program parameters. Figure 6.20(a) shows current program context.

Steps 1 and 2 mean, saving the currently running program context. Steps 4 and 5 mean loading the new program context. Figure 6.20(b) shows steps on context switching when new program executes with new context.

Context saving is essential. The process ensures that (i) on return the saved program starts from same state as at the instance of change to new program, (ii) when new program starts then it also starts from same state as at the instance of earlier change from the program.

Context switching is performed in the system when

1. A foreground program interrupts and ISR starts execution,
2. When an ISR interrupts by higher priority ISR and new ISR starts,
3. When returning to previously running program,
4. When a signal is issued and signal handler executes,
5. When an exception is thrown on exceptional condition and exception handler (catch function) executes, and
6. When a thread (or task or process) starts waiting for a message or parameter and blocks, and system software starts new thread.

Figure 6.20(c) shows context switching to new routine and another switch on return to current routine. Figure 6.20(d) shows context switching for a new routine and another switch on higher priority routine.

Context switching period equals the processor time spent in saving the context plus time taken in loading the new context.

> Each running program has a context at an instant. Context reflects a CPU state [instruction pointer, stack pointer(s), registers and program state (variables that should not be modified by another routine)]. Context saving on the call of another program is essential before switching to another context. Context loading is essential so that a new one starts from the previously left context. Program means foreground program, process, thread, task, routine, ISR, signal handler or exception handler.

6.11 INTERRUPT LATENCY

When a processor interrupts the service of the interrupt by execution, the ISR may not start immediately after context switching. The interval between occurrence of interrupt and start of execution of the ISR is called interrupt latency.

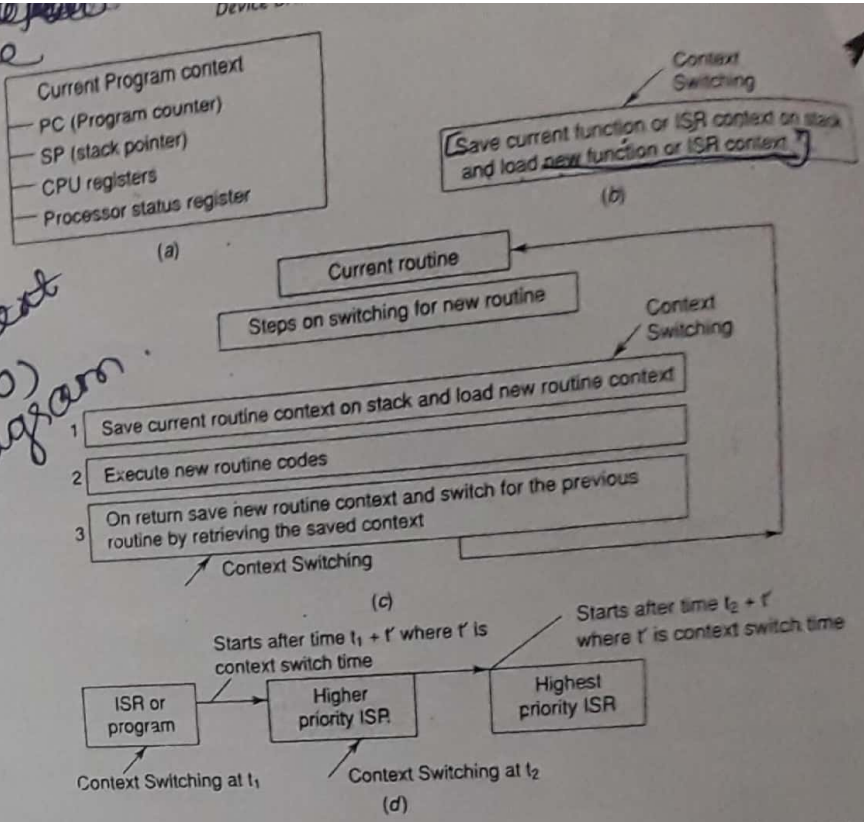


Fig. 6.20 (a) Current program context (b) Steps on context switching when new program executes with new context (c) Context switching to new routine and another switch on return to current routine (d) Context switching for new routine and another switch on higher priority routine

1. When the interrupt service starts immediately on context switching, the interrupt latency = T_{switch} = context switching period. When instructions in a processor takes variable clock cycles, maximum clock cycles for an instructions are taken into account for calculating latency. Figure 6.21(a) shows latency in case of interrupt service starts immediately.
2. When the interrupt service does not start immediately by context switching, but context switching starts after all the ISRs corresponding to the higher priority then the interrupts complete the execution and processes. If sum of the time intervals for completing the higher priority routines = ST_{exec} then interrupt latency = $T_{switch} + ST_{exec}$. Figure 6.21(b) shows latency in case of interrupt service starts after ISRs of higher priority, before the present interrupt finishes the execution.
3. Interrupt-system disabling instruction disables any other process of ISR from running when a routine enters a critical section and disabling instruction enables the processes or interrupts when the routine exits the critical section codes. The $T_{disable}$ is the period for which a routine is disabled in its critical section and may or may not be included depending upon the programmer's approach. The interrupt service latency from the routine with interrupt-disabling instruction (due to presence

PROGRAMMING IN ASSEMBLY LANGUAGE (ALP) AND IN HIGH-LEVEL LANGUAGE 'C'

7.1.1 Assembly-Language Programming

Assembly-language coding of an application has the following advantages:

1. Assembly codes are sensitive to the processor, memory, ports and devices. *Assembly software gives a precise control* of the processor internal devices, and makes *full use of processor-specific features* of processor instruction set and addressing modes.
2. Machine codes are *compact*, and are processor and memory sensitive. The system thus needs a smaller memory. No additional memory need due to data type's selection, conditions, and declarations of rules. A program is also not compiler specific and library-function specific.
3. Certain codes such as device-driver codes may need only a *few assembly instructions*. Assembly codes for these can be compact and precise, and are conveniently written.
4. *Bottom-up-design* approach is easily usable. Approach to this way of designing a program is as follows: First code the basic functional modules (submodules) and then use these to build a bigger module. Submodules of the specific and distinct sets of actions are first coded. Programs for delay, counting, finding time intervals and many applications can be written. Then the final program is designed by integrating the modules.

7.1.2 High-level Language Programming

High-level language coding of source files in C, C++, C#, Visual C++, or Java provides great programming ease, and has many advantages. Therefore, most embedded programming is in the high-level language. Basic advantages are as follows:

1. Program-development cycle is much shorter in high-level language even for complex systems due to the following: use of (i) routines (procedures) [called *functions* in C/C++ and *methods* in Java], (ii) standard library functions, and (iii) modular programming, top-down design and object oriented design approaches. Application programs in high-level language are structured. This ensures that the software is based on sound software engineering principles. Application programs are programmed using given operating system, file systems, device drivers and network drivers, and have access to devices by generic functions. Application program uses Application Program Interfaces (APIs), which makes task of development of an application simple.
2. Top-down-design programming approach, used in high-level language is as follows: Main program is designed first, then its modules, submodules, and finally, the functions.
3. A high-level language program facilitates declaration of data types. Type declarations simplify the programming. Each data type is an abstraction for the methods, which are permitted for using, manipulating, representing, and for defining a set of permissible operations on that data.
4. The program facilitates 'type checking'. This makes a program less prone to error. For example, type checking does not permit subtraction, multiplication and division on the char data types.
5. The program facilitates use of program-flow-control structures, such as loops and conditional statements.
6. The program has portability, and is not processor specific. Therefore, when hardware changes, only the modules for the device drivers and device management, initialisation and program-locator modules, and initial boot-up record data need modifications. OS takes care of these functions.

Additional advantages of C as a high-level languages are as follows:

It is a *language between low (assembly) and high-level language*. Inserting the assembly language codes in between is called *in-line assembly*. A direct hardware control is thus also feasible by in-line assembly, and the complex part of the program can be in high-level language.

Example 7.1

1. Square root is a library function. Use of standard square root ( ) function saves the programmer time for coding.
2. Four types of integers are *int, unsigned int, short* and *long*. When dealing with positive only values, data type variable is declared as unsigned int. Number of ticks to a clock, numTicks is an unsigned integer. Arithmetical calculations use signed integer, int (32-bit). An integer can also be declared as following data types, short (16-bit) or long (64-bit).
3. A data type is *char* for manipulating text or a string of characters.
4. Control structures widely used are *while, do-while, break* and *for* and conditional statements widely used are *if, if-else, else-if* and *switch-case*.
5. Type checking permits + operator to be used for concatenation when using char data types and lets + operator to be used for arithmetic addition when using *int, unsigned int, short* and *long* type of data. Concatenation operation can be understood as follows: The micro + controller concatenate into microcontroller, where *micro* is an array of *char* values and *controller* is another array of char values.

High-level language programming makes the program-development cycle short, enables use of the modular-programming approach, and lets us follow sound software-engineering principles. It facilitates the program development with top-down-design approach. Embedded system programmers have since long preferred C for the following reasons: (i) Feature of embedding assembly codes using in-line assembly, and (ii) Readily available modules in C compilers for the embedded system and library codes that can directly port into the system-programmer codes.

7.2 'C' PROGRAM ELEMENTS: HEADER AND SOURCE FILES AND PREPROCESSOR DIRECTIVES

A 'C' program has the following structural elements:

1. Preprocessor declarations, definitions and statements,
2. Main function, and
3. Functions, exceptions and ISRs.

A 'C' program has the following preprocessor structural elements.

1. Include directives for the file inclusion
2. Definitions for preprocessor global variables (global means all throughout the program module)
3. Definitions of constants
4. Declarations for global data types, type declaration and data structures, macros and functions

'C' program elements, header and source files and preprocessor directives are as follows:

7.2.1 Include Directive for the Inclusion of Files

Include is a preprocessor directive to include the contents (codes or data) of a file. The files that can be included are given below. Inclusion of all files and specific header files has to be as per requirements.

A 'C' program first includes the header and source files. These are readily available files. The purpose of each included file is mentioned in the comments within the /* and */ symbols as per practice in 'C'.

Example 7.2

```
# include "vxWorks.h"/* Include VxWorks functions */
# include "semLib.h"/* Include Semaphore functions Library */
# include "taskLib.h"/* Include multitasking functions Library */
# include "msgQLib.h"/* Include Message Queue functions Library */
# include "ioLib.h"/* Include File-Device Input-Output functions Library */
# include "sysLib.c" /* Include system library for system functions */
# include "netDrvConfig.txt"/* Include a text file that provides the 'Network Driver Configuration'. It
  provides the frame format protocol (SLIP or PPP or Ethernet) description, card description/make,
  address at the system, IP address (s) of the node (s) that drive the card for transmitting or receiving
  from the network. */
# include "prctlHandlers.c"/* Include file for the codes for handling and actions as per the protocols
  used for driving streams to the network. */
```

- **Including Codes Files** These are the files for the codes already available. For example, `# include "prctlHandlers.c"`.
- **Including Constant Data Files** These are the files for the codes and may have the extension '.const'.
- **Including Strings Data Files** These are the files for the strings and may have the extension '.strings' or '.str' or '.txt'. For example, `# include "netDrvConfig.txt"`.
- **Including Initial Data Files** Initial or default data files are for locating in ROM of embedded-system. Boot-up program is copied later into RAM and has extension, '.init'. On the other hand, RAM data files have the extension, '.data'.
- **Including Basic Variables Files** Files for the local or global static variables are stored in RAM. Variables do not possess initial (default) values. Static means that there is a common not more than one instance of that variable at memory address and it has a static memory allocation. There is only one real-time clock in the system. Therefore, only one instance of that variable address exists. These basic variables are stored in the files with the extension '.bss'.
- **Including Header Files** It is a preprocessor directive. It includes the contents (codes or data) of a set of source files. These are files for specific modules. A header file has the extension '.h'.

Example 7.3

1. A preprocessor directive is `# include <math.h>` for including standard library functions for mathematical operations in a program. Programs using mathematical expressions need mathematical functions, *square root, sin, cos, tan, atan* and so on. Functions become available by including a header file, called "*math.h*".
2. A preprocessor directive is `# include <string.h>` for including standard library functions for the strings in a program. Programs using strings need string-manipulation functions. These become available once a header file called "*string.h*" is included.
3. A program includes header files for the codes in assembly, I/O operations (conio.h), OS or RTOS functions. `# include "vxWorks.h"` is directive to compiler, which includes VxWorks RTOS functions. [Certain compilers provide for `conio.h` in place of `stdio.h`. This is because embedded systems usually do not need the file functions for opening, closing, read and write operations using the computer keyboard and video monitor. So including `stdio.h` file, makes the code too larger.]

What is the difference between inclusion of a header file, and a text file or data file or constants f
Consider inclusions of *netDrvConfig.txt* and *math.h*. (*i*) The header files are well tested and debu
modules. (*ii*) The header files provide access to standard libraries. (*iii*) The header file can inclu
several text file or C files, (*iv*) A text file is just a description of text for specific information.

7.2.2 Source Files

Source files are program files for the functions of application software. The source files need to be compiled, tested and validated. A source file also possesses the preprocessor directives of application software. The file has the *first function (main function) from where the processing will start*. The code in C for first function is *void main ( )*. The *main* function calls other functions.

7.2.3 Configuration Files

Configuration files are the files for the configuration of a subsystem. Device configuration codes can be put in a file of basic variables and included when needed. If these codes are in the file "`serialLine_cfg.h`", then # include "*serialLine_cfg.h*" *will be preprocessor directive*. Consider another example "*# include "os_cfg.h*". It will include os_cfg header file.

7.2.4 Preprocessor Directives

A preprocessor directive starts with a sharp (hash) sign. These commands are for the following directives to the compiler for processing.

- **Preprocessor Global Variables** For example, in a program "*# define* time"
- **Preprocessor Constants** "*# define* false 0" is a preprocessor directive in an example. It means it is a directive before processing to assume 'false' as 0. "*# define* pi 3.147"

Strings can also be defined. Strings are the constants, for example, those used for an initial displ on the screen in a mobile system. For example, *# define* welcome "Welcome To ABC Telecom".

Embedded C programs use preprocessor constants, variables, and inclusion of configuration files, text files, header files and library functions.

7.3 PROGRAM ELEMENTS: MACROS AND FUNCTIONS

Table 7.1 lists the uses and other features of programming elements called macros and functions.

Table 7.1 Uses of macros and functions

Program Element	Uses	Saves context on the stack before its start and retrieves them on return	Feasibility of nesting one within another
Macro	Executes a named small collection of codes.	No	None
Function	Executes a named set of codes with values or references to the values passed from the calling function through its arguments. Also returns data when function is not declared as void.	Yes, Each function has the context saving and retrieving overheads.	Yes, can call another function and can also be interrupted.

(Contd)

Table 7.1 (Contd.)

Program Element	Uses	Saves context on the stack before its start/and retrieves them on return	Feasibility of nesting one within another
Main Function	A function which runs first. Main has declarations of functions and data types, and typedef. Main function executes a named set of codes, calls a set of functions, and interrupts for executing ISRs or starts OS Kernel.	No	None
Reentrant Function	A function is called reentrant function when it enters into same state of variables and processor registers after in-between diversion to other functions or ISRs	Yes, has the context saving and retrieving overheads.	Yes to another reentrant function only
Interrupt Service Routine or Device Driver Function	Declarations of functions and data types, typedef, and executes a named set of codes. Must be short so that other sources of interrupts are also serviced within deadlines. Either must be a reentrant routine or must have a solution to the shared data problem. Not allowed to wait for inter process messages (semaphore or mailbox or queue messages) and allowed to send the interprocess messages.	Yes, has the context saving and retrieving overheads.	To ISR of higher priority unmasked sources
Process or Task or Thread	A set of instructions, which run under control of the OS. [Either must be a reentrant routine or must have a solution to the shared data problem.]	Yes, has the context saving and retrieving overheads.	None
Recursive Function	A function that calls itself. It must be a reentrant function also. Most often its use is avoided in embedded systems due to memory constraints.	Yes, has the context saving and retrieving overheads. Stack grows after each recursive call and stack overflow may choke the memory space available.	Yes

■ **Preprocessor Macros** A macro is a collection of codes that is defined in a program by a name. It differs from a function in the sense that once a macro is defined by a name, the compiler puts the corresponding codes for it at every place wherever that macro name appears.

Example 7.4

1. Consider two macros, 'enable_PortA_Intr ( )' and 'disable_PortA_Intr ( )'. [The pair of brackets in the macro is optional. It is present, it improves readability as it distinguishes it macro from a constant.] Whenever the name 'enable_PortA_Intr' appears, the compiler inserts the codes designed for it.
2. Test macros or test vectors are also designed during programming and used for debugging a system.

How does a macro differ from a function?

1. The codes for a function are compiled only once. Processor has to save the context on calling that function, and on return, processor restores the context. Further, a function may return nothing (*void* declaration case) or return a Boolean value, an integer, or any primitive or reference type of data. [Primitive means similar to an integer or character. Reference type means similar to an array or structure]. For example, the enable_PortA_Intr ( ) and disable_PortA_Intr ( ) are the function calls. [The brackets for the function calls are not optional.]
2. The codes for Macro are compiled in every function wherever that macro name is used. Compiler, before compilation, puts the codes at the places wherever the macro is used. When a macro runs in a function, the processor does not save the context, and thus does not have to restore the context, as there is no return in macro.
3. Macros are used for short codes only. This is because, if a function call is used for short codes instead of macro, the overheads (context saving and other actions on function call and return) take additional time, $T_{overheads}$. The time is the same order of magnitude as the time, T_{exec} for execution of short codes within a function. We use a function for codes when the $T_{overheads} << T_{exec}$ and a macro for codes when $T_{overheads}$ ~= or > T_{exec}.

Macros and functions are used in C programs. Functions are used when the codes should be compiled only once. However, on calling a function, the processor has to save the context, and on return, restore the context. Macros are used when short functional codes are to be inserted in a number of functions in a program.

7.4 PROGRAM ELEMENTS: DATA TYPES, DATA STRUCTURES, MODIFIERS, STATEMENTS, LOOPS AND POINTERS

7.4.1 Use of Data Types

Whenever data is named using an identifier, it will have the address(es) allocated at the memory. Number of allocated addresses for data depends upon data type. For example, if a data-type long is declared for numTicks (number of ticks), then numTicks will be allocated 4 memory-addresses.

The 'C' allows following primitive data types: *char* (8-bit) for characters, *byte* (8-bit), *unsigned short* (16-bit), *short* (16-bit), *unsigned int* (32-bit), *int* (32-bit), *long double* (64-bit), *float* (32-bit) and *double* (64-bit). [Certain compilers do not take the 'byte' as a data-type definition. The 'char' is then used instead of 'byte'. Most C compilers do not take a Boolean variable as data type. Then *typedef* is used to create a Boolean type variable in the C program. Embedded C compilers take a Boolean variable as data type.]

A data type appropriate for hardware is used. For example, a 16-bit timer can have only the unsigned short-data type, and its range can be from 0 to 65535 only.

The *typedef* is also used. It is made clear by the following example. A compiler version may not process the declaration as unsigned byte for 8-bit port data. If 'unsigned character' can be used as a data-type then it can then be declared as follows:

```
typedef unsigned character portAdata
#define Fbyte portAdata 0xF1
```

7.4.2 Use of Modifiers

A *modifier* modifies the actions of a data type.

Example 7.5

1. Modifier 'auto' or no modifier, if outside a function block, means that there is ROM allocation for the variable by program locator, if it is initialised in the program. RAM is allocated by the locator, if it is not initialised in the program.
2. Modifier 'unsigned' is modifier for a short or int or long data type. It is a directive to permit only the positive values of 16, 32 or 64 bits respectively.

7.4.3 Use of Pointers and Null Pointers

Pointers are powerful tools when used correctly and according to certain basic principles. A *pointer* is a reference to a starting memory address. A pointer can refer to a variable, data structure or function. In the 'C' language, symbol * is used before a pointer. For example, unsigned `char * 0x1000` means a character of 8 bit at the address 0x1000.

A NULL pointer declares the following: '*#define NULL (void*) 0x0000*'. [We can assign any address instead of 0x0000 that is not in use in a given hardware.]

7.4.4 Use of Data Structures: Stack, Queue, Array, List, Tree, Pipe, Table and Hash Table

A data structure is a way of organising several data elements of same types or different types. A data in a data structure can then be identified and accessed with the help of a few pointers and/or indices and/or functions.

A data structure is an important element of any program. A data structure saves a set of memory addresses in an organised way. Any data structure element can be retrieved. Few important data structures are *stack*, *one-dimensional array*, *queue*, *circular queue*, *pipe*, a *table* (two-dimensional array), *lookup table*, *hash table* and *list*. For example, the marks (or grades) of a student in the different subjects studied in a semester are put in a proper table. The table in the marksheet shows them in an organised way. Similarly, when there is a large amount of data, it must be organised properly.

Table 7.2 summarises the uses and show exemplary uses of queues, stacks, arrays, lists and trees.

Table 7.2 Uses of the various data structures in a program element

Data Structure	*Definition and when used*	*Example(s) of its use*
Stack	It is a structure with a series of elements with its last element waiting for an operation. An operation can be done only in the last in first out (LIFO) mode. It is used when an element is not accessible by any index or pointer directly, but only through the LIFO. An element can be pushed (inserted) only at the top in the series of elements and is still waiting for an operation. There is only one pointer used for pop (deleting) after the operation as well as for push (inserting). Pointers increment or decrement after an operation. It depends on push or pop.	(1) Pushing of variables on interrupt or call to another function. (2) Retrieving the pushed data onto a stack. [Note: Network stack and thread stack have different meaning than stack as data structure.]

(Contd.)

Table 7.2 (Contd.)

Data Structure	*Definition and when used*	*Example(s) of its use*
Queue	It is a structure with a series of elements with a header element waiting for a read operation, which is called deletion operation. An operation can be done only in the first in first out (FIFO) mode. It is used when an element is not accessible by any index and pointer directly, but only through the FIFO. An element can be inserted only at the end at the series of elements waiting for an operation. There are two pointers, one for deleting after the operation and other for inserting. Both increment after an operation.	Network stack frames ... network [Each frame ... queue of a stream of bytes ... byte has to be sent for des... receiving it as from a FIFO.
Array (one-dimensional vector)	It is a structure with a series of elements, with each element accessible by an identifier name and an index. Its element can be used and operated easily. It is used when each element of the structure is to be given a distinct identity by an index for easy operation. Index stars from 0 and is +ve integers.	Total_salary = 12 * salary [0]. Total salary is 12 times the first month salary, salary [0]. marks_max [4] = marks_max [0]; Maximum marks in the subject with index 4 is assigned the same as in the subject with index 0.
Multidimensional array	It is a structure with a series of elements each having another subseries of elements. Each element is accessible by identifier name and two or more indices. It is used when every element of the structure is to be given a distinct identity by two or more indices for easy operation. The dimension of an array equals the number of indices that are needed to distinctly identify an array element. Indices start from 0 and are +ve integers. [An image frame is an array of pixels with horizontal and vertical indices for each pixel# on screen.]	Handling a matrix or tensor. Consider a pixel in an image frame. Consider Quarter-CIF image pixel in 144 × 176 size image frame. *pixel* [108, 88] will represent a pixel at 108th horizontal row and 88th vertical column.#
List	Each element has a pointer to its next element. Only the first element is identifiable by list-top pointer (Header) does it. No other element is identifiable and hence is not accessible directly. By going through the first element, and then consecutively through all the succeeding elements, an element can be read, read and deleted, can be added to a neighboring element, or replaced by another element.	A series of tasks which are active. Each task has pointer of next task. Another example, ... menu that points to a submenu.
Tree	There is a root element. It has two or more branches each having a daughter element. Each daughter element has two or more daughter elements. The last one (leaf node) does not have daughters. Only the root element is identifiable and it is done by the treetop pointer (header). No other element is identifiable and hence is not accessible directly. Proceeding continuously by traversing from the root element through all the succeeding daughters, a tree element can be read or read and deleted, or can be added to another daughter or replaced by another element. A tree has data elements arranged in branches. A binary tree is a tree with a maximum of two daughters (branches) in each element.	An example is a directory. It has a number of file folders. Each folder has a number of other file folders, and so on. There is a file in the end.

Note: #pixel [0,0] represents the pixel at the left corner on top and pixel [144, 176] to the pixel at the right ... *pixel* [10,108, 88] is a pixel data element in a three-dimensional array form. It represents pixels position (108 × 88) in the 10^{th} frame.

7.5 Stack

A pointer which points to stack top is needed for handling each stack. It is called SP (Stack Pointer) or which points to the top of the stack and changes on each push or pop.

A processor has at least one stack pointer. Figure 7.1 shows two stacks created

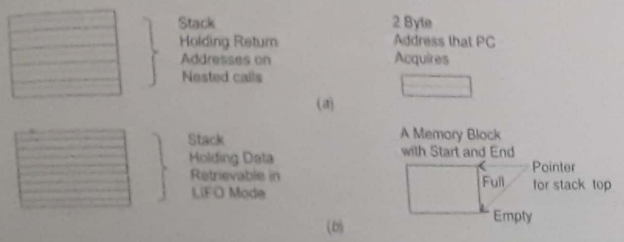


Fig. 7.1 (a) A stack due to nested function calls and pushing of program counters (b) Stack of pointers and parameter pushed on to stack for before the context switch

A processor stack pointer SP (stack pointer) points to instruction stack. The SP facilitates calling and return from the routines. A stack can also be a special data structure at the memory. It has a pointer address that points to the top of a stack. A value from the stack is retrieved from the memory in LIFO mode.

7.4.6 Multiple Stacks

Various stack structures may be created during processing. When a processor has only an SP, then memory addresses are used as stack pointers for each stack structure. Figure 7.2(a) shows multiple stacks pushed on the stacks each having separate pointer. Figure 7.2(b) shows multiple stacks of registers for the multiple tasks which are pushed on to stack.

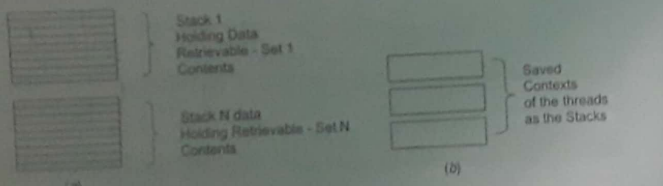


Fig. 7.2 (a) Multiple stacks pushed on the stacks each having separate pointer (b) Multiple stacks of CPU registers for the multiple tasks, which are pushed on to stack before context switches

Example 7.6

1. A processor may have several stack pointers. Advanced processors may provide for SP, RIP (Return Instruction Pointer), FP (data Frame Pointer) and PFP (previous Program Frame Pointer).
2. Motorola MC68010 processor provides USP (User Stack Pointer) and SSP (Supervisory Stack Pointer). MC68040 provides for USP (User Stack Pointer), SSP (Supervisory Stack Pointer), MSP (Memory Stack frames Pointer), and ISP (Instruction Stack Pointer). The program runs in two modes: user mode and supervisory mode. User functions execute in supervisory mode. Operating system functions execute in supervisory mode.
3. ARM processor does not have an SP. A GPR (generally r13) can be used as a stack pointer.

7.4.7 Array

A data structure, *array*, is an important programming element. An array has a multiple data elements, each identifiable by an index or by a set of indices. Index is an integer that starts from 0 to (array length – 1) in a one dimension. Data word can be retrieved from any element address in the block that is allocated to the array.

Example 7.7

(a) Consider `unsigned int [ ] phone_num` means an array of phone numbers, `phone_num [0]` refers to first phone number, `phone_num [1]` refers to second phone number, and so on. The `phone_num` itself points to first element.]

(b) Consider `unsigned char [ ] name` means an array of characters for the name. `name [0]` refers to first character, `name [1]` refers to second character, and so on. The `name` without index points to first array element.

(c) Consider the results of a test in a class with 30 students with roll numbers 1 to 30. Let *i* be an index used instead of roll number. Let marks in the test of roll number 1 be in the scalar integer variable, *M* [0]. Let *M* [0], *M* [1], ..., *M* [28] and *M* [29] be variables for the marks of roll numbers 1, 2, ..., 29, 30, respectively. A base pointer register points to the first scalar value M [0]. A register called index pointer may point to M[0]. Index register could then be incremented from 0 to 29 by an instruction within a loop to point to the marks of students of succeeding roll numbers.

(d) Figure 7.3 shows an array at a memory block with a pointer and index that jointly point to its element.

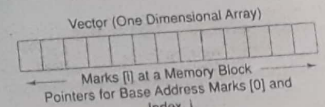


Fig. 7.3 Array at a memory block with one pointer for its base, first element with index = 0. Data word can be retrieved from any element by defining the pointer and index

7.4.8 Queue

The data structure, called *queue*, is another important programming element. The reading is with the help of indices in case of array, and first element address (identifier address). Therefore, any element can be read or written at any instance. Each element is read in queue from an address next

to the address from where queue element was last read. This reading is called *deletion*. It is written to an address next to the address from where queue element was last written. This writing is called *insertion*.

Figure 7.4 shows a memory block with the two pointers needed one for insertion and other for deletion.

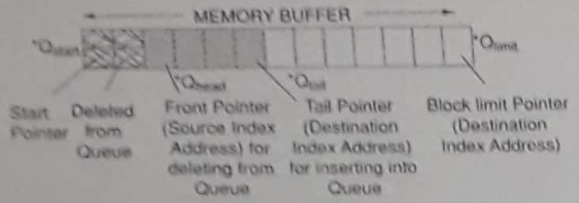


Fig. 7.4 Queue at a memory block with two pointers to point its two elements at the front (head, and back (tail). A data word always retrieves in FIFO mode from a queue

A queue is a data structure with an allotted memory block (buffer) from which a data element is always retrieved in FIFO mode. It has two pointers, one for its head and the other for its tail. Any deletion is made from the head address and any insertion is made at the tail address. An exception (an error indication) must be thrown whenever the pointer increments beyond the block end boundary so that appropriate action can be taken.

7.4.9 List

A *list* is for nonconsecutively located objects at the memory. A *list* is a data structure with a set of number of distinct sets of memory addresses, one for each element. A list has a top (head) pointer for the memory address from where the list starts. Each list element at the memory also stores a pointer to the next element at next set of addresses. The last element in the list points to *null*. Figure 7.5 shows the memory blocks with the pointers in a list.

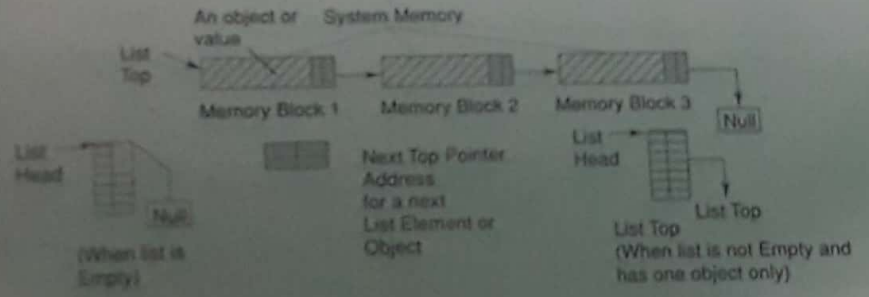


Fig. 7.5 Pointers in a list with elements at set of memory addresses

A list is a data structure in which each element stores a pointer to the next element at the list. It has a set of memory addresses allotted to each element. List-top pointer points to its first element and the last element points to null.

7.4.10 Circular Queue

A circular queue is a special queue. A pointer on reaching a limit $*Q_{limit}$, returns to its starting $*Q_{start}$. A print buffer and a display buffer are examples of circular queue. Each character is printed in FIFO mode. Figure 7.6 shows a memory block with a circular queue with its two pointers needed for insertions and deletions.

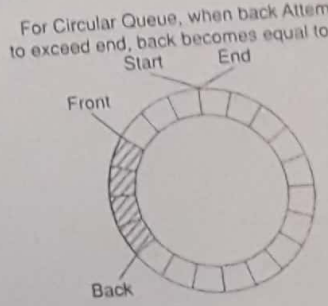


Fig. 7.6 Circular queue at a memory block with two pointers to point its two elements at the front and back. A pointer on reaching a limit of the block returns to the start of the block

A circular queue is a queue in which both pointers cannot increment beyond the memory block (buffer) and reset to starting value on insertion beyond the boundary.

7.4.11 Priority Queue

A *priority queue* is a special queue. Insertion takes place at the queue $*Q_{end}$ if priority of element is higher than previously inserted element, else at the $*Q_{end}$. Messages posted (inserted) from the tasks arrange in priority queue. Each message is taken (deleted) as per priority. Figure 7.7 shows a priority queue with elements arranged in priority order between memory addresses between head and tail pointers.

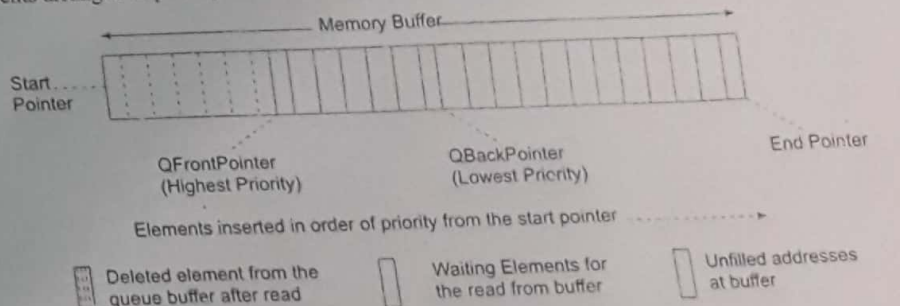


Fig. 7.7 Priority queue with elements arranged in priority order head and tail point

7.4.12 Use of Pipe

A *pipe* is a device, which uses device-driver functions. Insertions are from source-end and from the destination end. Source and destination are connected by a function pipe_connect( ) shows memory block used as data elements in a pipe.

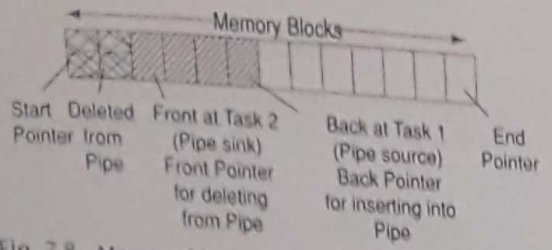


Fig. 7.8 Memory block used as data elements in a pipe

7.4.13 Use of Data Structures: Table and Hash Table

1. Table

A *table* is a two-dimensional array (matrix). It is an important data set in memory block. There is always a base pointer for a table. The base points to its first element at the first column, first row. There are two indices, one for a row and the other for a column. Figure 7.9 shows a memory block with the pointers for a table. Any element can be retrieved from three addresses for table base, column index and row index. When instead of a pointer, a value for specifying column and row indices is used in an instruction. That value is called *displacement*. Displacement can be for a column or row and is from the base.

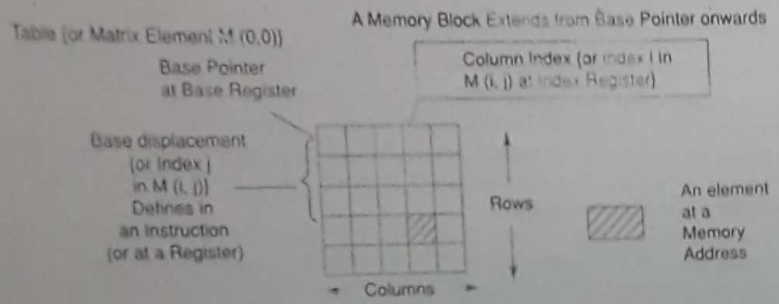


Fig. 7.9 A memory block with the pointers for a table

A table is a data set, allocated with a memory block Three pointers: *base*, *column* and *index* are used to access a data element.

2. Look-up Table and Hash Table

A look-up *table* is a two-dimensional array (matrix) and is an important data set. Each row has key and look at the key traces the addressed data. Look-up tables store like a hash. A table is called look-up table when the first column is used as a key (pointer to the value) and the second or succeeding columns as a value (s). A *hash table* is a data set that is a collection of pairs of a key and a corresponding value. A hash table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at nonconsecutive memory addresses. Figure 7.10 shows a memory block with the pointers for a look-up table for hash keys.

A hash table is a data set allocated with a memory block for key and value pairs. Just as an index identifies an array element, a hash key identifies a hash element. Table first column element specifies key and succeeding columns values associated with the key.

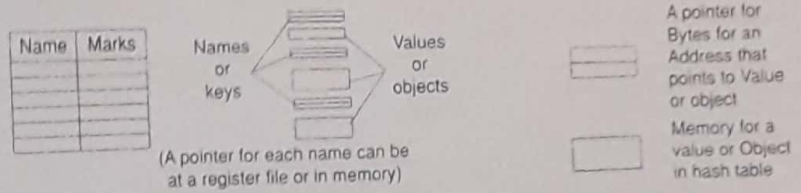


Fig. 7.10 Memory block with the pointers for a look-up table for hash keys

7.5 USE OF LOOPS, INFINITE LOOPS AND CONDITIONS

A set of statements is repeated in a loop each time with changed index value. Generally, in case of an array, the index changes and the same set is to be repeated for another element of the array. Then the loop is convenient to use. A loop starts from an initial value of a variable or condition and executes until the limiting value or condition is fulfilled. There can be certain parameter changing each time from its initial condition up to a limiting condition.

For example, consider the following, `for (i = 0; i < = 100; i++){/*` a set of statements which repeatedly execute`*/}`. The initial condition is assigned as $i = 0$ and last condition for loop to execute i is less or equal to 100. The set of statements in bracket executes from start to end and between return to start i increments by 1. The *for* statement lets the set of statements repeatedly executes 101 times with values of i =0, 1,..., 99, 100 and 101.

For another example, consider the following. `i = 0; while (i < = 100){/*` a set of statements which repeatedly execute `*/i++;}`. The initial condition is assigned as $i = 0$ and is set before the while loop. The while loop executes till i remains less or equal to 100. The i++ increments before the return and test for while condition takes place. The *while* statement lets the set of statements repeatedly executes with values of i =0, 1,..., 99, 100, 101.

If a condition remains true then while loop will execute infinitely until an interrupt source causes interrupt. For example, `while (1){/*` a set of statements which execute repeatedly execute `*/}`. The loop will execute infinitely because 1 always remains 1 (=true). Infinite loops are never desired in usual programming. This is because the function or task will never end and never exit or proceed further to the codes after the loop. *Infinite loop is a feature in embedded system programming!* The system software in the telephone has to be always in a waiting loop that finds the ring on the line on processor interrupt by external input. An exit from the loop except for servicing the interrupt will make the system hardware redundant.

Example 7.8 gives a 'C' program design in which the program starts executing from the main ( ) function. There are calls to the functions and calls on the interrupts in between. It has to return to the start. The system main program is never in a halt state. Therefore, the main ( ) is in an infinite loop within the start and end.

Example 7.8

```
# define false 0
# define true 1
/*************************************************************/
void main (void) {
/* The Declarations here and initialisation here */

/* Infinite while loop follows. Since the condition set for the while loop is always true, the statements
   within the curly braces continue to execute */
while (true) {
/* Codes that repeatedly execute */
.
.
}
```

Example 7.9 gives an example for use of polling for an event or message in a program.

Assume that the function *main* has a waiting loop and simply passes the control to an RTOS. Each task is controlled by RTOS and it will also have the codes in an infinite loop. Example 7.9 demonstrates the infinite loops within each task.

Example 7.9

```
# define true 1
/*************************************************************/
void main (void) {
/* Call RTOS run here */
rtos.run ( );
while (1) {
/* Infinite while loops follows in each task. So never there is return from the RTOS. */
}
}
/*************************************************************/
void task1 (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag1) {.....}; flag1 =0;
/* Codes that execute for posting message to the kernel */
message1 ( );
}
}
/*************************************************************/
void task2 (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag2) {.....}; flag2 =0;
/* Codes that execute for posting message to the kernel */
message2 ( );
};
}
/*************************************************************/
.
.
.
/*************************************************************/
void taskN (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flagN) {.....}; flagN =0;
/* Codes that execute for posting a message to the kernel */
messageN ( );
};
}
```


How does more than one infinite loop co-exist? Code inside the loop waits for an inter-process communication (IPC) message or event or a set of events. Operating system sends the IPC message.

An ISR or the code inside the loop of running task generates a message that transfers to the kernel. The OS kernel, which passes to the waiting task message, detects it and when that task starts, the OS preempts the previously running task.

Assume an event be setting of a flag, which trigger the running of a task whenever the kernel passes to the waiting task. The instruction SWI may be used send message to another task function for a service.

Conditional statements are used very often. If a defined condition(s) is fulfilled then statements within the curly braces after the condition (or a statement without the braces) is executed, otherwise program proceeds to next statement or to the next set of statements.

A set of statement is called switch case. A program switches to a case as per the result of switch expression result. For example, Switch (*i*) means switch as per the case for value of *i*. Example 7.10 also an application of infinite loop and switch-case statement for programming for GUI in mobile phone.

Example 7.10

Consider a smart mobile phone. Assume that screen state *j* is between 0 and *K*, among 0, 1, 2, or *K* - 1 possible states when number of sets for the menus = *K*. An interrupt occurs on a touch screen GUI, and therefore an ISR posts an event message *m* = 0, 1, 2, ..., or *N* - 1 as per the selected menu choice 0, 1, 2, ..., *N* - 1, when there are *N* menu choices for a mobile phone user to select from a screen in state *j*. The ISR posts *m* as per choice of user. *m* will depend on the screen position at the touched position. Figure 7.11 shows use of a programming model, which facilitates execution of one of the multiple possible function calls. A function executes after polling for screen state *j* and for a message *m*.

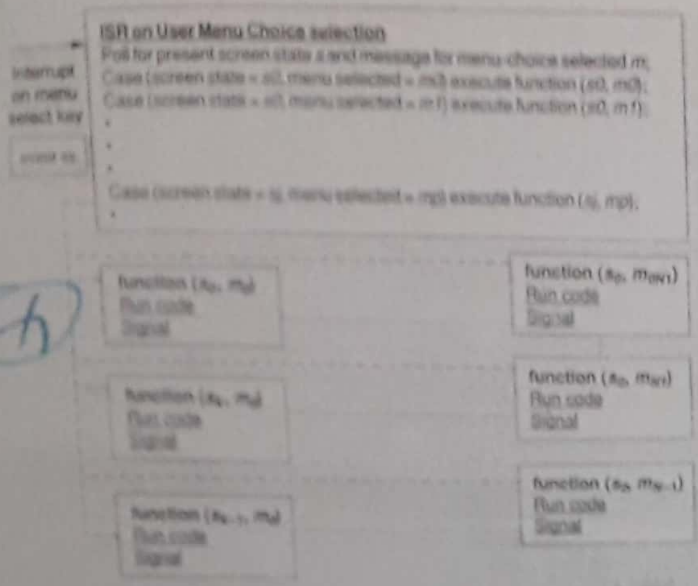


Fig. 7.11 Programming model for facilitating execution of one of the multiple possible function calls and a function executes after polling for the screen°state *j* and message *m* from an ISR

```
# define true 1
# define false 0
/*************************************************************/
void main (void) {
/* declarations */
while (true ) { /* Execute infinite loop */
poll_Screen_State ( ); /* Call a function to poll screen state. A state means a set of choices of menu
displayed on the touch screen */
}
/*************************************************************/
void poll_Screen_State ( )
/* Let number i identify a screen state */
Switch (i) {
Case 0: poll_menu0 ( ); exit ( )
Case 1: poll_menu1 ( ); exit ( )

Case j: poll_menuj ( ); exit ( )

Case K: poll_menuK ( ); exit ( )
}
}
/*************************************************************/
void poll_menu0 (/* Code for polling for choice from menu 0 */ )
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
```



```
Switch (m ) {
Case 0: {/*Code, which executes when the choice is menu 0 Screen state 0*/ ; exit ( );
Case 1: {/*Code, which executes when the choice is menu 1 Screen state 0*/ */ ; exit ( );
.
.
Case N - 1: {/*Code, which executes when the choice is menu N - 1 Screen state 0*/ */ ; exit ( );
 }
}
/*************************************************************/
. /* Codes for Screen state 1, 2, ..., j - 1 */
.
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
void poll_menuj (/* Code for polling for choice from menu m for screen state j */
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
Switch (m )
Case 0: {/*Code, which executes when the choice is menu 0 Screen state j*/ ; exit ( );
Case 1: {/*Code, which executes when the choice is menu 1 Screen state j*/ */ ; exit ( );
.
.
Case N - 1: {/*Code, which executes when the choice is menu N - 1 Screen state j*/ */ ; exit ( );
 }
}
/*************************************************************/
. /* Codes for Screen state j + 1, 2, ..., K - 2 */
.
void poll_menuK (/* Code for polling for choice from menu m for screen state K*/
}
}
/*************************************************************/
```

7.6 USE OF FUNCTION CALLS

A special function is for starting the execution of a program. It is '*void main (void)*'. Number of functions may be used in a program. Given below are the steps to be followed when calling a function in a program.

1. Declaring a Function

Just as each variable has to have a declaration, each function must be declared. Two variables for data at port A and flag are declared as follows: `unsigned char *portAdata; boolean charAFlag;` [Flag sets (=1) when a character is present at port and resets when not present]. Similarly, a function is declared as follows:

```
boolean checkPortAChar ( );/* An interrupt service function to return the flag, if there
is character received at port A */
void inPortA (unsigned char *); /* An ISR, which gets the input character at the
address portAdata */
```

2. Passing the Values (Elements)

When values pass from a calling function f1 to the called function f2, the values are copied from arguments in f1 into the arguments of f2. When the f1 passes the values, f2 executes in a way that it does not change variable's value at f1 during operations at f2. A function that has already passed the

7.11 OBJECTED-ORIENTED PROGRAMMING

When a large program is made, an object-oriented language offers many advantages. An *Objected-Oriented Programming (OOP) language* provides for the followings:

(i) defining of an object or set of objects, which are common or similar objects within a program and between many programs,
(ii) defining the methods that manipulate the objects without modifying their definitions,
(iii) creation of multiple instances of the defined object or set of objects or new objects,
(iv) inheritance,
(v) data encapsulation, and
(vi) design of reusable components.

An object can be characterised by the following.

1. An *identity* (a reference to a memory block that holds its state and behavior).
2. A *state* (its data, property, fields and attributes).
3. A *behavior* (method or methods that can manipulate the *state* of the object).

A procedure-based language, such as FORTRAN, COBOL, Pascal and C, are large programs that are split into simpler functional blocks and statements. In an object-oriented language like Smalltalk, C++ or Java, logical groups (also known as *classes*) are first made. Each group defines the data and the methods of using the data. A set of these groups then gives an application program. Each group has internal user-level data fields and methods of processing that data at these fields. Each group can then create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the user's data. The language provides for formation of classes by the definition of a group of objects having similar attributes and common behavior. A class *creates the objects. An object is an instance of a class.*

7.12 EMBEDDED PROGRAMMING IN C++

1. Advantages of C++

C++ is an *Object-Oriented Program (OOP) language, which in addition, supports the procedure oriented codes of C.* Program coding in C++ codes provides the advantage of objected-oriented programming as well as the advantage of C and in-line assembly. Programming concepts for embedded programming in C++ are as follows:

(i) A class binds all the member functions together for creating objects. The objects will have memory allocation as well as default assignments to its variables that are not declared *static*. Let us assume that each software timer is an object. It gets the count input from a real-time clock. It has a terminal count value after which it generates a software interrupt. It is initialised to a count value. Now consider the codes for a C++ *class RTCSWT*. A number of software timer objects can be created as the instances of *RTCSWT*. Each instance of RTCSWT can have different values of present, initial and terminal counts but has identical methods to manipulate the count.

(ii) A class can derive (inherit) from another class also. Creating a *child* class from RTCSWT as a *parent* class creates a new application of the RTCSWT.

(iii) Methods (C functions) can have same name in the inherited class. This is called *method overloading*. Methods can have the same name as well as the same number and type of arguments in the inherited class. This is called *method overriding*. These are the two significant features that are extremely useful in a large program.

(iv) Operators in C++ can be overloaded like in method overloading. The operators '++' and '!' can be overloaded to perform a set of operations. [Usually the '++' operator is used for post-increment and pre-increment and the '!' operator is used for a *not* operation.]

```
const OrderedList & operator ++ ( ) {if (ListNow != NULL)
ListNow = ListNow -> pNext;
return *this;}
boolean int OrderedList & operator ! ( ) const {return
(ListNow != NULL) ;};
```

[Java does not support operator overloading, except for the '+' operator. It is used for summation as well string concatenation.]

There is *struct* that binds all the member functions together in C. But a C++ *class* has object features. It can be extended and child classes can be derived from it. A number of child classes can be derived from a common class. This feature is called *polymorphism*. A class can be declared as public or private. The data and methods access is restricted when a class is declared private. *Struct* does not have these features.

2. Disadvantages of C++

The program codes become lengthy, particularly when certain features of the standard C++ are used. Examples of these features are as follows:

(a) Template
(b) Multiple Inheritance (Deriving a class from many parents)
(c) Exceptional handling
(d) Virtual base classes
(e) Classes for I/O Streams [Two library functions are *cin* (for character (s) in) and *cout* (for character (s) out)]. The I/O stream class library provides for the input and output streams of characters (bytes). It supports *pipes*, *sockets* and *file-management features*.

Embedded system programmers use C++ due to the OOP features of software re-usability, extendibility, polymorphism, function overriding and overloading along with the portability with the C codes and in-line assembly codes. C++ also provides for overloading of operators. Embedded C++ is a C++ version, which makes large program development simpler by providing Object-Oriented Programming (OOP) features of using an object, which binds state and behavior and which is defined by an instance of a class. We use objects in a way that minimises memory needs and run-time overheads in the system.

7.13 OPTIMISATION OF CODES AND MEMORY NEEDS IN EMBEDDED C++ PROGRAMS TO ELIMINATE THE DISADVANTAGES

Embedded system codes can be optimised when using an OOP language by the following:

(a) Declare *private* as many classes as possible. It helps in optimising the generated codes.
(b) Use *char*, *int* and *boolean* (scalar data types) in place of the objects (reference data types) as arguments and use local variables as much as feasible.
(c) Recover memory already used once by changing the reference to an object to NULL.

A special compiler for an embedded system can facilitate the disabling of specific features provided in C++. **Embedded C++** is a version of C++ that provides for a selective disabling of the above features so that there is a less runtime overhead and less runtime library. The solutions for the library functions are available and ported in 'C' directly. The I/O stream library functions in an embedded C++ compiler are also reentrant. So using embedded C++ compilers or the special compilers make the C++ a significantly more powerful coding language than 'C' for embedded systems.

The GNU C/C++ compilers (called gcc) find extensive use in the C++ environment in embedded software development. Embedded C++ is a new programming tool with a compiler that provides a small runtime library. It satisfies small runtime RAM needs by selectively de-configuring features like, template, multiple inheritance, virtual base class, etc., when there is a less runtime overhead, and when the less runtime library using solutions are available. Selectively removed (deconfigured) features could be template, runtime type identification, multiple inheritance, exceptional handling, virtual base classes, I/O streams and foundation classes. [Examples of foundation classes are GUIs (Graphic User Interfaces). Exemplary GUIs are the buttons, checkboxes or radios.]

An embedded system C++ compiler (other than gcc) is *Diab* compiler from *Diab Data*. It also provides the target (embedded system processor) specific optimisation of the codes. The runtime analysis tools check the expected runtime error and give a profile that is visually interactive.

7.14 EMBEDDED PROGRAMMING IN JAVA

7.14.1 Java Programming Basics

Java programming starts from coding for the classes. A class has members. A field is like a variable or struct in 'C'. A method defines the operations on the fields, similar to function in 'C'. Table 7.3 summarises basics uses and show exemplary uses. Instance fields and instance methods of class are members, whose new instances are also created as when the objects are created from the class. Class is a named set of codes that has a number of members such as data fields (variables), and methods (functions). Class is used to create objects with instances of these members. The operations are done on the objects by passing the messages to the objects in object-oriented programming. Each class is a logical group with an identity, a state and a behavior specification.

Table 7.3 Java program elements

Java Program Element	*Explanation*	*Example(s) of its use*
Local Variable	A variable within a block of codes that is defined inside the curly braces and has limited scope.	`{for (int i = 0; int totalOfMarks = 0; i<5; i++) {totalOfMarks += subjectMarks[i];} return totalOfMarks;}`. Here, *i* is local variable. The *i* does not have any scope outside the 'for loop'.
Instance Method	Blocks of Java codes, which are given a name, a call (invocation) is made by other Java codes that can also pass (transmit) the needed reference to the values, parameters, etc.	findTotalMarks ( ) { }

(Contd.)

Table 7.3 *(Contd.)*

Java Program Element	*Explanation*	*Example(s) of its use*
Instance Field	An identifier with a name using which, a declaration is made in a Java class. It does have a default value and the field is also present in the objects which are instances of the class.	`String tele_number;` here, tele_number is an instance field of the class and will also be in the objects created from that class.
Class	A class is a basic structural unit in a Java program. A class consists of data fields and methods that operate on the fields. A class is a group of objects with similar attributes and common behavior and relationships. A class is used to create objects as its instances.	public class Salaries { public float monthly salary, totalSalary; public float findTotalSalary ( ) { };}
Inheritance	Java Class inherits members when a Java class is extended from a parent class called super class. The inherited instance fields and methods can be overridden by redefining them in extended class. Methods can be overloaded by redefining them for different number of arguments.	`public class AccountDetails extends BankDetails {…}`. The class AccountDetails will inherit members of class BankDetails.
Interface	Interface has only abstract methods and corresponding data fields. Methods do not have implementation at the Interface. A Java class which is interfaced to an Interface, implements the abstract methods specified at the Interface.	public class AccountDetails extends BankDetails implements InterestComputations{…}.
Data Types	Java Class uses primitive data types: Byte (8-bit), shot (16-bit), int (32-bit, long (64-bit, float, double, char (16-bit). Java Class uses reference data types. A reference can be Class type in which there are groups of fields and methods to operate on the fields. A reference can be array type in which there are groups of objects as array elements.	byte portData; /* 8-bit port data */ short counts; /* 16-bit count data */ int numTicks; /* 32-bit number of clock ticks */ String accountNum, eMailId; /* account Number and email ID as String class objects */
Exception	Java has built-in exception classes. The occurrences of exceptional conditions are handled when exception is thrown. It is also possible to define exception conditions in a program so that exceptions are thrown from try block codes and caught by catch-exception method.	`java.lang.ArrayIndexOutOfBoundExceptions: 0 at addArray ( …`. This throws an exception sue to javal.lang packagae has an Object java.lang.Throwable. We can also define exceptions in try {…} catch (Exception e1){ } Finally { };

7.14.2 Java Programming Advantages

Java has advantages for embedded programming as follows:

1. Java is completely an OOP language. Java program starts with classes. Application program consists of classes and interfaces.
2. There is a huge class library on the network that makes program development quick.
3. Java has extensibility.
4. Java has in-built support for creating multiple threads. It obviates the need for an operating system (OS) based scheduler for handling the tasks.
5. Java generates the byte codes. These are executed on an installed JVM (Java Virtual Machine) on a machine. Virtual machine takes the Java byte codes in the input and runs on the given platform [(processor, system and OS);[Virtual Machine (VM) in embedded systems is stored at the ROM.] Therefore, Java codes can host on diverse platforms. Platform independence in hosting the compiled codes permit Java for network applications.
6. Platform independence gives *portability* with respect to the processor and the OS used. Java is considered as write once and run anywhere.
7. Java is the language for most Web applications and allows machines of different types to communicate on the Web.
8. Java is easier to learn by a C++ programmer.
9. Java does not permit pointer manipulation instructions. So it is robust in the sense that memory leaks and memory related errors do not occur. A memory leak occurs, for example, when attempting to write to the end of a bounded array.
10. Java does not permit dual way of object manipulation by value and reference. There are no struc, enum, typedef and union. Java does not permit multiple inheritances and operator overloading, except for + sign used for string concatenation.

7.14.3 Disadvantages of Java

Java has following disadvantages for embedded programming as follows:

1. Since Java codes are first interpreted by the JVM, it runs comparatively slowly. Java byte codes can be converted to native machine codes for fast running using Just-In-Time (JIT) compilation.
2. Java byte codes that are generated need a larger memory. An embedded Java system may need a minimum of 512 kB ROM and 512 kB RAM because of the need to first install JVM and run the application.

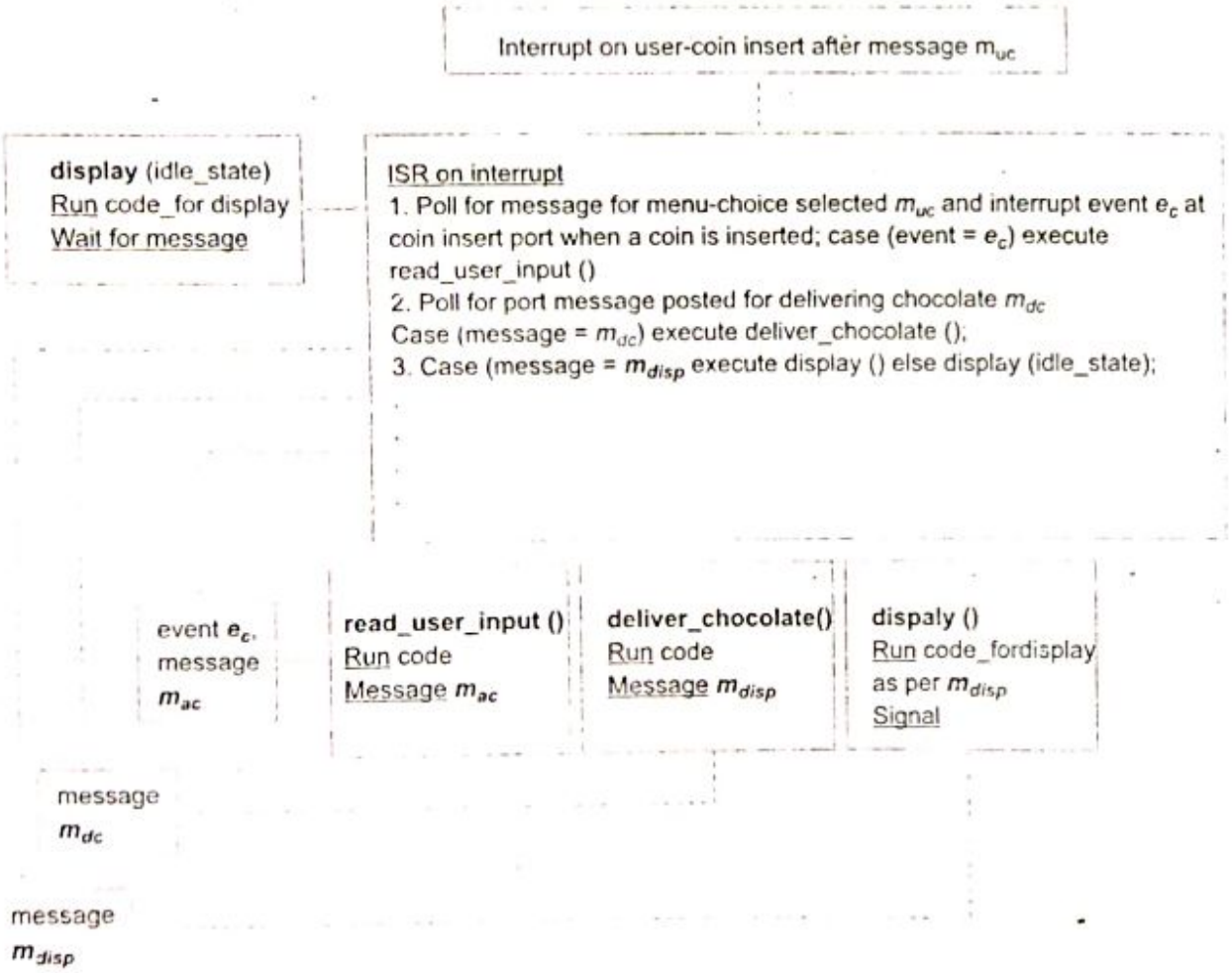
Java objects bind state and behavior by the instance of a Java class. Java has large number of readily available classes for the I/O stream, network and security. Object-oriented features and ready availability of classes make a large program development simpler task. JVM is configured to minimise memory needs and runtime overheads in the system. Java programs possess the ability to run under restricted permissions.

1. Polling for Events-Based Model

A program model is polling in a cyclic loop. Polling is for the events, state variables, messages and signals. Polling is performed using switch-case statements. A function is called for each event, state, message or signal found in the loop.

Figure 8.1 shows a polling for events-based model for an automatic chocolate-vending machine (ACVM). The following functions run on events

1. A poll for interrupt event from coin-insert port when a coin is inserted, and if event found then run ISR read_user_input ( ) for obtaining input for the choice of chocolate from the customer.
2. A poll for port message posted for delivering chocolate and if message found then run deliver_chocolate (port_message).
3. A poll for display message posted for display and if message found then run function display (display_message), else display (idle state message).



Polling for events-based program model in ACVM example

2. Sequential Program Model

A sequential programming model is to execute multiple function calls within a function in a sequential order. ISRs provide short period deviations from the sequence, execute short codes for servicing the interrupts and send the function pointers as messages inserted into the queue. Even then, the pointed functions are executed in sequential first-in first-out (FIFO) order.

Example 8.2

Figure 8.2 shows a sequential program model for an automatic chocolate-vending machine (ACVM). The following functions run in sequence.

1. Run function get_user_input ( ) for obtaining input for the choice of chocolate from the customer.
2. Run function read_coins ( ) for reading the coins inserted into the ACVM for the cost of chocolate.
3. Run function deliver_chocolate for delivering the chocolate.
4. Run function display_thanks for displaying 'Collect the nice chocolate. Visit again!'

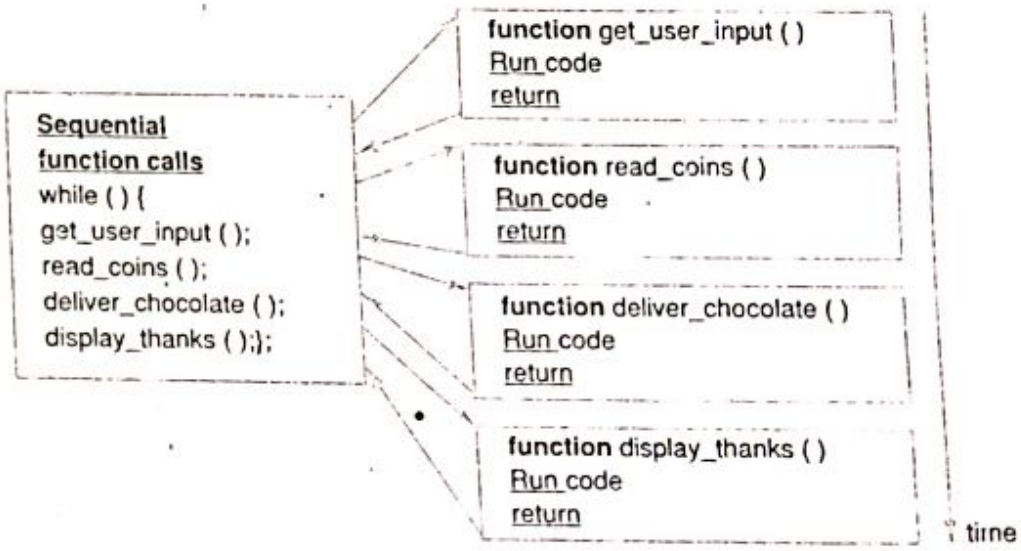


Fig. 8.2 Sequential Programming Model of an ACVM

3. Concurrent Processes and InterProcess Communication Model

A programming model is that which has several concurrent tasks (or processes or threads) and each task has sequential codes in an infinite loop. An OS controls the order of priorities for execution or controls the time slice allotted for execution of a task. A task posts an interprocess message or signal to OS, which passes it to another task waiting for that message or signal. A task, which gets a message or signal from the OS, runs and remaining tasks remain in blocked (wait) state. Example 8.3 gives the concurrent process model for the program model in Example 8.2.

Example 8.3

Figure 8.3 shows a program model based on concurrent running of the processes in an ACVM. Assume that the program consists of the following processes, which run concurrently.

1. Process `get_user_input ( )` for a user interrupt service. It obtains input for the choice of chocolate from the child. It posts interprocess signal for process read_coins.

2. Process `read_coins( )` waits for the signal of `get_user_input( )` and when OS signals, it starts and read coins inserted in the ACVM for the cost of chocolate. It posts an interprocess signal for process `deliver chocolate ( )` and also posts a signal to process `display_wait( )` to start.
3. Process `deliver_chocolate( )` waits for signal of `read_coins( )` and when OS signals, it starts and delivers the chocolate as per choice input at step 1. It posts an inter process signal for `display_thanks ( )`.
4. Process `display_wait( )` waits for signal of `read_coins( )` and when OS signals, it starts displaying 'Wait few moments!'
5. Process `display_thanks( )` waits for signal of `deliver_chocolate ( )` and when OS signals, it starts displaying 'Collect the nice chocolate. Visit again!'

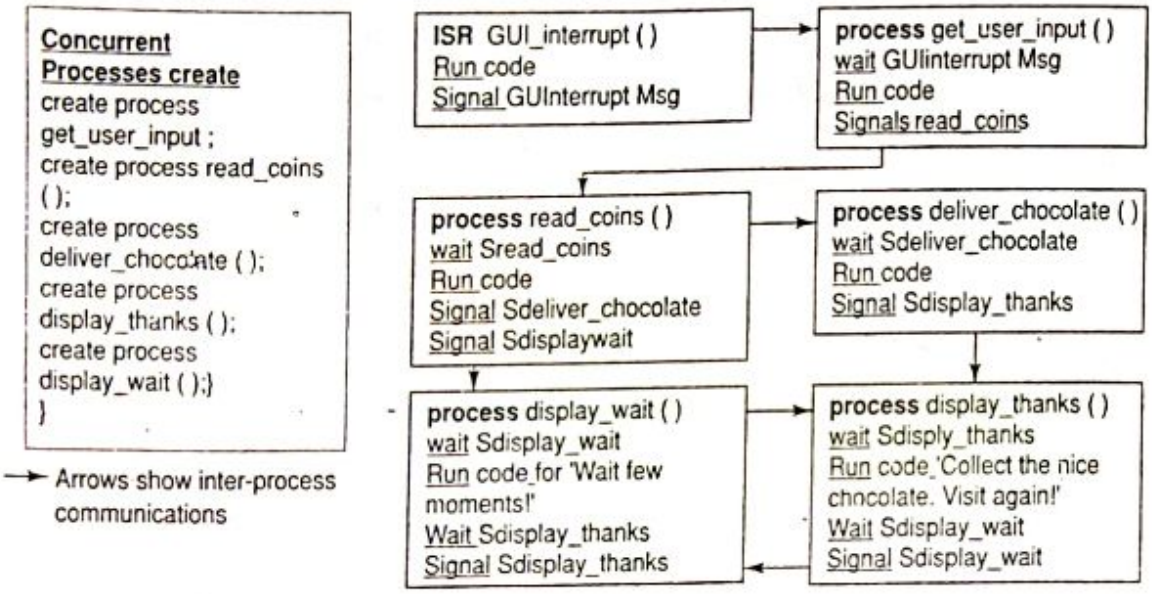


Fig. 8.3 Concurrent processing programming model of an ACVM

4. Object-Oriented Programming Model

An object is characterised by its identity (a reference to it that holds its state and behavior). State of object means its data, property, fields and attributes. Behavior means operations, method or methods, which can manipulate the state. Objects are created from the instances of a class. Defining the logically related group makes a class.

Class defines the state and behavior. It has internal user-level fields for its state and behavior. It defines the methods of processing the fields. A class can thus create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the states as per the defined behavior. A set of classes and their objects then create an application program. An object-oriented language is used for the following features:

(a) Data encapsulation within an object
(b) Re-usable objects or set of objects defined, that are common within a program or between the many applications
(c) Abstraction of data fields and methods in a class
(d) Creation of new objects creation a inherited class, which extends or redefines or overrides data fields and methods of a class.
(e) Creation of new objects using polymorphism.

Example 8.4

An object-based model is used instead of ACVM sequential program and processes-based models. Figure 8.4 shows the features of classes objects, and inheritance interface in a model for an ACVM. The following can be the classes and objects.

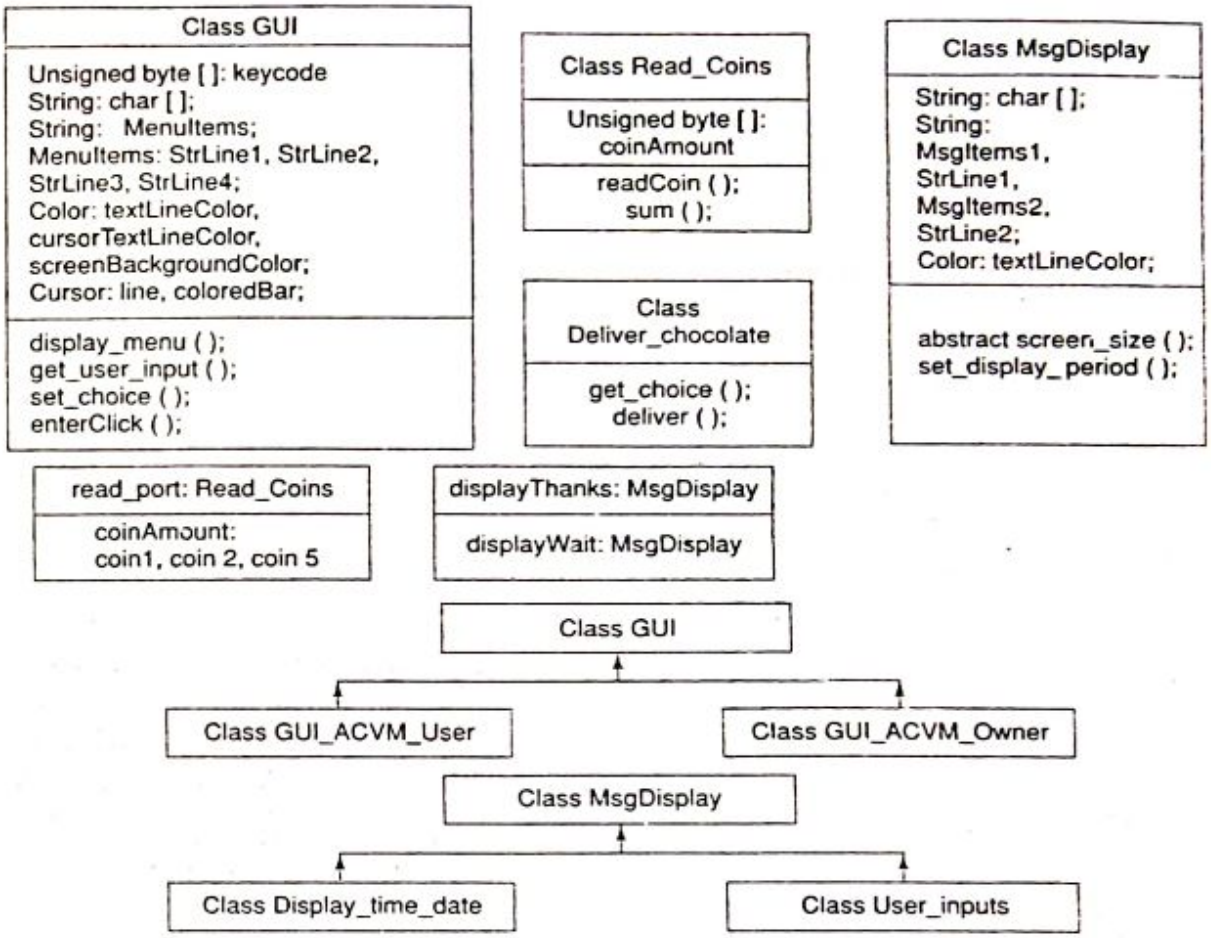


Fig. 8.4 Classes and objects and inheritance and interface features in a program model based for the ACVM

1. `Class GUI` for graphic-user interaction. It has two methods, `display_menu ( )` and `get_user_input ( )` for obtaining input for the choice of chocolate from the customer. It has the method `set_choice ( )` to set the choice selected.
2. `Class Read_Coins ( )` for reading the coins inserted. It has a method `readcoin ( ). readcoin ( )` reads one, two and five rupee coins from three ports and a method `sum ( )` for summing the total coins.
3. `Class Deliver_chocolate`. It has methods, `get_choice ( )` to get the choice and `deliver ( )` for delivering the chocolate.
4. `Class MsgDisplay`. It has methods `display_wait ( )` and `display_thanks ( )` for display wait message and thank message.

8.2.3 Synchronous Data Flow Graph (SDFG) Model

When there are number of tokens (inputs) required for a computation to generate number of tokens (outputs) in a single firing, the data flow is said to be synchronous. The SDFG model is as follows. Let an arc represent a buffer in physical memory. The arc can contain one or more initial tokens with the delays. A token, till it is received at the vertex, does not fire the computations at a vertex. *Vertices (circles) in this graph are called the actors. Actors do the computations.* An actor also represents a complete DFG within itself. *An edge between the vertices (arcs with an arrow for the direction) represents a queue of output values from one vertex and a queue of input values to another vertex. Edges carry the values from one actor to another.*

Example 8.7

Let *X* and *Y* be two sets of instructions that once fired (started), and do not need any further inputs from any source during the computations. Let *X* generate the output values (tokens/data) *a*, *b* and *c*. Let *Y* get the input values (tokens/data), *a*, *c*, *i* and *j* and let *i* have a delay. The number of inputs to *Y* need not equal the number of outputs from *X*. *Y* gets additional inputs and does not need all the outputs from *X*. These computations and data are now modeled by a directed data flow graph that exists from *X* to *Y*. The number of outputs and inputs are labeled near the arc-origin and arc-end.

Figure 8.9 shows actors (vertices, which does the computations on firing) and arcs in a directed graph between *X* and *Y*. The figure shows the outputs *a*, *b* and *c* and inputs *a*, *c*, *i* and *j*. The *i* is with a delay (dot). The dot on an arc represents the initial token(s) in an SDFG model. Then an initial token may also represent a *delay* that is shown by a dot on the edges of SDFG. If there are more than one initial token the number of initial tokens are mentioned on the dot. The *i* and *j* are initial tokens for the vertex *Y* showing that *i* has a *delay*.

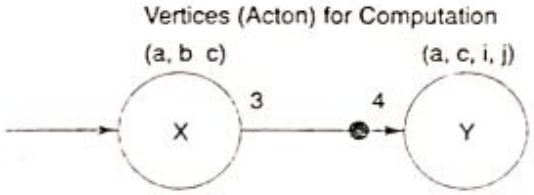


Fig. 8.9 Actors and arcs in a directed graph between *X* and *Y*, outputs *a*, *b* and *c* at *X*, and inputs *a*, *c*, *i* and *j* at *Y*; *i* is with a delay (dot)

A number of vertices may be present in a system. *All computations are static scheduled in SDFG execution at each vertex (firing elements for the computations and creating another set of output tokens). SDFG model program translates into a sequential model program.*

An SDFG model is like a DFG, but also models the delays as well as the number of inputs and outputs. The edges directed to the circle can be assumed to have a physical memory buffer and until the buffer has the data, the computations do not fire.

8.3 STATE-MACHINE PROGRAMMING MODELS FOR EVENT-CONTROLLED PROGRAMS

A program's output or actions for present input may depend on the previous input and output conditions. It means the previous state is also input along with the new input to determine the program's next state.

A program model means that there are different states and the model considers a system as a (which is producing the states one after another until it returns to the initial state.

8.3.1 State-Machine Programming Model

A state-machine is a model in which it is assumed that there are states and state transition functions, which produce the states. A state transition function is a function which changes a state to its next state.

Example 8.8

1. A display may have different states. A state corresponds to a displayed menu, and the program action depends on the previous display state (menu). The program is sequentially polled for the screen state and menu choice selected by the user.
2. A mobile phone has nine keys marked w, e, r, s, d, f, z, x, c as well as 1, 2, 3, 4, 5, 6, 7, 8, 9. When a phone number is dialed, the keys interpret as 1, 2, ..., 8 and 9. When SMS message is keyed-in then a key inputs a number, 1, 2, ..., 8 or 9 if marked * is keyed-in before else inputs an alphabet w, e, ..., x or c.
3. If case-shift key 'aA' is keyed in then an upper case alphabet is output on keying-in a character previous state is lower case alphabet. Else, vice-versa shift of Case takes place.
4. A telephone system has five finite states, *Idle*, *Receiving a ring*, *Dialing*, *Connected* and *Exchangi messages*.
   a. Consider an example of the running state in a timer. The count-input is the clock-input. T changed count value is the output. The output function is the increment in the count val The state transition function is the time-out on overflow when a predetermined numbers count-inputs are reached. A timer has four finite states: 'Idle', 'Start', 'Running' and 'finish
   (a) *'Idle' State*: It starts state transition on loading an input, *numTicks* (number of ticks at which the timer finishes).
   (b) *'Running' State*: On each clock input for decrement, the count value decrements.
   (c) *'Finish' State*: Program flows to finished state. This is when the count value reaches 0.
5. A task has four finite states — *idle, ready (waiting), running and finished*. An output from one state becomes the input to next state. A token from OS scheduler changes a task state.

When is a system modeled as the states and state machine? Frequently, there are inputs to a prog that change the state of a system to a new state, and generate outputs, which may also be the inputs for the next state. Now, it can be assumed that in a model the running of the program and its flow can be considered as running of a machine which generate the states. The program flow can be modeled simply by interstate transitions (from one state to another) from next state transition functions (Moore model) or next output state transition functions (Mealy model).

There can be transition of the present state to the next state, which depends on the inputs and state transition function. A set of outputs represents a state in Moore model and a set of outputs represents a state transition in Mealy model.

Let a circle represents a state and let a directed arc (or an arrow) represent the program flow from a state to another.

The steps that model or represent the *states* and *interstate transitions* in a data path are as follows.

1. A transition to a new state occurs from the previous state *on an event (input)*. The event may be setting a value of certain parameter or the result of the execution of certain codes. A transition may also be interrupt-flag driven (after a flag sets), semaphore driven or interrupt-source servicing-need driven.
2. A state can receive multiple tokens (inputs, messages, flags interrupts or semaphores) from another state(s). *A token (event) is used here as a general term that means either an input or event-input.*

An event-input characteristic is that it is asynchronous (one never knows when an event may happen). An event-input may happen when there is setting or resetting of a flag. It may occur when there is (*i*) a semaphore given or taken, or (*ii*) some indication for a resource or signal or data-item generated, or (*iii*) completion of execution of a set of codes.

3. A state can generate multiple tokens (outputs, messages, flags interrupts or semaphores). An output or set of outputs and variables identify a next state on mapping the inputs, variables and previous states using the output-state transition (action) function (Mealy model). A flag indicating sate condition or a set of codes being executed or a set of values of certain parameters identifies a next state on mapping the inputs, variables and previous states using the next-state transition function (Moore model).

The state machine model is a model in which running program and its flow can be considered as running of a machine, which generates states. The program flow can be modeled simply by interstate transitions (from one state to another) from next state transition functions (Moore model) or next output state transition functions (Mealy model). A circle represents a state and running program codes and a directed arc (or an arrow) represent the program flow in one state to another.

8.3.2 Finite States Machine (FSM) Model

The FSM model states that there is finite number of possible states in a system, and a system can only be in one of these states at an instance.

When modeling a process as finite state machine (FSM), the software designer specifies the following for each state.

1. A state among one of a finite number of states.
2. Finite set of inputs (tokens, event flags or status flags) with their values for the state.
3. Finite actions (for example computations) during the state and finite set of outputs with their possible values (or tokens, event flags or status flags) and an output (action) *function* for the state that gives the outputs.
4. *State transition function* for each state to take it to the next state.

Example 8.9

Figure 8.10 shows the FSM states in a program model of an ACVM. AVCM has four finite states—get_user_input ( ), read_coins ( ), deliver_chocolate and display_thanks. An output from one state becomes the input to next state. A signal from OS scheduler or interrupt from user changes an ACVM state.

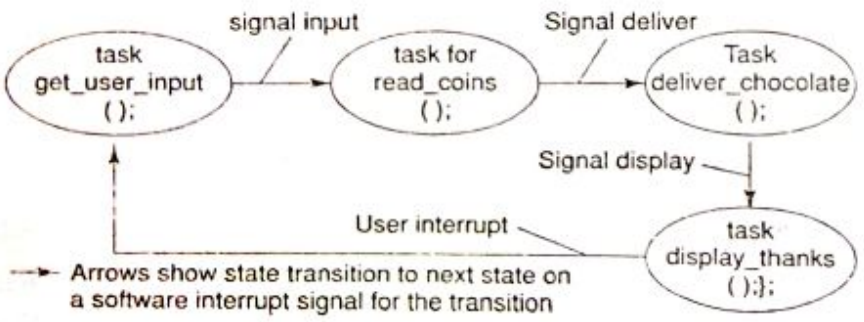


Fig. 8.10 FSM states in a program model for ACVM

8.3.3 FSM State Tables

When an FSM model is represented graphically with circles and directed arcs, it becomes complex in the case of a complex process with large number of states. A *state table* can then be designed for representation of every state in its rows to design software using the model. The following columns are made for each row.

Present State name or identification
Action(s) at the state until some event(s)
The events (tokens) that cause the execution of state transition function
Output(s) from the state output function(s)
Next State
Expected Time Interval for finishing the transitions to a new state after the event.

The coding using each row can now be easily done as follows.

```
while (presentState) {action ( ); if (event = .....; token = .... )
{output = ......; stateTransitionFunction ( ); };}
or
Switch (State)
Case presentState: action ( ); if (event = .....; token = .... )
{output = ......; stateTransitionFunction ( ); };}'
```

Here, *presentState* is a Boolean, which is true as long as the present state continues and turns false on transition to the next. The ***action*** ( ) is a function that executes at the state. If certain events occur and tokens are received (for example, clock input in a timer), a state transition function, ***stateTransitionFunction***, is executed which also makes *presentState* = false and transition occurs to the next state by setting *nextState* (a Boolean variable) = true.

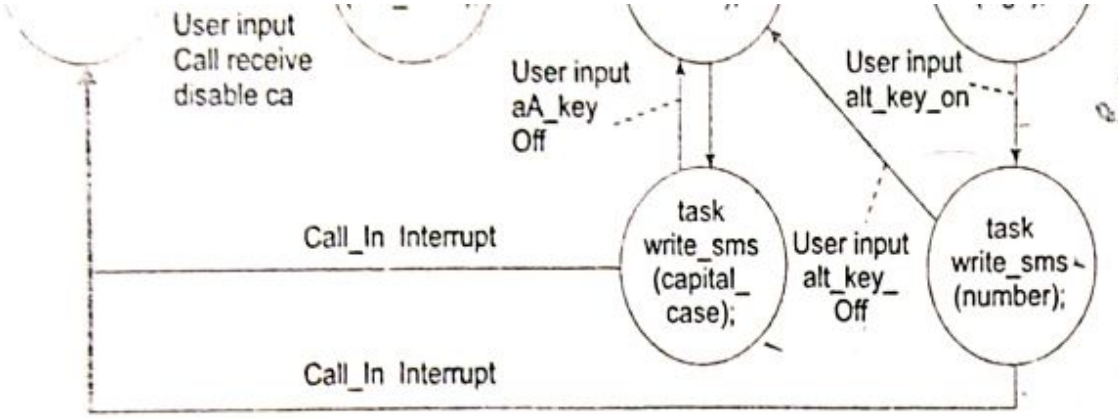
Example 8.10

Figure 8.11 shows the states, state transitions, events, outputs from state output function and finite number of state transitions of a mobile phone key marked as w. Total number of states are finite in number. State variables of state *S* are state_phone, state_sms, aA_key, alt_key, key_w, sign_key. Three keys are aA, alt and sign, and only one is active at an instant. .

State_phone active means the mobile is in dial a phone mode. State_sms active means the mobile is in sms keying-in mode.

When state_phone is active, state_sms is inactive, aA_key = inactive, alt_key = inactive, key_w = inactive, sign_key = inactive then key_w output state = 0 (idle). When key_w interrupt event activates then key_w output state undergoes transition to '1'. It means when S (state_phone, state_sms, aA_key, alt_key, key_w, sign_key) undergoes transition from initial state S(1, 0, 0, 0, 0, 0) to state S(1, 0, 0, 0, 1, 0) then state transition function generates key_w_output = '1'. ['1' means character 1. 1 means active. '0' means character 0 and 0 means inactive.]

When S (0,1, 0, 0, 0, 0) then on interrupt from key_w the next state is S(0,1,0,1, 0) and output key_w_output = 'w'.



→ Arrows show state transition to next state on a software interrupt (signal for the transition) or hardware interrupt

Fig. 8.11 State S (state_phone, state_sms, aA_key, alt_key, key_w, sign_key) undergoing state transitions and finite number of state transitions for S in mobile phone alphanumeric Qwerty keypad

Example 8.11

Make a state table for the FSM in Example 8.10. Table 8.1 gives the state table for the key 'w' in alphanumeric keypad of a mobile.

Table 8.1 State table for the key 'w' State S (state_phone, state_sms, aA_key, alt_key, key_w, sign_key) in alphanumeric keypad

Present State S	Action	Event key_w interrupt flag kwI	Next State S	Output key_w_output	Output kwI
S (1,0,0,0,0, 0)	wait	0	S (1,0,0,0,0, 0)	0	0
S (1,0,0,0,0, 0)	wait	1	S (1,0,0,0,1, 0)	'1'	0
S (0,1,0,0,0, 0)	wait	0	S (0,1,0,0,0, 0)	0	0
S (0,1,0,0,0, 0)	wait	1	S (0,1,0,0,1, 0)	'w'	0
S (0,1,1,0,0, 0)	wait	0	S (0,1,0,0,0, 0)	0	0
S (0,1,1,0,0, 0)	wait	1	S (0,1,0,0,1, 0)	'W'	0
S (0,1,1,0,0, 0)	wait	0	S (0,1,1,0,0, 0)	0	0
S (0,1,1,0,0, 0)	wait	1	S (0,1,1,0,1, 0)	'w'	0
S (0,1,0,1,0, 0)	wait	0	S (0,1,0,0,0, 0)	0	0
S (0,1,0,1,0, 0)	wait	1	S (0,1,0,0,1, 0)	'1'	0

```
# define true 1
# define false 0
# define state0 "000000"
# define state1 "100000"
# define state3 "010000"
# define state5 "011000"
# define state7 "010100"
# define state2 "100010"
# define state4 "010010"
# define state6 "011010"
# define state8 "010110"
void Key_w_FSM ( ) {
boolean idle_state
char key_w_output;
key_w_output = idlestate;
kwl =0;
State = State0;
while (true) { /*  An infinite loop */
/*-------------------------------------------------------------*/
/* function display ("x") shows character x on the screen, display (0) shows idle state which means same
as before and function cursor_next ( ) moves the cursor position to the next when keying in a phone
number or SMS text message. */
Switch (State) {*************************************************/
State0: if (kwl == 0) { idleState = 1; display (0 ); /* No change*/}
break;
/*************************************************************/
State1: if (kwl == 0) { idleState = 1; display (0 ); /* No change*/};
if (kwl == 1) { idleState = 0; State = State 2; key_w_output = '1'; display ('1' ); /* display character 1*/}
cursor_next ( ); kwl == 0; idleState = 1;
break;
/*************************************************************/
State3: if (kwl == 0) { idleState = 1; display (0 ); /* No change*/};
if (kwl == 1 && kw_state =0) { idleState = 0;State = State 4; key_w_output = 'w';State display ('w' ); /*
display character w*/};
cursor_next ( ); kwl == 0; idleState = 1;
break;
/*************************************************************/
State5: if (kwl == 0) { idleState = 1; display (0 ); /* No change*/};
if (kwl == 1 && kw_state ='w') { State = State 6; key_w_output = 'W'; display ('W' ); /* display character W*/};
if (kwl == 1 && kw_state ='W') { State = State 4; key_w_output = 'w'; display ('w' ); /* display character w */};
cursor_next ( ); kwl == 0;
break;
/*************************************************************/
State7: if (kwl == 0) { idleState = 1; display (0 ); /* No change*/};
if (kwl == 1) { State = State 8; display ('1' ); /* display character 1*/};
cursor_next ( ); kwl == 0; idleState = 1;
break;
/*************************************************************/
}
/*-----------------------------End of Switch -case ----------------------*/
State = State0;} /* End of While infinite loop */
} /* End of Key_w_FSM */
```

A finite state machine model assumes the finite number of states and reduces the programming tasks to the following. (i) Coding for each state transition function and each output function. The FSM model is appropriate for one process at a time, for the sequential flows from one state to the next state, and for controlled flow of the program. When using an FSM model, a state table representation becomes very handy while coding.

8.4 MODELING OF MULTIPROCESSOR SYSTEMS

8.4.1 Multiprocessor Systems

A large complex program can be partitioned into tasks (or processes or threads), ISRs and sets of instructions. The tasks and ISRs can run concurrently on different processors and by an appropriate mechanism. Tasks can communicate with each other.

Example 8.13

(a) Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has 4 ISRs, ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor PA is statically scheduled to run task 2, task 4, ISR_B, and ISR_D. Processor PB is statically scheduled to run task1, task 3, ISR_A, and ISR_C. Figure 8.12(a) shows the scheduling on two processors.

(b) Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has 4 ISRs, ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor has a dual core with one core is statically scheduled to run the tasks and other the ISRs. ISRs sends the messages to the tasks running on other core. Figure 8.12(b) shows the scheduling on a dual-core processor.

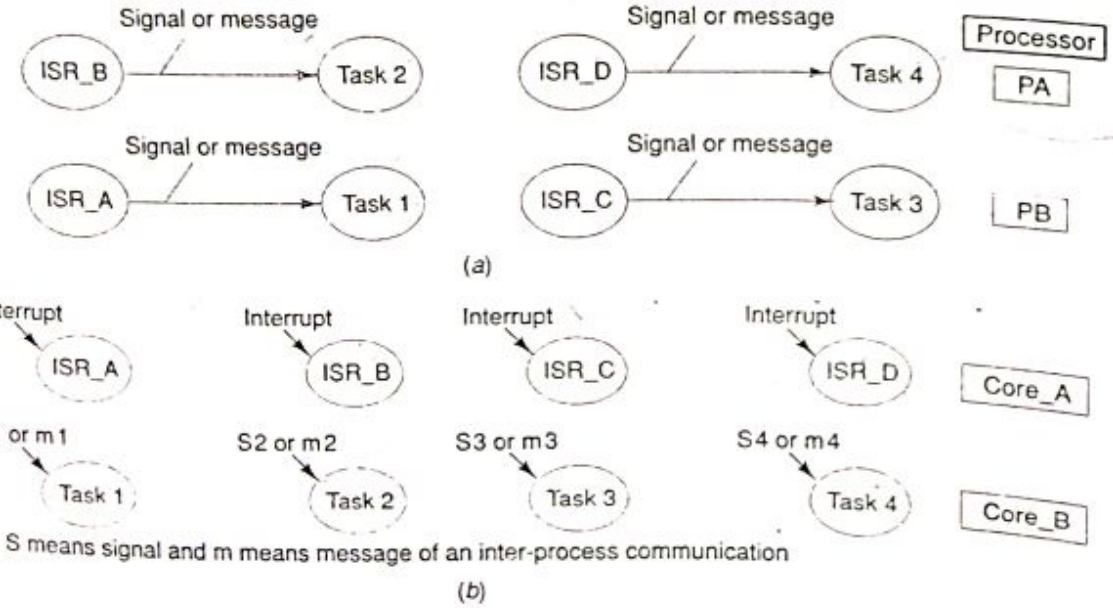


Fig. 8.12 (a) Static scheduling of tasks and ISRs on two processors (b) Static scheduling on two processor cores

The problem is how to partition the program into tasks or sets of instructions between the various processors, and then how to schedule the instructions and data over the available processor times and resources so that there is optimum performance. Should there be static scheduling for running one task on one processor? Then, suppose one processor finishes computations earlier than the other. What is the performance cost? Performance cost is more if there is idle time left from the available. What is the performance cost if one task needs to send a message to another and the other waits (blocks) till the message is received? Following are the problems in modeling the processing of instructions in a multiprocessor system:

1. Partitioning of processes, instruction sets and instruction(s).
2. Concurrent processing of *processes* on each processor.
3. The static scheduling by compiler, analogous to scheduling in a superscalar processor. Each superscalar processor has multiple processing units in parallel.
4. When superscalar units are present in a processor, it means two or more pipelines of instructions are executed in parallel. A pipeline has number of stages (3 to 9) and different instructions are at different stages. The problem then, is not only scheduling of concurrent processing instructions on different processors, but also scheduling of concurrent processing instructions on each superscalar unit and pipeline in the processor.
5. Hardware scheduling, for example, whether static scheduling of hardware (processors and memories) is feasible or not. [It is simpler and its use depends on the types of instructions when it does not affect the system performance.]
6. Static scheduling issue [for example, when the performance is not affected and when the processing actions are predictable and synchronous.]
7. Synchronising issues; synchronisation means use of interprocessor or process communications (IPCs) such that there is a definite order (precedence) in which the computations are fired on any processor in multiprocessor system. [IPC is a message or signal to another process or processor so that it can proceed further. Section 9.7 will describe the IPC in detail.]
8. Dynamic scheduling issues [for example, when the performance is affected when there are interrupts and when the services to the tasks are asynchronous. It is also relevant when there is pre-emptive scheduling as that is also asynchronous.]
9. Scheduling of the instructions, SIMDs (single instruction multiple data), MIMDs (multiple instructions and multiple data) and VLIWs (very large instruction words within each process and scheduling them for each processor.

Several methods of scheduling and synchronising can be used for execution of the instructions, SIMDs, MIMDs, and VLIWs in a multiprocessor system. Scheduling is done after analysing the scheduling and synchronising options.

Consider two processors, *PA* and *PB*, interfaced with the memory in a system.

- **Case 1** The processors share the same address space through a common bus, called tight coupling between processors.
- **Case 2** The processors have different autonomous address spaces (like in a network) as well as shared data sets and arrays, called loose coupling. Figures 8.13(a) and (b) show both the cases.
- **Case 3** The processors share the memories in alternative bus architecture, for example, three-dimensional mesh, ring, torrid or tree in place of a shared bus between the different tightly coupled processors.

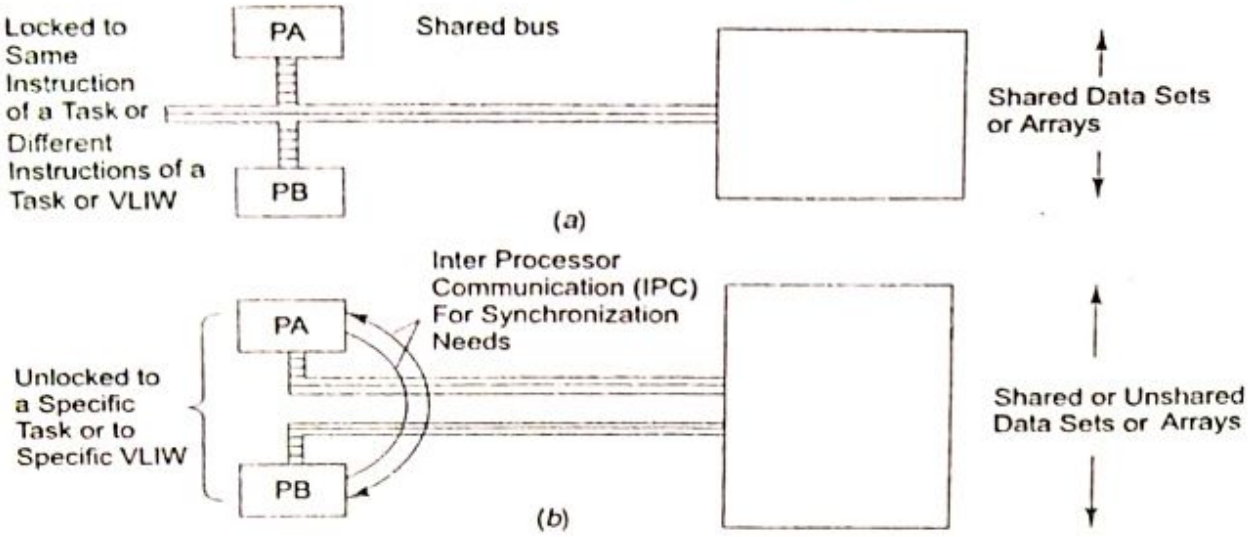


Fig. 8.13 (a) Tightly coupled processors sharing the same address space while processing multiple tasks (b) Loosely coupled processors having separate autonomous address spaces as in a network as well as shared address space for data sets and arrays

Processors process concurrently as follows:

1. One way of concurrent processing is to schedule each task so that it is executed on different processors and to synchronise the tasks by some interprocessor communication mechanism.
2. The second way is, when an SMID, MIMD or VLIW instruction has different data, each task is processed on different processors (tightly coupled processing) for different data. This is analogous to the execution of a VLIW in TMS320C6, a recent Texas Instruments DSP series processor. It employs two identical sets of four units and a VLIW instruction word can be within 4 and 32 bytes. It has instruction-level parallelism when a compiler schedules such that, the processors run the different instruction elements into the different units in parallel.

Note: The compiler does *static scheduling for VLIW's*. Static scheduling is one in which a compiler compiles such that the codes are run on different processors or processing units as per the schedule decided, and this schedule remains static during the program run even if a processor waits for the others to finish the scheduled processing.

3. An alternate way is that a task instruction is executed on the same processor, or different instructions of a task can be done on different processors (loosely coupled). A compiler schedules the various instructions of the tasks among the processors at an instance.

8.4.2 Applications of the Graphs to Multiprocessor Systems: Partitioning and Scheduling

When there are multiple processors in parallel, the partitioning of a program is done as follows:

1. There is a minimum number of IPCs so that the total time of IPC delays (waiting periods) minimises.
2. There is load balancing. Each processor has the least waiting time by sharing the processing load.
3. The performance cost minimises. *Performance cost means the execution time required (a) for computations for the tokens and delays at the edge (communication time), (b) the computation time before firing (computations) by a vertex (transition), and (c) context switch time.*

The graph of a program thus partitions into the functions, tasks or threads. One of three follo... strategies can schedule a program for running.

1. (a) Each task or function is executed on an assigned processor. (b) Each task or function is executed on different processors at different periods. (c) Instructions of four different tasks partitioned on two processors. (d) Instructions of four different tasks partitioned and scheduled on two processors differently in different periods. [Figures 8.14(a) to 8.14(d) show these four partitioning and scheduling strategies.]

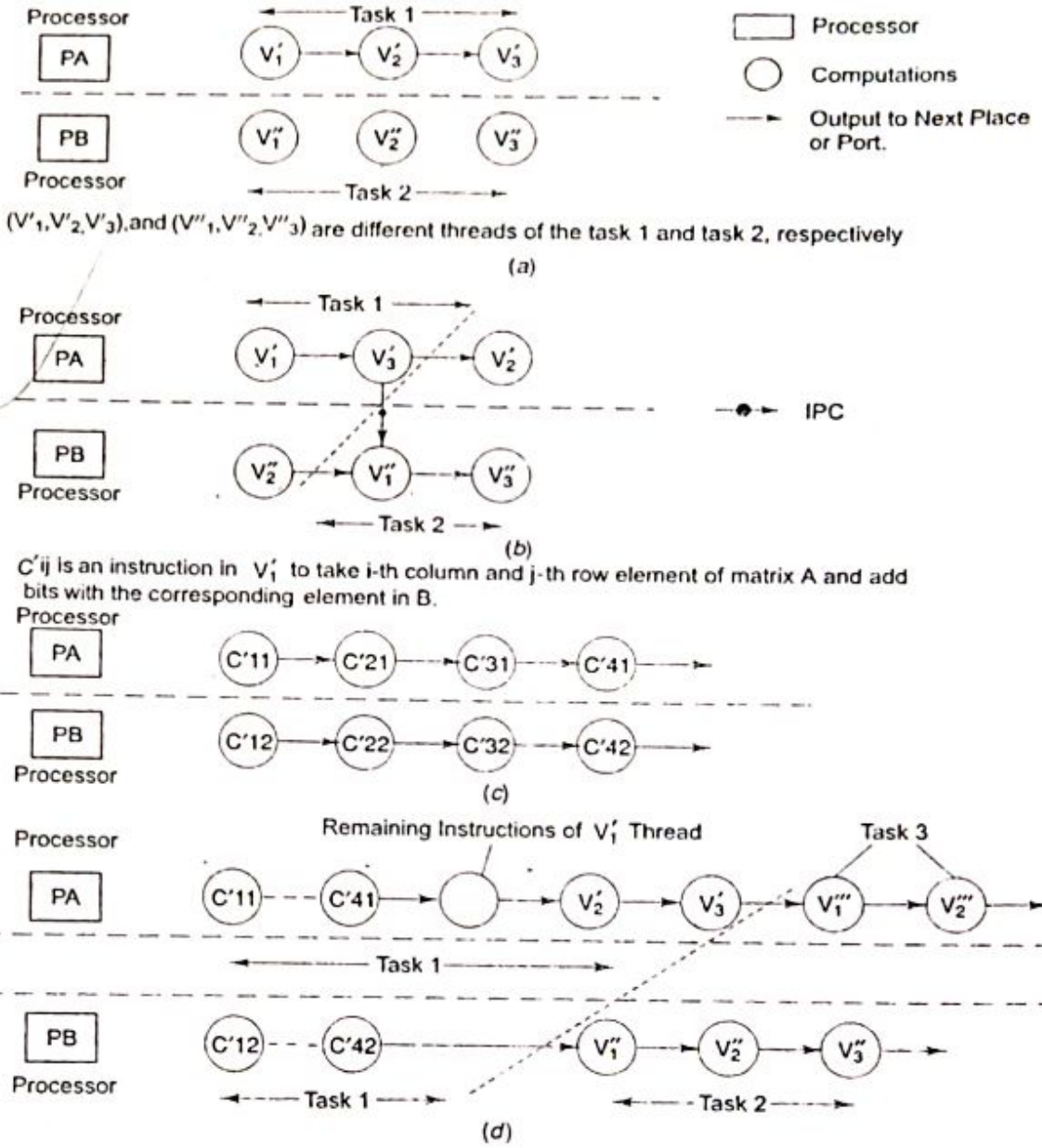


Fig. 8.14 (a) Each task or function is executed on an assigned processor (b) Each task or function is executed on different processors at different periods (c) Instructions of four different tasks partitioned on two processors (d) Instructions of four different tasks partitioned and scheduled on two processors differently in different periods

- Each set of data is partitioned in a VLIW instruction and is executed on different processors, which execute the same program. Consider a matrix addition process. Each row can be added on a different processor when the data of the rows are partitioned among the processors. Such data partitioning is preferred when processing a DSP-VLIW.

A combined partitioning is done both at data level as well as task (or function) level. Different functions themselves may run concurrently on different processors but at the micro or atomic level, data is partitioned and the instructions are run.

Partitioning and scheduling of vertices can be done in number of ways. (a) Each task or function is executed on an assigned processor. (b) Each task or function is executed on the different processors at different periods. (c) Instructions of four different tasks partitioned on two processors. (d) Instructions of four different tasks partitioned and scheduled on two processors differently in different periods. (e) Data partitioning in case of SIMDs, MIMDs and VLIWs.

8.5 ‖ UML MODELING

Concepts used in object-oriented language are also used in software designing.

1. Object-oriented design has feature of re-usability of the defined software components as object or set of objects (reusable components). New components can be abstracted from the existing. New components and object designs are created by the object inheritances and polymorphs. Information encapsulates within a designed component or object.
2. An object characterises by its identity (a reference to it that holds its state and behavior), by its state (its designs for data, property, fields, attributes and algorithms) and by its behavior (method or methods that can manipulate the state of the design).
3. New object designs are created as the instances of a class. Class defines the state, attributes, operations and behavior of a design concept. It has internal user-level fields for its state and behavior. It defines the ways of using the designs.
4. A class can then create many component objects (designs) by copying the group and making designs functional. Each design is a functional design. Each object design can interface with other designs to process the states as per the defined behavior.
5. A set of classes then gives the complete software design for a system.

UML is a Unified (common) Modeling Language for any general system for which object-oriented analysis and design are feasible and which can be abstracted by models. Unification in UML means its common applicability to many designs or processes. We can then model the following by a similar set of diagrams: (i) Software Visualising, (ii) Data Design(s), (iii) Algorithm Design(s), (iv) Software Design(s), (v) Software specifications, (vi) Software Development Process, and (vii) An Industrial Process.

UML is a language for modeling. Details of the language can be learnt from standard textbooks. Following is a description of UML features and its applications in designing of embedded systems.

Figures 8.15(a) to (f) show six basic UML elements: class, package, stereotype, object, anonymous object and state.

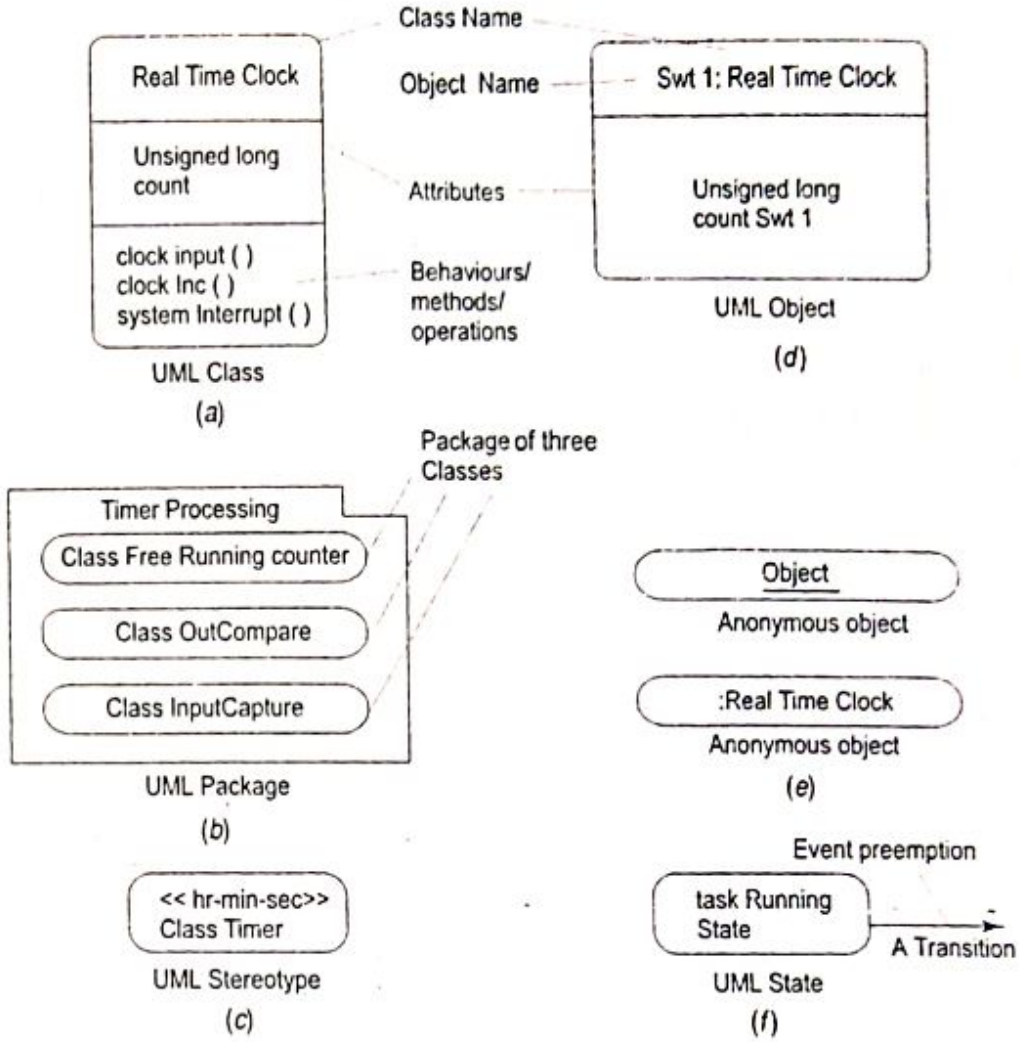


Fig. 8.15 Representation of UML basic elements: (a) Class (active class and abstract or inactive class) (b) Package (c) Stereotype (d) Object (e) Anonymous object (f) State

Table 8.2 gives its elements.

Table 8.2 UML basic elements

Modeling Diagram	What does it Model and Show?	Exemplary Diagrammatic Representation
Class	Class defines the states, attributes and behavior. A class can be active or abstract.	Rectangular box with divisions [Figure 8.15(a)] for class names, for its identity, attributes, and behaviors (operations, methods, routines, or functions)
Abstract Class	A class, in general, may be abstract when either one or more states, operations or behaviors not completely defined, being in an abstract stage, or when it is not for creating objects but only a class, which extends that class and implements the abstract behaviors (methods) and specifies the	Rectangular box with divisions for class names for its identity, attributes, and operations, but with prefix abstract with each abstract behavior and attribute

10.1.1 OS Services Goal

OS services Goal of perfection and correctness'. OS facilitates the following:

1. **Easy sharing of resources as per schedule and allocations**. Resources mean processor(s), memory, I/O, devices, pipes, sockets, system timer, keyboard, displays, printer and other such resources, which processes (tasks or threads) request from the OS. No processing task or thread uses any resource until it has been allocated by the OS at a given instance.
2. **Easy implementation** of application software with the given system hardware. An application uses the OS functions and processes which are provided in the OS.
3. **Scheduling**, context switching and interrupt-servicing mechanisms.
4. **Management of the processes**, tasks, threads, memory, IPCs, devices, and other functions [Management means creation, resources allocation, resources freeing, scheduling or synchronising and deletion.]
5. Files, I/O and Network **subsystems and protocols**.
6. **Portability** of the application on different hardware configurations.
7. **Interoperability** of the application on different networks.
8. **Common set of interfaces** that integrates various devices and applications through standard and open systems.
9. Easy use of the interfacing functions, **GUIs and APIs**.
10. Maximising the **system performance** to let different processes (or tasks or threads) share the resources most efficiently. OS provides the protection and security. Examples of security breach are tasks as follows: obtaining illegal access to other task data directly without system calls, overflow of stack areas into the memory, and overlaying of process and control blocks and thread stacks in memory.

10.1.2 User and Supervisory Mode Structure

When using an OS, the processor in the system runs in two modes. There is a clock, called system clock. At every clock tick of system-clock, there is an interrupt. On interrupt, the system time updates, the system context switches to supervisory mode from the user mode. After completing the supervisory functions in the OS, the system context switches back to user mode.

ample 10.1

RTOS permits running of the processes, tasks and threads in supervisory mode (kernel mode) and rotected kernel space. Therefore, the threads execute fast.

0.1.3 Structure

ble 10.1 gives the layers at the structure in the system.

Table 10.1 Layers in the System Structure

Layer from Top	Top-down Structure Layers	Actions
1	Application Software	Executes as per the applications run on the given system hardware using the interfaces and the system software
2	Application Programming Interface (API)	Provides the interface (for inputs and outputs) between application and system software so that it is able to run on the processor using the given system software.
3	System Software other than the one provided at the OS	OS may not have the functions, for example, for a specific network and for certain device drivers, such as a multimedia device. This layer gives the system software services other than those provided by the OS service functions.
4	OS Interface	Interface *(for inputs and outputs)* between the above layers and OS
5	OS	Kernel supervisory mode services, file management and other functions for the OS services
6	Hardware – OS Interface	Interfaces to let the functions execute on the given hardware (processor, memory, ports and devices).
7	Hardware	Processor(s), memories, buses, interfacing circuits, ports, physical devices, timers, and buses for devices networking.

0.1.4 Kernel

An OS is the middle layer between the application and system hardware. An OS includes some or all of the following structural units.

1. Kernel with file management and device management as part of the kernel in the given OS
2. Kernel without file management and device management as part of the kernel in the given OS and any other needed functions not provided for at the kernel

The kernel is the basic structural unit of any OS. Memory space of the kernel functions, data, and stack is protected from access by any function call other than the system call. It can be defined as a secured unit of the OS that operates in supervisory mode. The remaining part and application runs in user mode.

The following are the functions (services) of the kernel.

(a) Process management
(b) Memory management
(c) File management
(d) Device management and device drivers
(e) I/O subsystem management

When considering the processes controlled by an OS, a process also means task in multitasking OS, and thread in multithreading OS. Memory, file and device-management functions may be the part of kernel in a given OS. However, these functions may be outside the kernel in a given OS, especially in embedded systems using the microkernel of RTOS. This makes the kernel code small.

10.2 PROCESS MANAGEMENT

Table 10.2 lists the process-management functions of the OS kernel. Process management also means task and thread management.

Table 10.2 Process-management functions of the OS kernel

Process Management Function	Actions
Creation to Deletion	Enables process creation, activation, running, blocking, resumption, deactivation and deletion and create process structure at a PCB (Process Control Block).
Process Structure Maintenance	Enables process structure maintenance and its information at PCB (Process Control Block).
Processing Resource Requests	Processing resource requests by processes made either by making calls which are known as system calls or by sending the message(s)
Scheduling	Processes *scheduling*, for example, cyclic scheduling or priority scheduling mode.
Inter Process Communication (IPC)	Processes synchronising by sending data as messages from one task to another. An OS effectively manages shared memory accesses by using the IPCs such as signals, exception (error) handling signals, semaphores, queues, mailboxes, pipes, sockets and RPCs. IPC enables communication between the tasks, ISRs, ISTs and OS functions.

10.3 TIMER FUNCTIONS

A real-time clock (hardware timer timeout) interrupts the system timer with a tick at regular interval. System timer intervals are predefined by the number of RTC ticks which will cause system timer interrupt. An interrupt on a tick can be denoted by *SysClkIntr* (system real-time clock timer interrupt). The number of system ticks per second, which means number of SysClkIntr are 60 or can be set by a system timer function. High-resolution system-timer intervals can be defined by an OS function. Each RTOS has a function for defining OS ticks per second, which defines the period of generation of the SysClkIntr interrupts and which in turn enables use of number timer functions of OS.

Example 10.2

1. (a) `# define OS_TICK_PER_SEC 100 /*µCOS-II function to define the number of ticks per second = 100 before the beginning of the main ( ) and the initiating the OS by OSInit ( ) function*/.`
   (b) `OSTickInit ( ) /*µCOS-II function to initiate the defined number of ticks per second after the beginning of the first task and creating all the tasks to which the context will be switched by the OS on the tick. It initiates SysClkIntr interrupts every 10 ms */.`
2. A Windows function **ExSetTimerResolution** enables definition of timer resolution for SysClkIntr intervals. Default the timers cause SysClkIntr interrupts 60 times per second. Windows 8.1 provides for ExXxxTimer high-resolution timer functions which timers and drivers can use.

10.6.3 I/O Subsystems

The I/O ports are the subsystems of OS device-management systems. Drivers use them to communicate with the many devices. The I/O instructions depend on the hardware platform. I/O systems differ in the different OS. Table 10.4 lists subsystems of a typical I/O system.

Table 10.3 I/O Subsystem in a typical I/O system in an OS

Subsystem Elements	Activities and layers between the subsystems
Application	An application having an I/O system. A sublayer may exist between the application and I/O basic functions.
I/O Basic Functions	Device-independent OS functions, for example, file-system functions for read and write, buffered I/O or file (block) read and write functions. A sublayer may also exist between basic I/O functions and I/O device driver functions.
I/O Device-Driver Functions	Device-dependent OS functions. A driver may interface with a set of library functions. For example, for serial communication device or network.
Device Hardware	Network or port or I/O interface card.

Synchronous and Asynchronous I/Os

Two types of I/O operations, synchronous and asynchronous, may be used. RTOS provisions for separate functions for the synchronous and asynchronous I/Os. A traditional OS may support only synchronous I/Os.

Synchronous I/O operations are at a certain fixed-data transfer rates. Therefore, a task (process) blocks until the completion of the I/O. For example, a write function, *write* ( ) for 1 KB data transfer to a buffer. Synchronous I/O operation means once synchronous I/O initiates, the data transfer will block the task until 1 KB data gets transferred to the buffer. Similarly, *read* ( ) once initiated, blocks the task till 1 KB is read.

Asynchronous I/O operations are at the variable data-transfer rates, because a process of high priority should not block during the I/Os.

10.7 INTERRUPT ROUTINES IN RTOS ENVIRONMENT AND HANDLING OF INTERRUPT-SOURCE CALLS

An RTOS or OS has an interrupt-servicing mechanism. Following sections explain these alternatives for responding to interrupt.

10.7.1 Direct Call to an ISR by an Interrupting Source

When an interrupt occurs, the process running at the CPU interrupts. Context switching takes place directly to the ISR. An i^{th} interrupt source leads to switch to i^{th} ISR. ISR_i just sends an ISR enter message at start for the RTOS. Figure 10.1 shows these steps.

Later the ISR code can post an IPC (mailbox or message queue, or semaphore). Task waiting for that IPC does not start before the return from the ISR. The ISR enter message is to inform the RTOS that an ISR has taken control of the CPU. The ISR continues execution of the codes needed for the interrupt service until the ISR exit message.

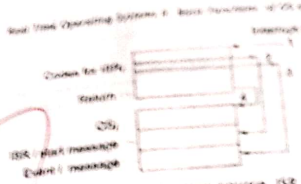


Fig. 10.1 Direct Call to an ISR by an interrupting source. ISR sends an ISR enter message to OS and ISR exit message before return to OS

Example 10.10

Consider the RTOS µCOS-II. Assume that a microcontroller has a hardware timer, which is programmed to interrupt every 10 ms, the microcontroller on timer interrupt calls and program counter changes to an ISR vector address, ISR_Timer_Addr. At ISR_Timer_Addr, there is a routine ISR_Timer for the servicing the timer interrupt.

ISR_Timer first executes OSIntEnter ( ) just after the start of ISR_Timer is called. ISR_Timer then executes OSIntExit ( ) before the return code.

The OSIntEnter ( ) sends the message to RTOS that there should be return to the ISR only after an system call, is made by the ISR until the OSIntExit ( ) executes in the ISR code. Any task waiting for the post of semaphore or mailbox message or queue message should not start on execution of the pos function within the ISR or in any other task or ISR.

10.7.2 RTOS First Interrupting on an Interrupt, then RTOS Calling the Corresponding ISR

When a task is interrupted on i-th interrupt source then context first switches to RTOS. The RTOS like servicing the hardware interrupt, switches the context to corresponding ISR_i. The ISR during execution then can post one or more IPCs for the task(s) waiting for the mailbox, queue or semaphore message. Context switches back to RTOS on return from ISR_i. Context then switches from RTOS to pending interrupt ISR or highest priority task. Figure 10.2 shows the steps.

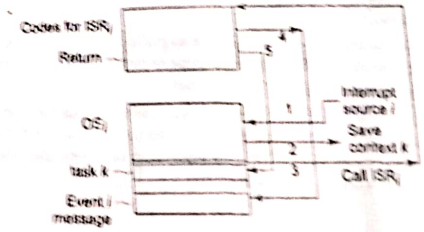


Fig. 10.2 RTOS first interrupting on an interrupt, then RTOS calling the corresponding ISR

10.6.3 I/O Subsystems

The I/O ports are the subsystems of OS device-management systems. Drivers use them to communicate with the many devices. The I/O instructions depend on the hardware platform. I/O systems differ in the different OS. Table 10.9 lists subsystems of a typical I/O system.

Table 10.9 I/O Subsystem in a typical I/O system in an OS

Subsystems Hierarchy	Action(s) and layers between the subsystems
Application	An application having an I/O system. A sublayer may exist between the application and I/O basic functions.
I/O Basic Functions	Device-independent OS functions, for example, file-system functions for read and write, buffered I/O or file (block) read and write functions. A sublayer may also exist between basic I/O functions and I/O device driver functions.
I/O Device-Driver Functions	Device-dependent OS functions. A driver may interface with a set of library functions. For example, for serial communication device or network.
Device Hardware	Network or port or I/O interface card.

Synchronous and Asynchronous I/Os

Two types of I/O operations, synchronous and asynchronous, may be used. RTOS provisions for separate functions for the synchronous and asynchronous I/Os. A traditional OS may support only synchronous I/Os.

Synchronous I/O operations are at a certain fixed-data transfer rates. Therefore, a task (process) blocks until the completion of the I/O. For example, a write function, *write* ( ) for 1 kB data transfer to a buffer. Synchronous I/O operation means once synchronous I/O initiates, the data transfer will block the task until 1 kB data gets transferred to the buffer. Similarly, *read* ( ) once initiated, blocks the task till 1 kB is read.

Asynchronous I/O operations are at the variable data-transfer rates, because a process of high priority should not block during the I/Os.

10.7 INTERRUPT ROUTINES IN RTOS ENVIRONMENT AND HANDLING OF INTERRUPT-SOURCE CALLS

An RTOS or OS has an interrupt-servicing mechanism. Following sections explain these alternatives for responding to interrupt.

10.7.1 Direct Call to an ISR by an Interrupting Source

When an interrupt occurs, the process running at the CPU interrupts. Context switching takes place directly to the ISR. An i^{th} interrupt source leads to switch to i^{th} ISR. ISR_i just sends an ISR enter message at start for the RTOS. Figure 10.1 shows these steps.

Later the ISR code can post an IPC (mailbox or message queue, or semaphore). Task waiting for that IPC does not start before the return from the ISR. The ISR enter message is to inform the RTOS that an ISR has taken control of the CPU. The ISR continues execution of the codes needed for the interrupt service until the ISR exit message.

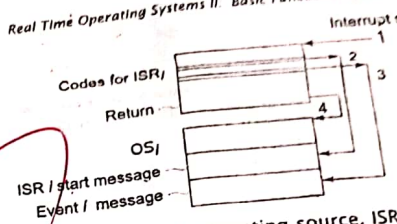


Fig. 10.1 Direct Call to an ISR by an interrupting source. ISR sends an ISR enter message to OS and ISR exit message before return to OS

Example 10.10

Consider the RTOS µCOS-II. Assume that a microcontroller has a hardware timer, which is programmed to interrupt every 10 ms. The microcontroller on timer interrupt calls and program counter changes to an ISR vector address, ISR_Timer_Addr. At ISR_Timer_Addr, there is a routine ISR_Timer for the servicing the timer interrupt.

ISR_Timer first executes OSIntEnter ( ) just after the start of ISR_Timer is called. ISR_Timer then executes OSIntExit ( ) before the return code.

The OSIntEnter ( ) sends the message to RTOS that there should be return to the ISR only after any system call is made by the ISR until the OSIntExit ( ) executes in the ISR code. Any task waiting for the post of semaphore or mailbox message or queue message should not start on execution of the post function within the ISR or in any other task or ISR.

10.7.2 RTOS First Interrupting on an Interrupt, then RTOS Calling the Corresponding ISR

When a task is interrupted on i-th interrupt source then context first switches to RTOS. The RTOS for servicing the hardware interrupt, switches the context to corresponding ISR_i. The ISR during execution then can post one or more IPCs for the task(s) waiting for the mailbox, queue or semaphore message. Context switches back to RTOS on return from ISR_i. Context then switches from RTOS to pending interrupt ISR or highest priority task. Figure 10.2 shows the steps.

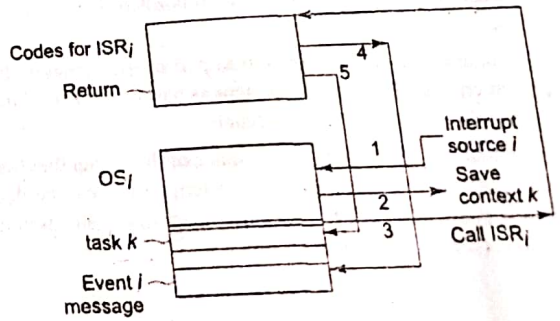


Fig. 10.2 RTOS first interrupting on an interrupt, then RTOS calling the corresponding ISR

The ISR must be short and it must simply post the messages for another task. This task runs the remaining codes whenever it is scheduled. The RTOS schedules only the tasks (processes) and switches the contexts between the tasks only. The ISR executes only during a temporary suspension of a task.

An RTOS may provide for the ISRs such that the RTOS initiates running of the ISR calls from a priority ordered FIFO.

Example 10.11

1. There is a routine *i*-th ISR and two processes in three memory blocks other than the interrupted *k*-th task. Then the RTOS to get the notice of that, *k*-th task finishes the critical code till the preemption point. The RTOS then calls the *i*-th ISR. Preemption points mean instruction where context switch is permitted by higher priority ISR or process. The preemption point is the last instruction of the critical part of the presently running function, after which the ISR being of highest priority is called.
2. Consider mobile device. Assume that using an RTOS, the touch screen ISR, ISR_TouchScreen has been created using a function OS_ISR_Create ( ). An ISR can share the memory heap with other ISRs. Heap means the data output generated during running of the codes. A function, IntConnect connects the touch-screen event with the event identifier in an interrupt handler, ISR_TouchScreenEventHandler. Let a touch-screen event occur, which means user of mobile taps the screen at a select icon or menu. The OS first lets presently running task run up to preemption point, then first the context switches to RTOS. RTOS then switches the context to ISR_TouchScreenEventHandler. ISR return instruction causes context switch back to RTOS. RTOS finds which ISR or task should run next and context then switches to that.

10.7.3 RTOS First Interrupts, Calls to corresponding ISR, then ISR sending messages to interrupt Service Threads

An RTOS can provide for two levels of interrupt-service routines, a first-level ISR (FLISR) and a slow-level ISR (SLISR). The FLISR can also be called hardware-interrupt ISR and the SLISR as software-interrupt ISR.

FLISR is called just the ISR in RTOS in Windows. SLISR is called Interrupt Service Thread (IST) in Windows.

The use of FLISR reduces the interrupt latency (waiting period) for an interrupt service and jitter (worst case and best case latencies difference) for an interrupt service. An IST function is a deferred procedure call (DPC) from the ISR. The *i*-th interrupt service thread (ISTi) is a thread to service an *i*-th interrupt source call. Figure 10.1(c) shows the steps on the interrupt.

There are the ISRs, number of ISTs, RTOS and tasks in the memory blocks other than the interrupted task. Any interrupt source causes the RTOS to get the notice of that, then finish the critical code till the preemption point and call the ISR. The ISR executes after saving the context on to a stack. The ISR can post a message into the FIFO for the interrupt service thread (IST) after recognising the interrupt source and its priority. The ISTs in the FIFO that have received the messages from the ISR execute as per their priorities on return from the ISR. The ISR has the highest priority and preempts all pending ISTs and tasks.

When no ISR of IST is pending execution in the FIFO, the interrupted task runs.

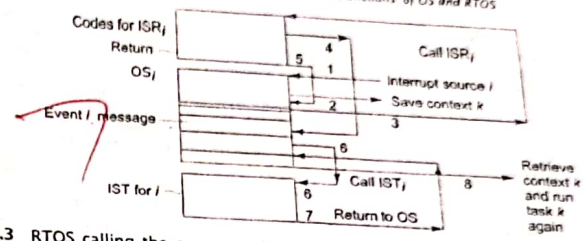


Fig. 10.3 RTOS calling the corresponding ISR, the ISR sending message(s) to an interrupt-service thread in a priority queue of ISTs

The ISRs must be short, run critical and necessary codes only, and then they must simply send the initiate call or messages to ISTs into the FIFO. It is the IST, which runs the remaining codes as per priority-based schedule. The ISTs run in the kernel space. The ISTs does not lead to priority inversion and have the priority-inheritance mechanism.

RTOS schedules the ISTs and tasks (processes), and switches the contexts between the ISTs and tasks.

Example 10.12

Consider Mac OS X. The Mac OS X is the RTOS for mobile Apple devices. An interrupt handler first receives the primary interrupt and then it then generates a software interrupt known as a secondary interrupt. The secondary software interrupt is sent to an interrupt-service thread.

The OS does not receive the actual interrupt but the low-level ones that intercept the interrupt. It calls a low-level (hardware level) ISR, *LISR*, resets the pending interrupt bit in the device interrupt controller and calls a device-specific ISR, say, DISR$_i$. The DISR$_i$ posts a message to an IST$_i$ specific to the device. The message notifies to IST$_i$ that an interrupt has occurred, and then the DISR$_i$ returns to LISR. The LISR resets another pending interrupt bit in the device interrupt controller and calls the another device-specific ISR, say, DISR$_j$.

When no further interrupts are pending, the OS control returns to the currently executing thread, which was interrupted when the OS passed control to the LISR.

The IST$_i$ are scheduled by the OS, the IST$_i$ finds that the interrupt has occurred, it starts and run the codes. ISTs run as if a thread is running.

10.7.4 Accepting an IPC Event by ISR

RTOS servicing mechanism provisions for following:

(a) ISRs have the higher priorities over tasks and most RTOS functions.

(b) An ISR should not wait for a semaphore, mailbox message or queue message. An ISR can use only the accept function for the events. An ISR should not also wait for mutex, else it has to wait for other task or ISR section code to finish before the ISR can run.

10.8 INTRODUCTION TO REAL-TIME OPERATING SYSTEMS

A Real-Time Operating System (RTOS) is multitasking operation system for applications, which require that system tasks and functions execute with real-time constraints. Real-time constraint means

Example 10.16

1. A program can be such that it reduces the brightness level of the LCD panel so that it takes less power when the system is used in fully lighted room. [A sensor senses the light level at specific intervals.]
2. An embedded system may need to run continuously, without being switched off; the system design, therefore, is constrained by the need to limit power dissipation while it is running. Total power consumption by the system in running, waiting and idle states should also be limited. A program can provide for auto switch-over of standby mode in case of the system not used within a specified time interval and stop mode when the system not used for long intervals.

■ **Disable Caches Mode** Yet another method is to disable use of certain structural units of the processor (for example, caches) when not necessary, and to keep in disconnected state those structure units that are not needed during a particular software-portion execution (for example, timers or I/O units). The software designer should enable the use of caches in a processor by an appropriate instruction, to obtain greater performance during the run of a section of a program, while simultaneously disabling the remaining sections in order to reduce the power dissipation and minimise the system-energy requirement. Hardware designers should select a processor with multi-way cache units so that only that part of a cache unit gets activated that has the data necessary to execute a subset of instructions. This also reduces power dissipation.

10.10 ‖ RTOS TASK-SCHEDULING MODELS

Following are the common scheduling models used by schedulers.

1. Cooperative scheduling of ready tasks in a circular queue. It closely relates to function queue scheduling.
2. Cooperative scheduling with precedence constraints
3. Cyclic and round-robin time-slicing scheduling
4. Preemptive scheduling
5. Scheduling using 'Earliest Deadline First' (EDF) precedence
6. Rate monotonic scheduling using 'higher rate of events occurrence first' precedence
7. Fixed-times scheduling
8. Scheduling of periodic, sporadic and aperiodic tasks
9. Advanced scheduling algorithms for multiprocessors and for complex distributed systems

An RTOS commonly executes the codes for the multiple tasks as priority based preemptive scheduler.

10.10.1 Model for Preemptive Scheduling

Preemptive scheduling means higher priority task and very high-priority ISRs preempt running of a lower priority task whenever the higher priority task is ready to run after receiving the pending interprocessor communication, such as semaphore or message.

1. The preemption event takes place when an interrupt occurs, and just before the return from the interrupt, there is a service call to the RTOS by the ISR. On this call to the RTOS, a token, the *preemptionEvent*, is set. The context then switches to ISR.
2. Each RTOS uses a system clock ticked by a *SysClkIntr* interrupt. The preemption event takes place when the *SysClkIntr* interrupt (real-time, clock-driven, software-timer interrupt) occurs at the RTOS. On this event, RTOS takes control of the processor and checks whether it should let currently executing task continue or to preempt it to make way for the higher priority task. This event makes

another higher priority task ready to run, on the switch of the flag to the latter. Task then undergoes transition to the state, *readyTask from blocked task state* and runs after scheduling by scheduler.

3. The preemption event takes place when any call to the RTOS occurs to enter the critical section or for sending the task message (outputs) to the RTOS, and if another higher priority task then needs to be serviced (take control of the CPU). [Now the preemption is before entering the critical section.]

10.11 ‖ OS SECURITY ISSUES

When a doctor has to treat multiple patients, protection of the patients from any confusion in the medication becomes imperative. When an OS has to supervise multiple processes and their access to the resources, protection of memory and resources from any unauthorized writes into the PCB or resource or mix up of accesses of one by another becomes imperative. *OS security issue is a critical issue.*

Each process determines whether it has a control of a system resource exclusively or whether it is isolated from the other processes, or whether it shares a resource common to a set of processes. For example, a file or memory blocks of a file will have exclusive control over a process and a free memory space will have the access to all the processes. The OS then configures when a resource is isolated from one process and a resource is shared with a defined set of processes.

The OS should also have the flexibility to change the configuration when needed, to fulfill the requirement of all the processes. For example, a process has a control of 32 memory blocks at an instance and the OS configures the system accordingly. Later when more processes are created, this can be reconfigured.

The OS should provide a protection mechanism and implement a system administrator (s) defining security policy. For example, a system administrator can define the use of resources to the registered and authorized users (and hence their processes).

What about issues of an *application* changing the OS configuration? The OS needs a protection mechanism for itself. An application-software programmer can find a hole in the protection mechanism and gain an unauthorized access. Thus, the implementation of protection mechanisms and enforcement of security policy for resources is a challenging issue before any OS software designer. The network environment complicates this issue.

Table 10.12 lists the security functions.

Table 10.12 Important security functions

Fuhction	*Activities*
Controlled Resource Sharing	Controlling *read* and *write* of the resources and parameters by user processes. For example, some resources write only for a process and some read only for a set of processes.
Confinement Mechanism	Mechanism that restricts sharing of parameters to a set of processes only.
Security Policy (Strategy)	Rules for authorizing access to the OS, system and information. A policy example is a communication system having a policy of peer-to-peer communication (connection establishment preceding flow of data packets).
Authentication Mechanism	External authentication mechanism for the user and a mechanism to prevent an application run unless the user is registered and the system administrator (software) authorised. Internal authentication for the process, and the process should not appear (impersonate) like other processes. User authentication can become difficult if the user disseminates passwords or other authentication methods.
Authorization Mechanism	User or process(s) allowed using the system resources as per the security policy.
Encryption	A tool to change information to make it unusable by any other user or process without the appropriate key for deciphering it.

10.13 RTOS INTERRUPT LATENCY AND RESPONSE TIMES OF THE TASKS AS PERFORMANCE METRICS

10.13.1 Latency and Deadlines as Performance Metric in Scheduling Models for Periodic, Sporadic and Aperiodic Tasks

An RTOS should quickly and predictably respond to the event. It should have minimum interrupt latency and fast context-switching latency. Different models have been proposed for measuring performances. Three performance metrics are as follows:

(i) Ratio of the sum of Interrupt Latencies with respect to the sum of the execution times

(ii) CPU load

(iii) Worst-case execution time with respect to mean execution time

'Interrupt latencies' in various task models can be used for evaluating performance metrics. The CPU load is another way to look at the performance. Worst-case performance can be calculated for a sporadic task.

10.13.2 CPU Load as Performance Metric

Each task gives a load to the CPU that equals the task execution time divided by the task period. [Task period means period allocated for a task.] Consider In_AOut_B intra network Receiver port A expects another character before 172 µs, i.e. task period is 172 µs for 64 kbps data rate. If the task execution time is also 172 µs then the CPU load for this task is 1 (=100%). In this case, the task execution time when a character is received must be less than 172 µs as the maximum load of the CPU is 1 (less than 100%).

The CPU load or *system load* estimation in the case of multitasking is as follows: Suppose there are m tasks. For the multiple tasks, the *sum* of the CPU loads for all the tasks and ISRs should be less than 1. The timeouts and fixed time-limit definitions for the tasks reduce the CPU load for the higher priority tasks so that even the lower priority tasks can run before the deadlines. What does it mean when the sum of the CPU loads equal 0.1 (10%)? It means the CPU is underutilised and spends its 90% time in a waiting mode. Since the execution times and the task periods vary, the CPU loads can also vary.

When a task needs to run only once then it is *aperiodic* (one shot) in an application. Scheduling of the tasks that need to run periodically with the fixed periods can be periodic and can be done with a CPU load very close to 1. An example of a periodic task is as follows. There may be inputs at a port with predetermined periods, and the inputs are in succession without any time gap.

When a task cannot be scheduled at fixed periods, its schedule is called Sporadic. For example, if a task is expected to receive inputs at variable time gaps then the task schedule is sporadic. An example s the packets from the routers in a network. The variable time gaps must be within defined limits.

A preemptive scheduler must take into account three types of tasks (aperiodic, periodic and sporadic) eparately.

(i) An aperiodic task needs to be preempted only once.

i) A periodic task needs to be preempted after the fixed periods and it must be executed before its next preemption is needed.

) A sporadic task needs to be checked for preemption after a minimum time period of its occurrence.

Usually, the strategy employed by the software designer is to keep the CPU load between 0.7 ± 0.25 for sporadic tasks.

10.13.3 Sporadic Task Model Performance Metric

Let us consider the following parameters.

T_{total} = Total length of periods for which sporadic tasks occur

e = Total task execution time

T_{av} = Mean periods between the sporadic occurrences

T_{min} = Minimum period between the sporadic occurrences

Worst-case execution-time performance metric, p is calculated as follows for worst case of a task in a model.

$$p = p_{worst} = (e * T_{total} / T_{av}) / (e * T_{total} / T_{min}).$$

It is because the average rate of occurrence of sporadic task = (T_{total} / T_{av}) and maximum rate of sporadic task burst = T_{total} / T_{min}. time.

10.14 OS PERFORMANCE GUIDELINES

OS performance affects most by the following: memory management, interrupt handling and scheduling functions.

Memory is an important system resource that all programs use. Memory systems mean virtual, physical and cache memory. Initialising, allocating and copying of memory address spaces are the operations for the codes, data and heap (data generated while running). When memory resources become low or memory leaks are detected, the memory notifications are also issued. Tracking, analysing memory usage, caching, purging and finding memory leaks in the application are the operations performed by OS.

Each operation requires time and resources. The time requirement, therefore, affects the overall system performance. Performance guidelines in an OS are for efficient use of the memory in systems. Guideline document provides background information about the memory systems and how program uses them efficiently. *Efficiency* means right amount of memory is to be allocated at the right instance.

A scheduling algorithm selects the process at given instance, which executes in given scheduling scheme. Overall performance maximises the function of the OS scheduler.

Performance tuning means to optimise the real-time system performance (CPU and Memory usages). An understanding of the hardware, operating system and application is required.

10.15 MIDDLEWARE: MEANING AND EXAMPLES

Middleware means the following:

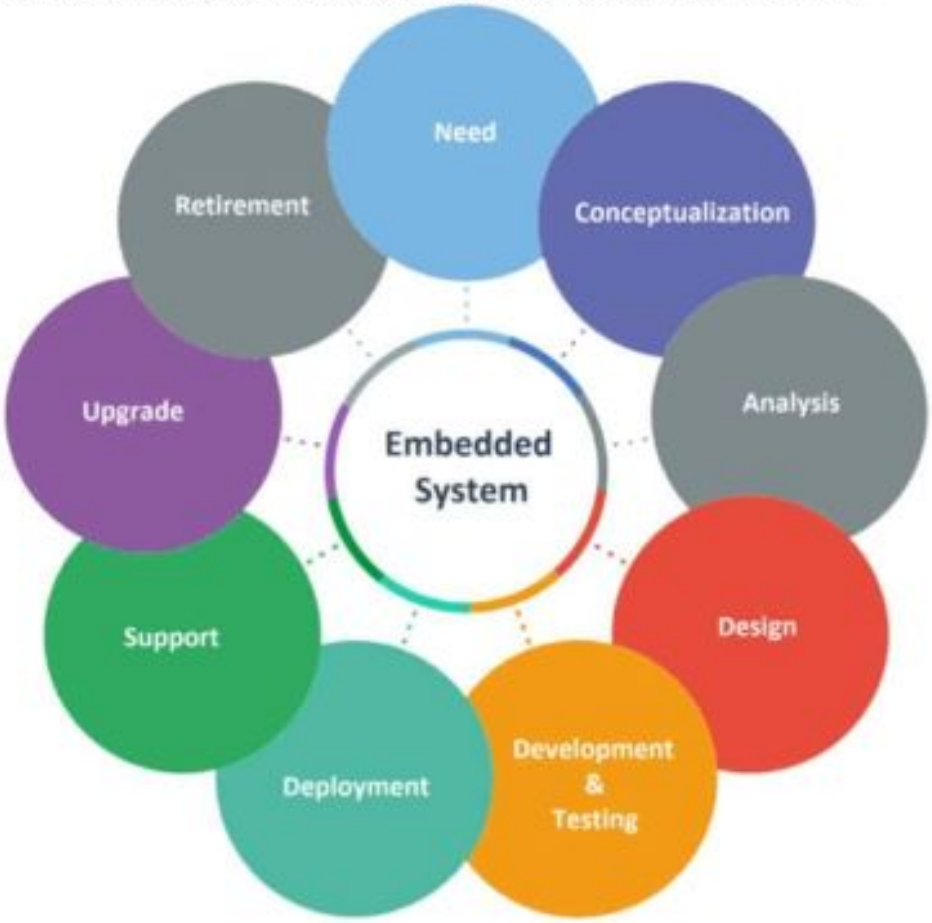
1. Software which provides the services other than available to an application using the OS
2. Software layer between applications and OS on each node in a distributed computing system in network
3. Software which enables an application to communicate between two systems consisting different OSs and hardware.
4. Software for data communication and management in the distributed computing system network.

DESIGN CYCIE

The life cycle is referred as models. A typical simple product contains 5 phases-

- Requirement analysis
- Design
- Development and test
- Deployment
- Maintenance

The embedded product life cycle model contains phases as follows-



i) Need:

The embedded product is an output of 'Need" from an individual/ public/ company. Based on the need a proposal is prepared, reviewed by seniors, approved and then the product goes to product development team.

The types of need could be as follows

- New or custom product development: Need for product which does not exist or as a competitor for an existing product.
- Product re-engineering: The market is dynamic and competitive. Therefore there is always a need of making changes in an existing product design and launching its new version. Product re-engineering includes Product maintenance (technical support to end user), Corrective maintenance (corrective action following a failure) and Preventive maintenance (scheduled maintenance to avoid failure)

ii) Conceptualization:

It is a product development phase which begins after approval. In this stage the following tasks are performed:

- Feasibility study: It is the careful examination of need and it suggests solutions to build product.
- Cost Benefit analysis: This analysis involves identifying total development cost and profits expected.
- Product scope: This means knowing what is in the scope and not in the scope for the product.
- Planning activities: This covers various plans required for product development.

iii) Analysis:

This stage starts after the conceptualization phase is approved by the client. It concentrates on developing functional model of product. The product is defined in detail with respect to input, process and output. This stage determines the function performed by product.

- Analysis and documentation: This phase analyzes business needs and purpose of product. It also addresses various functional aspects and quality attributes.
- Interface definition and documentation: This defines interface between product and other parts of systems.
- Defining Test plan and procedure: This defines the tests to be performed and what should be included in the test. Some tests that are carried out are- Unit testing (unit/ module level), Integration testing (Integrating each module), System testing (functional aspects) and User acceptance testing (meeting all requirements).

iv) Design:

The entire design of product as per requirements is done in this phase.

v) Development and testing:

This phase transforms design into realizable product.

vi) Deployment:

Deployment is nothing but launching first fully functional model of product. It includes some important tasks as follows-

- Notification of product deployment: Launching ceremony details to stake holders and public.
- Execution of training plan: Train the end user.
- Product installation: Install product to ensure it is fully functional.
- Product – Post implementation review: To determine success of product.

vii) Support:

Support means operation and maintenance of product in product environment. The activities are-

- To set up a dedicated support wing
- To identify bugs and areas of improvement.

viii) Upgrade:

It is necessary to upgrade the product already present in market. Upgrades deals with feature enhancement, bug fixes, etc.

ix) Retirement/ Disposal:

The product is declared as obsolete and is discontinued from market due to revolutionary technology changes.

Embedded system Design issues and code design issues :

Design Requirement:

Embedded System must satisfy the following:

1. Real time operation reactive to external events.
2. conform to size and weight limits.
3. Budget power and cooling consumption.
4. Satisfy safety and reliability requirements.
5. Meet tight cost targets.

Real time/reactive operation:

Real time operation:

Correctness of the computation depends on the time at which it is delivered.

System design must also consider worst case performance.

On complicated architecture predicting worst case performance is difficult.

Example of Real time operation are signal processing and mission critical systems.

Reactive operation:

Software executes in response to an external event.

These events can either be periodic or aperiodic.

Periodic events results with guanteed performance whereas aperiodic events may end up with worst case situation.

Design challenge:

worst case design analysis

Embedded computers are physically located within artifacts.

The form factors are

- Dictated by aesthetics
- Existing in pre electronic versions

weight is considered as a major factor in transportation and portable systems.

For example the mission critical system has much more stringent size and weight requirements than the others because of its use in a flight vehicle.

Small size and low weight:

Design challenge:

Non-rectangular, non-planar geometries.

packaging and integration of digital, analog, and power circuits to reduce size.

Safe and reliable:

Embedded system failures may result in severe damages.

For example in mission-critical applications such as aircraft flight control, embedded system failures may result in following:

- Severe personal injury.
- equipment damage.

Embedded systems that fails cannot tolerate the added cost of redundancy in hardware or processing capacity needed for traditional fault tolerance techniques.

Design challenge:

Accurate thermal modeling.

De-rating components differently for each design, depending on operating environment.

Cost sensitivity:

Cost is one of the issue in developing a system. A little change in cost affects the manufacturing quantity of the system.

Design challenge:

Variable "design margin" to permit trade-off between product robustness and aggressive cost optimization.

System level requirements:

To be competitive in market designer should consider the following:

1. End-Product utility
2. System safety & reliability
3. controlling physical systems
4. Power management.

End-Product utility:

Embedded products are typically sold on the basis of capabilities, features, and system cost.

Embedded system mechanisms and their associated I/O are largely defined by the application.

Software is used to coordinate the mechanisms and define their functionality.

Finally, computer hardware is made available as infrastructure to execute the software and interface it to the external world.

System Safety & reliability:

Mechanical safety backups are activated when the computer system loses control.

A bigger and more difficult issue at the system level is software safety and reliability.

A set of unexpected circumstances can cause software failures leading to unsafe situations.

Design challenge:

Reliable software.
cheap, available systems using unreliable components.
Electronic vs. non-electronic design tradeoffs.

Controlling physical systems:

Embedding a computer to interact with the environment by monitoring and controlling external machinery.

For this, analog inputs and outputs must be transformed to and from digital signal levels.

significant need to operate motors, light fixtures, and other actuators.

These lead to a large computer circuit board dominated by non-digital components.

"smart" sensors and actuators (that contain their own analog interfaces, power switches, and small CPUs) may be used to off-load interface hardware from the central embedded computer.

Power Management:

Need for power management to either minimize heat production or conserve battery power.

with evolution of laptops, significantly lower power is needed in order to run from inexpensive batteries for 20 days in some applications and up to 5 years in others.

Design challenge:

ultra-low power design for long-term battery operation.

component acquistion:

Embedded system are more application driven than a typical technology-driven desktop computer design.

There may be more leeway in component selection.

component acquistion costs can be taken into account when optimizing system life-cycle cost.

For example, the cost of a component generally decreases with quantity, so design decisions for multiple designs should be coordinated to share common components to the benefit of all.

Long-term component availability:

This redesign might need to take place even if the system is no longer in production, depending on the availability of a replacement system.

This problem is a significant concern on the distributed example system.

Design challenge:

cost-effectively update old designs to incorporate new components.

Use of Scope and Logic Analysis for System Verification

Scope :-

Scope of embedded system in the present times, embedded also has found immediate application in telecommunications, defense instruments, railroad networks, consumer electronics, electronic payments, and Auto and industry.

Work :-

ES works by incorporating a rugged motherboard into an industrial enclosure with associated I/O and Embedded OS also to fulfil a function in a embedded environment

combinational logic to switch an electrical
signal between zero ("0") to downstream
blocks in digital design combinational
logic uses these bits to send and
receive data within embedded systems
Logic gates are the physical devices that
enable processing of many 1's and 0's

Software testing :-

Embedded sw testing is testing
in an embedded sw testing

is similar to other testing types.
The embedded also be tested for
their performance, consistency and
validated as per the requirements of the
client or the also development team.

Types of hlw :-

Acceptance testing
Analog signature analysis
Analog verification
Automated optical inspection
Automated x-ray inspection
Automatic test equipment.

VALLUVAR COLLEGE OF SCIENCE AND MANAGEMENT,KARUR

DEPARTMENT OF COMPUTER SCIENCE AND APPLICATIONS

EMBEDDED SYSTEMS

SUB.CODE: P16CSE2A **CLASS:I-M.Sc.,(CS)**

UNIT-I

2 MARK:

1.Define system.

2. Define Embedded systems.

3. What are the essential units of processor?

4. List out the types of the processor.

5. What is microprocessor?

6. What is microcontroller?

7. Define ROM image.

8. List out the types of language processors.

9. Define ISR.

10. Define RTOS.

11. List out any five structural units in a processor.

12. What is memory and list out its type?

13. Define RAM.

14. Define ROM.

15. List out the types of ROM.

16. What is serial access memory?

17. What is direct access memory?

18. Define cache memory.

19. What is virtual memory?

20.List out the design parameters to select the appropriate memory type.

21.What are all the needs to be factored during processor selection for an embedded system?

5 MARK:

22) List out and explain the classification of embedded systems.

23) Discuss about DSP and ASSP.

24) Explain about the software in high level languages.

25) List out and explain the software tools in designing of embedded systems.

26) Briefly explain about the structural units in a processor.

27) How to select processor for embedded systems?

28) How to select memory for embedded systems?

10 MARK:

29) Explain about i) Final machine implementable software for a product

ii)Software in processor specific assembly language.

30) Explain about the memory devices.

31) Elaborate in detail about the allocation of memory to program segments and blocks and memory map of a system.

UNIT-II

2 MARK:

1.Define ISR.

2. What is context in embedded system?

3. What is context switching in embedded system?

4. Define interrupt latency.

5. What is interrupt service deadline?

6. what are the criteria by which appropriate programming language is chosen for embedded system of a given system?

7. what is the most important feature in C that makes it popular for an embedded system?

8. What is the advantage of polymorphism, when programming using C++?

9. What is a preprocessor directive?
10. Differentiate macros and functions.

11. Define data structure.

12. What is array?

5 MARK:

13) Briefly explain interrupt servicing mechanisms.

14) Explain about the context and periods for context switching.

15) Briefly explain about the assembly language programming and high level programming.

16) List out the program elements and its uses.

17) List out and explain the include directive for the inclusion of files.

18) Explain about the loops and conditions.

19) Explain the concept of table and hash table.

20) Explain in detail about the function pointers and function queues.
21) What is function calls and explain its usage.

10 MARK:

22) Explain in detail about the data structures with neat diagram.

23) What is object oriented programming and write the advantages and disadvantages of C++ and java.

UNIT-III

2 MARK:

1. What is software analysis?
2. What is data-flow?
3. What is CDFG?
4. Define software implementation.
5. What is the benefit of the FSM?
6. Define petri nets.
7. What is state transition function?
8. Define petri table.
9. What is multiprocessor system?

5 MARK:

10) Briefly explain about the DFG model.

11) Write java program elements and explain its usage.

12) Briefly explain the CDFG model.

13) Explain about the state machine programming model and give some examples.

14) Explain about synchronous data flow model.

15) Differentiate between Functions,ISR, Tasks.

16) What is meant by a pipe? How does a pipe differ from queue?

10 MARK:

17) Explain in detail about the program models with an example.

18) Discuss about the modeling of multi processor system.

19) Explain with one example each, APEG,SDFG,HSDFG

UNIT-IV

1.What is meant by RTOS?

2.what is the role of RTOS in Embedded Systems?

3 Advantages of RTOS in Embedded Systems?

4.What are the common OS services?

6.What are the Example for RTOS?

7.What are the two types of RTOS?

8.what are the Hard and soft real time systems?

9.What are the interrupt in RTOS?

10. What is interrupt routine in RTOS?

11.What is RTOS task scheduling?

12.what are the algorithm used in scheduling?

13. what are the two types of task scheduling ?

14.what is interrupt latency in RTOS?

15. What is interrupt response time?

16. How does a preemptive scheduling works?

17. How does a non preemptive scheduling works?

18. Define performance metrics in RTOS?

19.What are the characteristics of RTOS ?

5 MARK:

20) Briefly explain I/O system in RTOS.

21) List out and explain the RTOS services.

22) Explain about the preemptive scheduling.

23) Discuss about the scheduling of periodic, sporadic and aperiodic tasks.

24) Briefly explain the advanced scheduling algorithms using the problabistic petri nets.

10 MARK:

25) Explain in detail about the OS services in embedded systems.

26) Explain in detail about the interrupt routines in RTOS environment and handling of interrupt source calls.

27) How to design an embedded system using RTOS.

28) List out and explain the following scheduling models i) Cooperative scheduling model

ii) Cooperative scheduling with precedence constraints iii) Cyclic and round robin scheduling

UNIT-V

2 MARK:

1) What are the approaches for the embedded system?

2) What are the components of the embedded system project management?

3) What is meant by embedded system design?

4) Explain codesign issues?

5) What is embedded system design process?

6) What is design metrics in embedded system?

7) What is the Design Cycle?

8) What is target system in embedded system?

9) What is a logic analyzer used for?

10) What are the Hardware–Software Codesign?

11) What is the phase representation in design cycle?

12) what is the use of software tools for development in embedded system?

13)What are the softwares used in embedded system?

14) What is the scope of embedded system?

15) What is the use of logic analyzer in embedded system?

16) What are the categories of a logical analyzer?

17) What are the core processors for project management in embedded system?

18) Define co-design.

19) Define co-design activities.

20) What is embedded hardware testing?

5 MARK:

21) How embedded system works in project management?

22) Explain about the design cycle.

23) List out and explain the uses of target system.

24) What are all the issues in embedded system design? and explain it.

10 MARK:

25) Elaborate in detail about embedded system design and codesign issues.

26) Discuss the uses of software tools for development.

27) Explain the uses of scope and logic analysis for system hardware tests.

VALLUVAR COLLEGE OF SCIENCE AND MANAGEMENT,KARUR

DEPARTMENT OF COMPUTER SCIENCE AND APPLICATIONS

EMBEDDED SYSTEMS

SUB.CODE: P16CSE2A **CLASS:I-M.Sc.,(CS)**

UNIT-I

2 MARK:

1.Define system.

2. Define Embedded systems.

3. What are the essential units of processor?

4. List out the types of the processor.

5. What is microprocessor?

6. What is microcontroller?

7. Define ROM image.

8. List out the types of language processors.

9. Define ISR.

10. Define RTOS.

11. List out any five structural units in a processor.

12. What is memory and list out its type?

13. Define RAM.

14. Define ROM.

15. List out the types of ROM.

16. What is serial access memory?

17. What is direct access memory?

18. Define cache memory.

19. What is virtual memory?

20.List out the design parameters to select the appropriate memory type.

21.What are all the needs to be factored during processor selection for an embedded system?

5 MARK:

22) List out and explain the classification of embedded systems.

23) Discuss about DSP and ASSP.

24) Explain about the software in high level languages.

25) List out and explain the software tools in designing of embedded systems.

26) Briefly explain about the structural units in a processor.

27) How to select processor for embedded systems?

28) How to select memory for embedded systems?

10 MARK:

29) Explain about i) Final machine implementable software for a product

ii)Software in processor specific assembly language.

30) Explain about the memory devices.

31) Elaborate in detail about the allocation of memory to program segments and blocks and memory map of a system.

UNIT-II

2 MARK:

1.Define ISR.

2. What is context in embedded system?

3. What is context switching in embedded system?

4. Define interrupt latency.

5. What is interrupt service deadline?

6. what are the criteria by which appropriate programming language is chosen for embedded system of a given system?

7. what is the most important feature in C that makes it popular for an embedded system?

8. What is the advantage of polymorphism, when programming using C++?

9. What is a preprocessor directive?
10. Differentiate macros and functions.

11. Define data structure.

12. What is array?

5 MARK:

13) Briefly explain interrupt servicing mechanisms.

14) Explain about the context and periods for context switching.

15) Briefly explain about the assembly language programming and high level programming.

16) List out the program elements and its uses.

17) List out and explain the include directive for the inclusion of files.

18) Explain about the loops and conditions.

19) Explain the concept of table and hash table.

20) Explain in detail about the function pointers and function queues.
21) What is function calls and explain its usage.

10 MARK:

22) Explain in detail about the data structures with neat diagram.

23) What is object oriented programming and write the advantages and disadvantages of C++ and java.

UNIT-III

2 MARK:

1. What is software analysis?
2. What is data-flow?
3. What is CDFG?
4. Define software implementation.
5. What is the benefit of the FSM?
6. Define petri nets.
7. What is state transition function?
8. Define petri table.
9. What is multiprocessor system?

5 MARK:

10) Briefly explain about the DFG model.

11) Write java program elements and explain its usage.

12) Briefly explain the CDFG model.

13) Explain about the state machine programming model and give some examples.

14) Explain about synchronous data flow model.

15) Differentiate between Functions,ISR, Tasks.

16) What is meant by a pipe? How does a pipe differ from queue?

10 MARK:

17) Explain in detail about the program models with an example.

18) Discuss about the modeling of multi processor system.

19) Explain with one example each, APEG,SDFG,HSDFG

UNIT-IV

1.What is meant by RTOS?

2.what is the role of RTOS in Embedded Systems?

3 Advantages of RTOS in Embedded Systems?

4.What are the common OS services?

6.What are the Example for RTOS?

7.What are the two types of RTOS?

8.what are the Hard and soft real time systems?

9.What are the interrupt in RTOS?

10. What is interrupt routine in RTOS?

11.What is RTOS task scheduling?

12.what are the algorithm used in scheduling?

13. what are the two types of task scheduling ?

14.what is interrupt latency in RTOS?

15. What is interrupt response time?

16. How does a preemptive scheduling works?

17. How does a non preemptive scheduling works?

18. Define performance metrics in RTOS?

19.What are the characteristics of RTOS ?

5 MARK:

20) Briefly explain I/O system in RTOS.

21) List out and explain the RTOS services.

22) Explain about the preemptive scheduling.

23) Discuss about the scheduling of periodic, sporadic and aperiodic tasks.

24) Briefly explain the advanced scheduling algorithms using the problabistic petri nets.

10 MARK:

25) Explain in detail about the OS services in embedded systems.

26) Explain in detail about the interrupt routines in RTOS environment and handling of interrupt source calls.

27) How to design an embedded system using RTOS.

28) List out and explain the following scheduling models i) Cooperative scheduling model

ii) Cooperative scheduling with precedence constraints iii) Cyclic and round robin scheduling

UNIT-V

2 MARK:

1) What are the approaches for the embedded system?

2) What are the components of the embedded system project management?

3) What is meant by embedded system design?

4) Explain codesign issues?

5) What is embedded system design process?

6) What is design metrics in embedded system?

7) What is the Design Cycle?

8) What is target system in embedded system?

9) What is a logic analyzer used for?

10) What are the Hardware–Software Codesign?

11) What is the phase representation in design cycle?

12) what is the use of software tools for development in embedded system?

13)What are the softwares used in embedded system?

14) What is the scope of embedded system?

15) What is the use of logic analyzer in embedded system?

16) What are the categories of a logical analyzer?

17) What are the core processors for project management in embedded system?

18) Define co-design.

19) Define co-design activities.

20) What is embedded hardware testing?

5 MARK:

21) How embedded system works in project management?

22) Explain about the design cycle.

23) List out and explain the uses of target system.

24) What are all the issues in embedded system design? and explain it.

10 MARK:

25) Elaborate in detail about embedded system design and codesign issues.

26) Discuss the uses of software tools for development.

27) Explain the uses of scope and logic analysis for system hardware tests.