Design and Development of Embedded Product – Firmware Design and Development – Design Approaches, Firmware Development Languages.

FIRMWARE DESIGN AND DEVELOPMENT

Introduction

- Embedded firmware is responsible for controlling various peripherals of the embedded hardware and generating responses in accordance with the functional requirements mentioned in the requirements for the particular product
- Firmware is considered as the master brain of the embedded systems
- Imparting intelligence to an embedded system is a one time process and it can happens at any stage of the design
- Once the intelligence is imparted to the embedded product, by embedding the firmware in the hardware, the product start functioning properly and will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware occurs
- Designing an embedded firmware requires understanding of embedded product hardware like, various component interfacing, memory map details I/O port details, configuration and register details of various hardware chips used and some programming language.
- Embedded firmware development process start with conversion of firmware requirements into a program model using modeling tools like UML or flow chart based representation
- UML diagram gives diagrammatic representation of the decision items to be taken and the task to be performed
- Once the program modeling is created, next step is the implementation of the task and actions by capturing the model using a language which is understandable by the target processor
- Following gives an overview of the various steps involved in the embedded firmware design and development

EMBEDDED FIRMWARE DESIGN APPROACHES

•Firmware design approaches depends on the

–Complexity of the function to be performed

–Speed of operation required ..

–Etc

•Two basic approaches for firmware design

1. Conventional Procedure based Firmware Design/Super Loop Design
2. Embedded Operating System Based Design

1. SUPER LOOP BASED APPROACH

- This approach is applied for the applications that are not time critical and the response time is not so important
- Similar to the conventional procedural programming where the code is executed task by task
- Task listed at the top of the program code is executed first and task below the first task are executed after completing the first task
- It is True procedural one```
- In multiple task based systems, each task executed in serial
- Firmware execution flow of this will be as following

1.Configure the common parameter and perform initialization for various hardware components, memory, registers etc.

2.Start the first task and execute it

3.Execute the second task

4.Execute the next task

5.….

6.….

7.Execute the last defined task

8.Jump back to the first task and follow the same flow

- From the firmware execution sequence, it is obvious that the order in which the task to be executed are fixed and they are hard coded in the code itself
- Operations are infinite loop based approach

  In terms of C program code as:

```
Void main(){
configuration();
 initializations();
 while(1){
    task1();
    task2();
```

```
…..
taskn();} }
```

- Almost all task in embedded applications are non-ending and are repeated infinitely throughout the operation
- By analyzing C code we can see that the task 1 to n are performed one after another and when the last task is executed, the firmware execution is again redirected to task 1 and it is repeated forever in the loop.This repetition is achieved by using an infinite loop(while(1))Therefore Super loop based Approach
- Since the task are running inside an infinite loop, the only way to come out of the loop is either

  –Hardware reset

  or

  –Interrupt assertion
- A Hardware reset brings the program execution back to the main loop
- Whereas the interrupt suspend the task execution temporarily and perform the corresponding interrupt routine and on completion of the interrupt routine it restart the task execution from the point where it got interrupted
- Super Loop based design does not require an OS, since there is no need for scheduling which task is to be executed and assigning priority to each task.
- In a super Loop based design, the priorities are fixed and the order in which the task to be executed are also fixed
- Hence the code for performing these task will be residing in the code memory without an operating system image
- This type of design is deployed in low-costembedded products where the response time is not time critical
- Some embedded products demand this type of approach if some tasks itself are sequential
- For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of the card, authenticating the operation, reading/writing etc..It should strictly follows a specified sequence and the combination of these series of tasks constitutes a single task namely read write
- There is no use in putting the sub tasks into independent task and running them parallel
- Example of " Super Loop Based Design" is

–Electronic video game toy containing keypad and display unit

- The program running inside the product must be designed in such a way that it reads the key to detect whether user has given any input and if any key press is detected the graphic display is updated. The keyboard scanning and display updating happens at a reasonable high rate
- Even if the application misses the key press , itwon’t create any critical issue Rather it will treated as a bug in the firmware

Drawback of Super Loop based Design

- Major drawback of this approach is that any failure in any part of a single task will affect the total system

  If the program hang up at any point while executing a task, it will remain there forever and ultimately the product will stop functioning

–Some remedial measures are there

•Use of Hardware and software Watch Dog Timers (WDTs) helps in coming out from the loop when an unexpected failure occurs or when the processor hang up

–May cause additional hardware cost and firmware overhead

- Another major drawback is lack of real timeliness

  – If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases.This brings the probability of missing out some events

  – For example in a system with keypad, there will be task for monitoring the keypad connected I/O lines and this need not be the task running while you press the keys

  – That is key pressing event may not be in sync with the keypad press monitoring task within the firmware

  – To identify the key press, you may have to press the key for a sufficiently long time till the keypad status monitoring task is executed internally.

  – Lead to lack of real timeliness

2. Embedded Operating System Based Approach

- Contains OS, which can be either a General purpose Operating System (GPOS) or real Time Operating System (RTOS)

General purpose Operating System (GPOS) based design

- GPOS based design is very similar to the conventional PC based Application development where the device contain an operating system and you will be creating and running user applications on top of it
- Examples of Microsoft Windows XP OS are PDAs, Handheld devices/ Portable Devices and point of Sale terminals

- Use of GPOS in embedded product merges the demarcation of Embedded systems and General Purpose systems in terms of OS
- For developing applications on the top of the OS , OS supported APIs are used
- OS based applications also requires 'Driver Software' for OS based applications also requires 'Driver Software' for different hardware present on the board to communicate with them

RTOS based design

- RTOS based design approach is employed in embedded product demanding Real Time Responses
- RTOS respond in a timely and predictable manner to events
- RTOS contain a real time Kernel responsible for performing pre- emptive multi tasking scheduler for scheduling the task, multiple thread etc.
- RTOS allows a flexible scheduling of system resources like the CPU and Memory and offer some way to communicate between tasks

–Examples of RTOS are

•Windows CE, pSOS, VxWorks, ThreadX, Micro C/OS II,Embedded Linux, Symbian etc…

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

For embedded firmware development you can use either

–Target processor/controller specific language (Assembly language) or

–Target processor/ controller independent language (High level languages) or

–Combination of Assembly and high level language

ASSEMBLY LANGUAGE BASED DEVELOPMENT

- Assembly language is human readable notation of machine language whereas machine language is a processor understandable language. Processor deal only with binaries
- Machine language is a binary representation and it consist of 1s and 0s.Machine language is made readable by using specific symbols called 'mnemonics'. Hence machine language can be considered as an interface between processor and programmer
- Assembly language and machine languages are processor dependant and assembly program written for one processor family will not work with others
- Assembly language programming is the task of writing processor specific machine code in mnemonics form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler
- Assembly language program was the most common type of programming adopted in the beginning of software revolution

- Some OS dependant task requires low level languages
- In particular assembly language is used in writing low level interaction between the OS and the hardware, for instance in device drivers
- The general format of an assembly language instruction is

Opcode   Operand

-Opcode tells what to do
-Operand gives the information to do the task

The operand may be single operand, dual operand or more

example–

MOV A, #30

–Move the decimal value 30 to the accumulator register of 8051

–Here MOV A is the opcode and 30 is Operand

–Same instruction in machine language like this

01110100   00011110

–Here the first 8 bit represent opcode MOV A and next 8 bit represent the operand 30

- The mnemonic INC A is an example for the instruction holding operand implicitly in the Opcode
  The machine language representation is 00000100
  –This instruction increment the 8051 Accumulator register content by 1
- LJMP *16 BIT ADDRESS* is an example of dual operand instruction
- The machine language for the same is
  10   addr_bit15 to addr_bit8   addr_bit7 to addr_bit0
- The first binary data is the representation of LJMP machine code
- The first operand that immediately follow the opcode represent the bit 8 to 15 of the 16 bit address to which the jump is requited and the second
- operand represent the bit 0 to 7 of the address to which the jump targeted

- Assembly language instructions are written in one per line
- A machine code program thus consisting of a sequence of assembly language instructions, where each statement contains a mnemonic(opcode+operand)
- Each line of assembly language program split into four field as given below

LABEL OPCODE OPERAND COMMENTS

- Label is an optional field. A label is an identifier to remembering where data or code is located
- LABEL is commonly used for representing

  –A memory location, address of a program, sub-routine, code portion etc…

  –The max length of the label differ between assemblers. Labels are always suffixed by a colon and begin with a valid character. Labels can contain numbers from 0 to 9 and special character _

  –Labels are used for representing subroutine names and jump locations in Assembly language programming

```
DELAY:   MOV R0, #255        ;load Register R0 with 255
          DJNZ R1, DELAY     ;Decrement R1 and loop
                             ; till R1=0

                   RET       ;return to calling program
```

The assembly program contain a main routine which start at address 0000H and it may or may not contain subroutines..In main program subroutine is invoked by the assembly instruction

LCALL *DELAY*

Executing this instruction transfers the program flow to the memory address referenced by the 'LABEL' DELAY .While assembling the code a ';' inform the assembler that the rest of the part coming in a line after the ';' symbol is comments and simply ignore it •Each assembly instruction should be written in a separate line.More than one ASM code lines are not allowed in single line
•In the previous example LABEL DELAY represent the reference to the start of the subroutine

```
DELAY:          MOV R0, #255          ;load Register R0 with 255

                DJNZ R1, DELAY        ;Decrement R1 and loop
                                      ; till R1=0

                RET                   ;return to calling program
```

We can directly replace the LABEL by putting desired address first and then writing assembly code for the routine

```
ORG 0100H
```

```
                    MOV R0, #255              ;load Register R0 with 255
                    DJNZ R1, DELAY            ;Decrement R1 and loop
                                              ; till R1=0
                    RET                       ;return to calling program
```

ORG 0100H is not an assembly language instruction; it is an assembler directive instruction. It tells the assembler that the instruction from here onwards should be placed at location starting from 0100H.Assembler directive instructions are known as 'pseudo ops'

They are used for

–Determining the start address of the program (eg. ORG 0100H)

–Determining the entry address of the program (eg. ORG 0100H)

–Reserving the memory for data variables, arrays and structures (eg. Var EQU 70H)

–Initializing variable values (e.g. val DATA 12H)

- ➢ EQU directive is used for allocating memory to a variable and DATA directive is used for initializing a variable with data
- ➢ No machine codes are generated for the 'Pseudo-ops'
- ➢ Assembly language program written in assembly code is saved as .asm file or an .src file
- ➢ Any text editor can be used for writing assembly instructions
- ➢ Similar to other high level programming, you can have multiple source files called modules in assembly language programming.
- ➢ Each module is represented by .asm or .src file
- ➢ This approach is known as modular programming
- ➢ Modular program is employed when program is too complex or too big.
- ➢ In modular programming the entire code is divided into sub modules and each module is made reusable
- ➢ Modular programs are usually easy to code,debug and alter

Conversion of assembly language into machine language is carried out by a sequence of operations

1. SOURCE FILE TO OBJECT FILE TRANSLATION
2. Library file creation and usage
3. Linker and Loader
4. Object to hex file convertor

1. **SOURCE FILE TO OBJECT FILE TRANSLATION**

•Translation of assembly code to machine code is performed by assembler

•The assemblers for different target machines are different and it is common that assemblers from multiple vendors are available in the market for the same target machines

•Some assemblers are supplied by single vendor only

•Some assemblers are freely available

•Some are commercial and requires license from vendors

–A51 Macro Assembler from Keil software is a popular assembler for 8051 family microcontroller

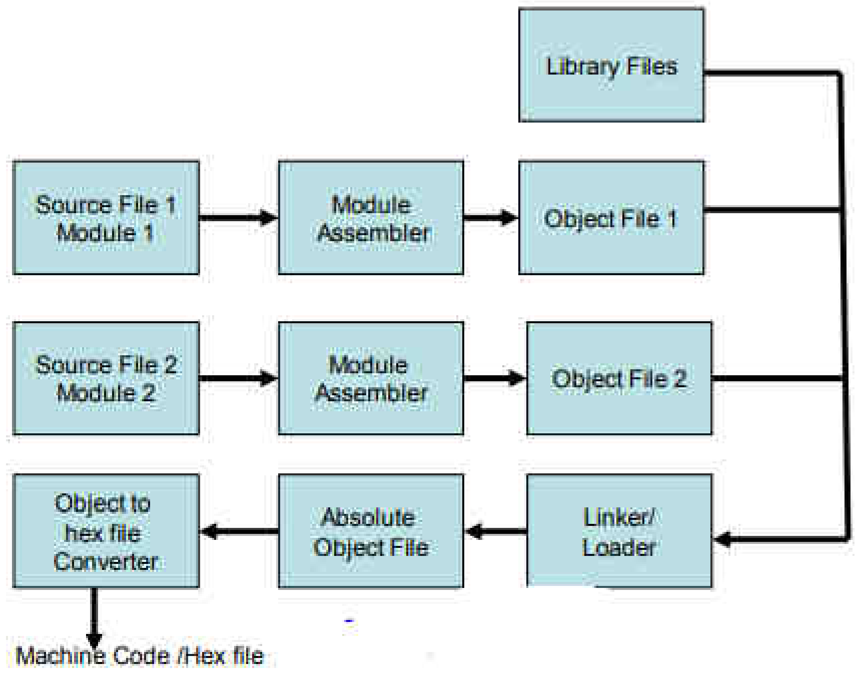


Fig .Assembling language to machine language convertion process

Each source module is written in assembly and is stored in .src or .asm file.Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions.On assembling of each .src/.asm file a corresponding object file is created with extension .obj.The object file does not contain the absolute address of where the generated code need to be placed on the program memory and hence it is called relocatable segment. It can be placed at any code memory location and it is responsibility of the linker/loader to assign absolute address for this module.Absolute address allocation is done at absolute object file creation stage.Each module can share variables and subroutine among them.Exporting a variable from a module is done by declaring that variable as PUBLIC in source module.Importing a variable or a function from a module is done by declaring that variable or function as EXTRN in the module where it is going to be accessed

•PUBLIC keyword inform the assembler that the variable / function need to be exported

•EXTRN inform that the variable/function need to be imported from some other modules

•Corresponding to a variable /function declared as PUBLIC in a module, there can be one or modules using these variables/function using EXTRN keyword

•For all those modules using variables or function with EXTRN keyword, there should be one and only one module which export those variables/functions PUBLIC keyword

•If more than one module in a project tries to export variables or functions with the same name using PUBLIC keyword, it will generate linker errors

•If a variable or function declared as EXTRN in one or two modules, there should be one module defining these variables or function and exporting them using PUBLIC keyword

•If no module in a project export the variable or functions which are declared as EXTRN in other modules it will generate linker warnings or error depending on the error level/warning level setting of the linker

2. Library file creation and usage

- ➢ Libraries are specially formatted, ordered program collection of object modules that may be used by the linker at a later time
- ➢ When a linker process a library, only those object modules in the library that are necessary to create the program are used
- ➢ Library files are generated with the extension '.lib'
- ➢ Library file is some kind of source code hiding technique
- ➢ If you don't want to reveal the source code behind the various functions you have written in your program and at the same time you want them to be distributed to application developers for making use of them in their applications,you can supply them as library files and give them the details of the public functions available from the library
- ➢ For using a library file in a project, add library to the project
- ➢ If you are using a commercial version of assembler suit for your development, the vendor of utility may provide you pre written library files for performing multiplication, floating point arithematic, etc. as an add-on utility

•Example LIB51 from keil software

3.Linker and Locator

- Linker and locator is another software utility responsible for" linking the various object modules in a multi module project and assigning absolute address to each module"
- Linker generate an absolute object module by extracting the object module from the library, if any and those obj files created by the assembler, which is generated by assembling the individual modules of a project
- It is the responsibility of the linker to link any external dependent variables or functions declared on various modules and resolve the external dependencies among the modules
- An absolute object file or modules does not contain any re-locatable code or data
- ALL code and data reside at fixed memory locations
- The absolute object file is used for creating hex files for dumping into the code memory of the processor/controller

•Example 'BL51' from keil software

3. Object to Hex File Converter

- This is the final stage in the conversion of Assembly language to machine understandable language
- Hex file is the representation of the machine code and the hex file is dumped into the code memory of the processor
- Hex file representation varies depending on the target processor make
- For intel processor the target hex file format willbe 'Intel HEX' and for Motorola, hex file should be in 'Motorola HEX' format
- HEX files are ASCII files that contain a hexadecimal representation of target application
- Hex file is created from the final 'Absolute Object File' using the Object to Hex file Converter utility

Example 'OH51' is an example utility for object to hex file convertor in8051 specific controller

Advantage of Assembly Language based Development

Assembly language based development is the most common technique adopted from the beginning of the embedded technology development**.**Thorough understanding of the processor architecture , memory organization , register set and mnemonics is very essential for Assembly Language based Development.Following are the advantages of the same.

1. **Efficient Code Memory and data Memory Usage (Memory Optimization)**

–Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations

–This lead to the less utilization of code memory and efficient utilization of data memory

–Memory is the primary concern in any embedded product

2. **High Performance**

–Optimized code not only improve the code memory usage but also improve the total system performance

–Though effective assembly coding optimum performance can be achieved for target applications

3. **Low level Hardware access**

–Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers and low level interrupt routine etc. are making use of direct assembly coding since low level device specific operation support is not commonly avail with most of the high level language compilers

4. **Code Reverse Engineering**

–Reverse Engineering is the process of understanding the technology behind a product by extracting the information from the finished product

–Reverse engineering is performed by 'hawkers' to reveal the technology behind the proprietary product

–Though most of the product employ code memory protection, if it may be possible to break the memory protection and read the code memory, it can easily be converted into assembly code using dis-assembler program for the target machine

DRAWBACKS OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

1. **High Development time**

–Assembly language programs are much harder to program than high level languages

–Developer must have thorough knowledge of architecture, memory organization and register details of target processor in use

–Learning the inner details of the processor and its assembly instructions are high time consuming and it create delay impact in product development

2. **Developer Dependency**

–There is no common rule for developing assembly language based applications whereas all high level language instruct certain set of rules for application development

–In Assembly language programming, the developers will have the freedom to choose the different memory locations and registers

–Also programming approach varies from developers to developers depending on their taste

–For example moving a data from a memory location to accumulator can be achieved through different approaches

–If the approach is done by a developer is not documented properly at the development stage, it may not be able to recollect at later stage or when a new developer is instruct to analyze the code , he may not be able to understand what is done and why it is done

–Hence upgrading/modifying on later stage is more difficult. This can be solved by Well Documentation

3. Non- Portable

–Target applications written in assembly instructions are valid only for that particular family of processors

•Example—Application written for Intel X86 family of processors

–Cannot be reused for another target processors

–If the target processor changes, a complete rewriting of the application using assembly instructions for the new target processor is required

HIGH LEVEL LANGUAGE BASED DEVELOPMENT

- Any High level language with supported cross compilers for the target processor can be used for embedded firmware development
- Cross Compilers are used for converting the application development in high level language into target processor specific assembly code
- Most commonly used language is C
- C is well defined easy to use high level language with extensive cross platform development tool support

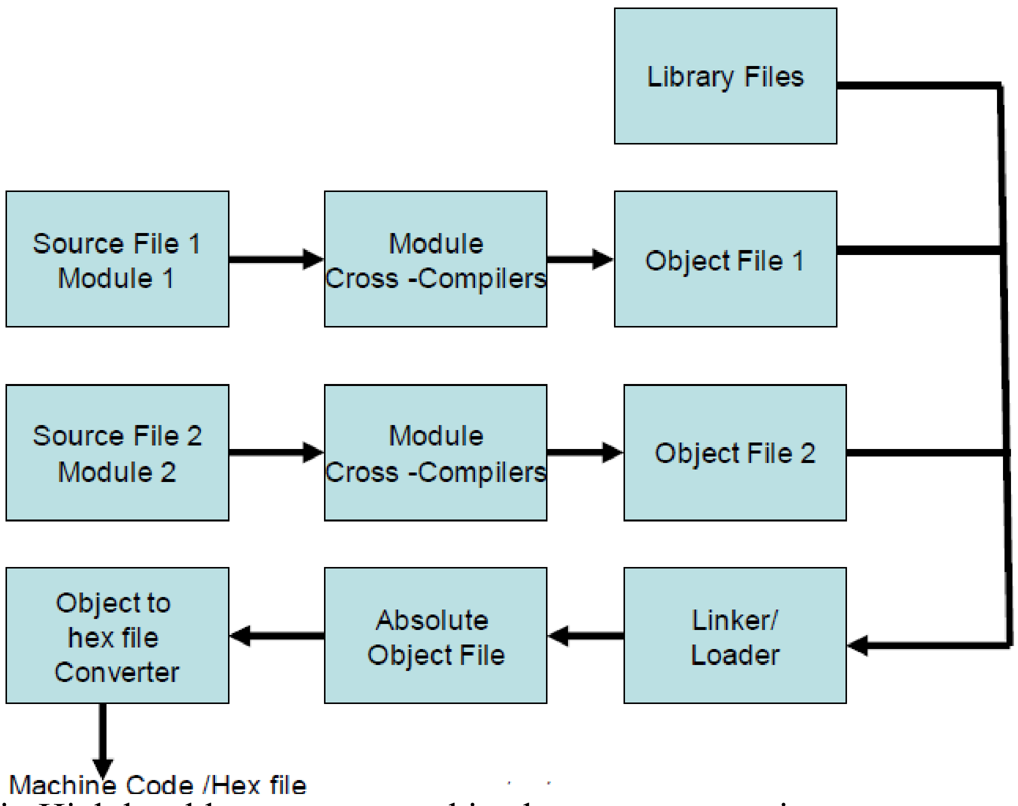


Fig:High level language to machine language convertion process

- The program written in any of the high level language is saved with the corresponding language extension
- Any text editor provided by IDE tool supporting the high level language in use can be used for writing the program
- Most of the high level language support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language
- The source file corresponding to each module is represented by a file with corresponding language extension
- Translation of high level source code to executable object code is done by a cross compiler
- The cross compiler for different high level language for same target processor are different
- Without cross-compiler support a high level language cannot be used for embedded firmware development

–Example C51 Compiler fromKeil

Advantages of High Level Languagebased Development

1. **Reduced Development Time**

–Developers requires less or little knowledge on the internal hardware details and architecture of the target processor

–Syntax of high level language and bare minimal knowledge of memory organization and register details of target processor are the only pre- requisites for high level language based firmware development

–With High level language, each task can be accomplished by lesser number of lines of code compared to the target processor specific assemblylanguage based development

2. **Developer Independency**

–The syntax used by most of the high level languages are universal and a program written in high level language can be easily be understood by a second person knowing the syntax of the language

–High level language based firmware development makes the firmware , developer independent

–High level language always instruct certain set of rules for writing code and commenting the piece of code

3. **Portability**

–Target applications written in high level languages are converted to target processor understandable format by a cross compiler

–An application written in high level language for a particular target processor can be easily converted to another target processor with little effort by simply recompiling the code modification followed by the recompiling the application for the required processor

–This makes the high level language applications are highly portable

Limitations of High level language based development

- Some cross compilers avail for the high level languages may not be so efficient in generating optimized target processor specific instructions
- Target images created by such compilers may be messy and no optimized in terms of performance as well as code size

MIXING ASSEMBLY AND HIGH LEVEL LANGUAGE

- High level language and assembly languages are usually mixed in three ways

  –Mixing assembly language with high level language

–Mixing high level language with Assembly

–In line assembly programming

MixingAssemblyLanguagewith HighlevelLanguage (Assembly Language with ‘C’)

- Assembly routines are mixed with C in situations where entire program is written in C and the cross compiler in use do not have built in support for implementing certain features like Interrupt Service Routine or if the programmer want to take the advantage of speed and optimized code offered by machine code generated by hand written assembly rather than cross compiler generated machine code
- When accessing certain low level hardware, the timing specification may be very critical and a cross compiler generated binary may not be able to offer the required time specifications accurately
- Writing the hardware access routine in processor specific assembly language and invoking it from C is the most advised method to handle such situations

•Mixing C and Assembly is little complicated in the sense-

–the programmer must be aware of how parameters are passed from the C routine to Assembly and

–values are returned from assembly routine to C and

–how the assembly routine is invoked from the C code

•These are cross compiler dependent

•There is no universal rule for it

•You must get the information from the documentation of cross compiler you are using

•Different cross compilers implement these features in different ways depending upon the general purpose registers and the memory supported by the target processor

The Example is to give an idea on how C%!cross compiler perform mixing of C with assembling language

1.Write a simple function in C that passes parameters and return values the way you want your assembly routine to

2.Use the SRC directive (#pragma SRC) so that C compiler generate an SRC file instead of .OBJ file

3.Compile the C code. Since the SRC directive is specified the .SRC file is generated. The .SRC file contain the assembly code generated for the C code you wrote

4.Rename .SRC to .A51 file

5.Edit .A51 file and insert the assembly code you want to execute in the body of the assembly function shell included in the .A51 file

As an example consider the following sample code (Extracted from Keil C51 documentation)

```
#pragma SRC
unsigned char my_assembly_func (unsigned int argument)
{
return (argument + 1);  // Insert dummy lines to access all args and
                        // retvals
}
```

This C function on cross compilation generates the following assembly *SRC* file.

```
NAME    TESTCODE
?PR?_my_assembly_func?TESTCODE        SEGMENT CODE
        PUBLIC   _my_assembly_func
; #pragma SRC
; unsigned char my_assembly_func (
```

```
#pragma SRC
unsigned char my_assembly_func (unsigned int argument)
{
return (argument + 1);  // Insert dummy lines to access all args and
                        // retvals
}
```

This C function on cross compilation generates the following assembly *SRC* file.

```
NAME    TESTCODE
?PR?_my_assembly_func?TESTCODE        SEGMENT CODE
        PUBLIC   _my_assembly_func
; #pragma SRC
; unsigned char my_assembly_func (
        RSEG   ?PR?_my_assembly_func?TESTCODE
        USING    0
_my_assembly_func:
;---- Variable 'argument?040' assigned to Register 'R6/R7' ----
; SOURCE LINE # 2
;    unsigned int argument)
; {
; SOURCE LINE # 4
; return (argument + 1);   // Insert dummy lines to access all args
; and retvals
; SOURCE LINE # 5
        MOV      A,R7
        INC      A
        MOV      R7,A
; }
; SOURCE LINE # 6
?C0001:
        RET
; END OF _my_assembly_func
        END
```

Mixing high level language with assembly(Eg C with assembly language)

Mixing the code written in a high level language like 'C' and Assembly language is useful in the following scenarios:

1. The source code is already available in Assembly language and a routine written in a high level language like 'C' needs to be included to the existing code.
2. The entire source code is planned in Assembly code for various reasons like optimised code, optimal performance, efficient code memory utilisation and proven expertise in handling the Assembly, etc. But some portions of the code may be very difficult and tedious to code in Assembly. For example 16bit multiplication and division in *8051* Assembly Language.
3. To include built in library functions written in 'C' language provided by the cross compiler. For example Built in Graphics library functions and String operations supported by 'C'.

Most often the functions written in 'C' use parameter passing to the function and returns value/s to the calling functions. The major question that needs to be addressed in mixing a 'C' function with Assembly is that how the parameters are passed to the function and how values are returned from the function and how the function is invoked from the assembly language environment. Parameters are passed to the function and values are returned from the function using CPU registers, stack memory and fixed memory. Its implementation is cross compiler dependent and it varies across cross compilers. A typical example is given below for the Keil C51 cross compiler

C51 allows passing of a maximum of three arguments through general purpose registers R2 to R7. If the three arguments are *char* variables, they are passed to the function using registers R7, R6 and R5 respectively. If the parameters are *int* values, they are passed using register pairs (R7, R6), (R5, R4) and (R3, R2). If the number of arguments is greater than three, the first three arguments are passed through registers and rest is passed through fixed memory locations. Refer to C51 documentation for more details. Return values are usually passed through general purpose registers. R7 is used for returning *char* value and register pair (R7, R6) is used for returning *int* value. The 'C' subroutine can be invoked from the assembly program using the subroutine call Assembly instruction (Again cross compiler dependent).

```
E.g.  LCALL     _Cfunction
```

Where *Cfunction* is a function written in 'C'. The prefix _ informs the cross compiler that the parameters to the function are passed through registers. If the function is invoked without the _ prefix, it is understood that the parameters are passed through fixed memory locations.

Inline Assembly

Inline assembly is another technique for inserting target processor/controller specific Assembly instructions at any location of a source code written in high level language 'C'. This avoids the delay in calling an assembly routine from a 'C' code (If the Assembly instructions to be inserted are put in a subroutine as mentioned in the section mixing assembly with 'C'). Special keywords are used to indicate that the start and end of Assembly instructions. The keywords are cross-compiler specific. C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

```
E.g.   #pragma asm
       MOV A, #13H
       #pragma endasm
```

MODULE-1

Introduction to Embedded System: Understanding the Basic Concepts, The Typical Embedded System – Characteristics and Quality attributes.

FUNDAMENTALS OF EMBEDDED SYSTEMS

SYSTEM

- A system is an arrangement in which all its unit assemble work together according to a set of rules.
- It can also be defined as a way of working, organizing or doing one or many tasks according to a fixed plan. For example, a watch is a time displaying system. Its components follow a set of rules to show time. If one of its parts fails, the watch will stop working. So we can say, in a system, all its subcomponents depend on each other.

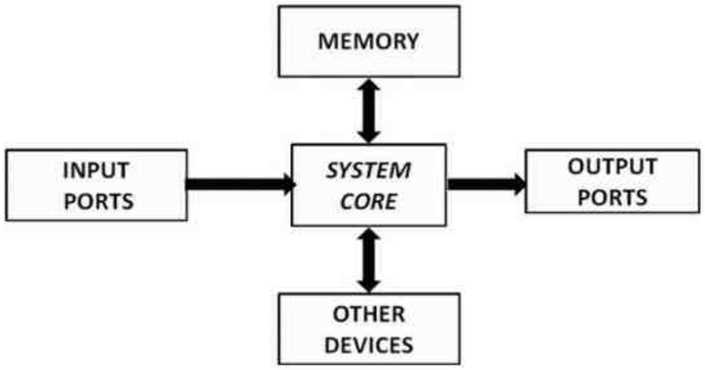
EMBEDDED SYSTEM

- Embedded means something that is attached to another thing.
- An embedded system can be thought of as a computer hardware system having software embedded in it.
- An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task.
- For example, a fire alarm is an embedded system; it will sense only smoke.
- An embedded system has three components − It has hardware. It has application software. It has Real Time Operating system that supervises the application software and provide mechanism to let the processor run a process as per scheduling by following a plan to control the latencies.
- RTOS defines the way the system works. It sets the rules during the execution of application program.
- A small scale embedded system may not have RTOS.
- So we can define an embedded system as a Microcontroller based, software driven, and reliable, real-time control system.
- An embedded system is designed to do a specific job only. Example: a washing machine can only wash clothes, an air conditioner can control the temperature in the room in which it is placed.
- The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.
- An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

An embedded system is a combination of 3 things

- Hardware
- Software
- Mechanical Components

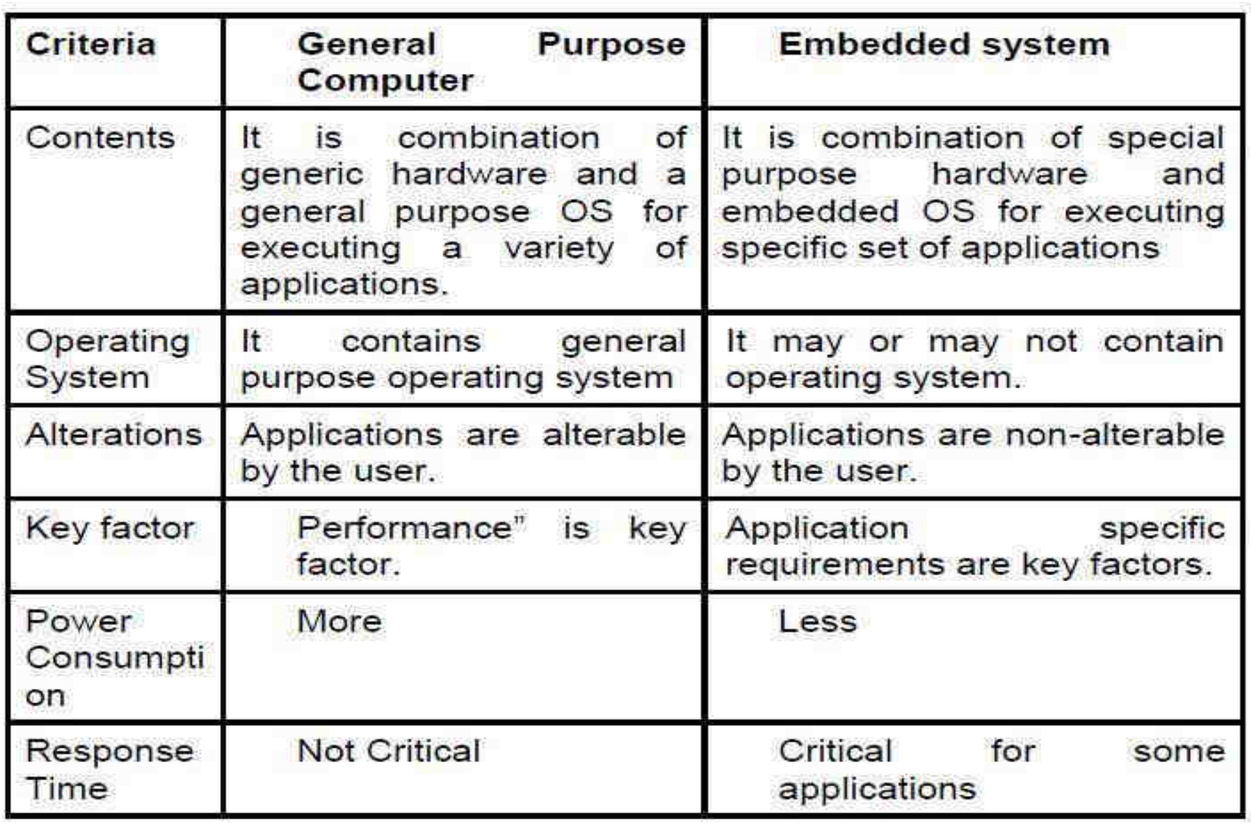
And it is supposed to do one specific task only. Diagrammatically an embedded system can be represented as follows:



EMBEDDED SYSTEM & GENERAL PURPOSE COMPUTER

- The Embedded System and the General purpose computer are at two extremes.
- The embedded system is designed to perform a specific task
- The general purpose computer is meant for general use. It can be used for playing games, watching movies, creating software, work on documents or spreadsheets etc.

Following are certain specific points of difference between embedded systems and general purpose computers:



Criteria	General Purpose Computer	Embedded system
Contents	It is combination of generic hardware and a general purpose OS for executing a variety of applications.	It is combination of special purpose hardware and embedded OS for executing specific set of applications
Operating System	It contains general purpose operating system	It may or may not contain operating system.
Alterations	Applications are alterable by the user.	Applications are non-alterable by the user.
Key factor	Performance" is key factor.	Application specific requirements are key factors.
Power Consumption	More	Less
Response Time	Not Critical	Critical for some applications

Embedded computing systems have to provide sophisticated functionality:

■ ***Complex algorithms:*** The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel utilization.

■ ***User interface:*** Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

■ ***Real time:*** Many embedded computing systems have to perform in real time—if the data is not ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline does not create safety problems but does create unhappy customers—missed deadlines in printers ,for example, can result in scrambled pages.

■ ***Multirate:*** Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of **multirate** behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

Costs of various sorts are also very important:

■ ***Manufacturing cost:*** The total cost of building the system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.

■ ***Power and energy:*** Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

CHARACTERISTICS OF AN EMBEDDED SYSTEM

- **Single-functioned** − an embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.

- **Tightly constrained** − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.

- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.

- **Microprocessors based** − It must be microprocessor or microcontroller based.

- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.

- **Connected** − It must have connected peripherals to connect input and output devices.

- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.

APPLICATION OF EMBEDDED SYSTEM

The application areas and the products in the embedded domain are countless.

1. Consumer Electronics: Camcorders, Cameras.

2. Household appliances: Washing machine, Refrigerator.

3. Automotive industry: Anti-lock breaking system (ABS), engine control.

4. Home automation & security systems: Air conditioners, sprinklers, fire alarms.

5. Telecom: Cellular phones, telephone switches.

6. Computer peripherals: Printers, scanners.

7. Computer networking systems: Network routers and switches.

8. Healthcare: EEG, ECG machines.

9. Banking & Retail: Automatic teller machines, point of sales.

10. Card Readers: Barcode, smart card readers.

CHALLENGES IN EMBEDDED COMPUTING SYSTEM DESIGN

- ***How much hardware do we need?***
- Control over the amount of computing power we apply to our problem.
- Select the type of microprocessor ,amount of memory and the peripheral devices
- To meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important
- Too little hardware and the system fail to meet its deadlines, too much hardware and it becomes too expensive.
- ***How do we meet deadlines?***
- The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster, that makes the system more expensive.
- It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system.
- ***How do we minimize power consumption?***
- In battery-powered applications, power consumption is extremely important. Even in non-battery applications, excessive power consumption can increase heat dissipation.
- One way to make a digital system consume less power is to make it run more slowly, but naively slowing down the system can obviously lead to missed deadlines.
- Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.
- ***How do we design for upgradability?***
- The hardware platform may be used over several product generations or for several different versions of a product in the same generation, with few or no changes.
- We want to be able to add features by changing software.
- ***Does it really work?***
- Reliability is always important when selling products
- Reliability is especially important in some applications, such as safety-critical systems.

Let's consider some ways in which the nature of embedded computing machines makes their design more difficult.

■ ***Complex testing:*** Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

■ ***Limited observability and controllability:*** Embedded computing systems usually do not come with keyboards and screens. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applications we may not be able to easily stop the system to see what is going on inside.

■ ***Restricted development environments:*** The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations. We generally compile code on one type of machine, such as a PC, and download it onto the embedded system. To debug the code, we must usually rely on programs that run on the PC or workstation and then look inside the embedded system.

PERFORMANCE IN EMBEDDED COMPUTING

- Embedded system designers have a very clear performance goal in mind—their program must meet its ***deadline***. The heart of embedded computing is ***real-time computing***
- The program receives its input data; the deadline is the time at which a computation must be finished. If the program does not produce the required output by the deadline, then the program does not work, even if the output that it eventually produces is functionally correct. This notion of deadline-driven programming is at once simple and demanding.
- We need tools to help us analyze the real-time performance of embedded systems; we also need to adopt programming disciplines and styles that make it possible to analyze these programs.
- In order to understand the real-time behavior of an embedded computing system, we have to analyze the system at several different levels of abstraction. Those layers include:
  - ***CPU:*** The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
  - ***Platform:*** The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
  - ***Program:*** Programs are very large and the CPU sees only a small window of the program at a time. We must consider the structure of the entire program to determine its overall behavior.
  - ***Task:*** We generally run several programs simultaneously on a CPU, creating a ***multitasking system***. The tasks interact with each other in ways that have profound implications for performance.
  - ***Multiprocessor:*** Many embedded systems have more than one processor—they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

THE EMBEDDED SYSTEM DESIGN PROCESS

The following figure summarizes the major steps in the embedded system design process. In this top–down view, we start with the system ***requirements***.

In the next step, ***specification***, we create a more detailed description of what we want. The specification states only how the system behaves, not how it is built.

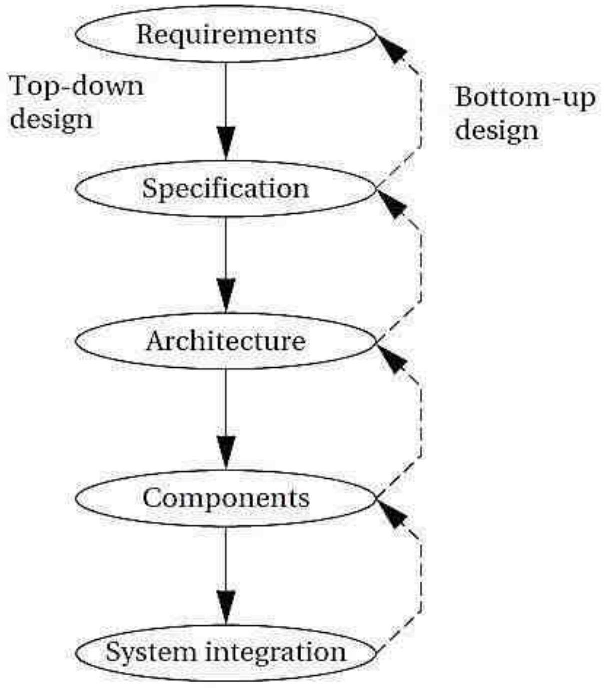


Fig: Major levels of abstraction in the design process

The details of the system's internals begin to take shape when we develop the **architecture**, which gives the system structure in terms of large **components**. Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system.

We also need to consider the major goals of the design:

■ **manufacturing cost**

■ **performance (both overall speed and deadlines)**

■ **power consumption.**

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

■ We must ***analyze*** the design at each step to determine how we can meet the specifications.

■ We must then ***refine*** the design to add detail.

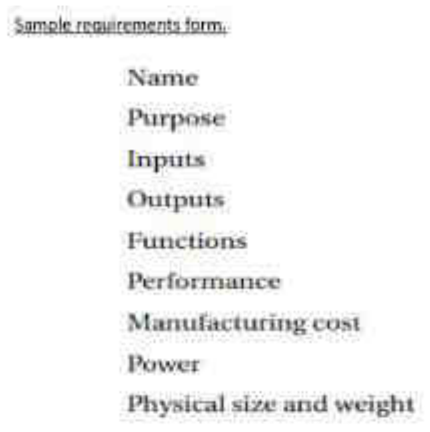
■ And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1.Requirements:

In first step we gather informal description from customers known as requirements ,we then refine that requirement into a specification that contain enough information to start design.Reurements are of 2 types

- Fuctional Reurements
  -focused on function of the system
- Non functional Reurements
  -performance
  -cost
  -physical size and cost
  -Power consumption

Requirement analysis of a large system is complex and time consuming .the following figure shows a sample requirement analysis form that need to filled during the start of a project.


Sample requirements form.

Name
Purpose
Inputs
Outputs
Functions
Performance
Manufacturing cost
Power
Physical size and weight

Name	:	Selection of name that depicts the theme of project
Purpose	:	One or two line description of what the system is supposed to do
Inputs and Outputs:		Type of data,analog digital or mechanical input or output
Functions:		Is a description of what the system does
Performance:		Measure of is the system work precisely?
Manufacturing cost:		Cost of hardware,application softwareincluding all the cost involved in the manufacturing
Power:		Measure of how much power the system consume
Physical size and weight:		It guided to certain architectural design

2. Specification

- The specification is more precise—it serves as the contract between the customer and the architects.
- The specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.
- The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.
- It should also be unambiguous enough that designers know what they need to build.
- If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

3. Architecture

The specification does not say how the system does things, only what the system does.

- Describing how the system implements those functions is the purpose of the architecture.
- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- The creation of the architecture is the first phase of what many designers think of as design.

To understand what an architectural description is, let's look at sample architecture for the moving map. The moving map is a handheld device that displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position. The moving map obtains its position from the GPS, a satellite-based navigation system. Following figure shows sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

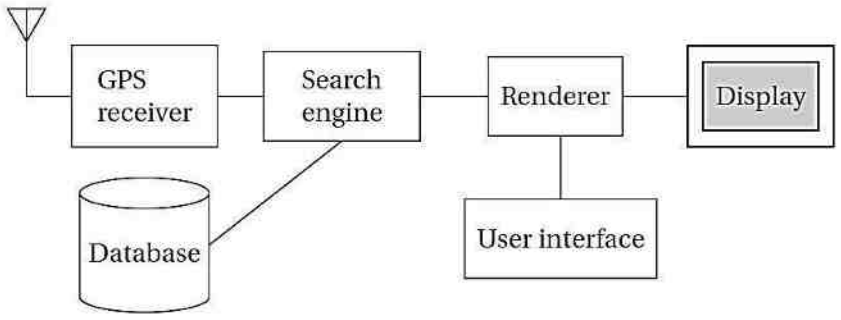


Fig: Block diagram for the moving map.

This block diagram is still quite abstract—we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on.

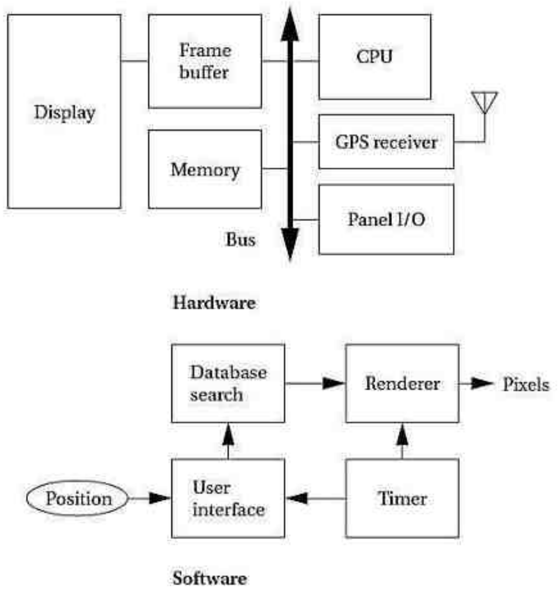


Fig: Hardware and software architectures for the moving map.

4. Designing Hardware and Software Components

- The architectural description tells us what components we need.
- The components will in general include both hardware—FPGAs, boards, and so on—and software modules.
- Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components.
- In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules.

5.System Integration

- After component design the next step is to integrate them together and see whether the system works correctly as per the requirements.
- Many bugs appear only at this stage.
- Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

Formalism for system design

Unified Modeling Language (UML) - UML was designed to be useful at many levels of abstraction in the design process. UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction. UML is an ***object-oriented*** modeling language.

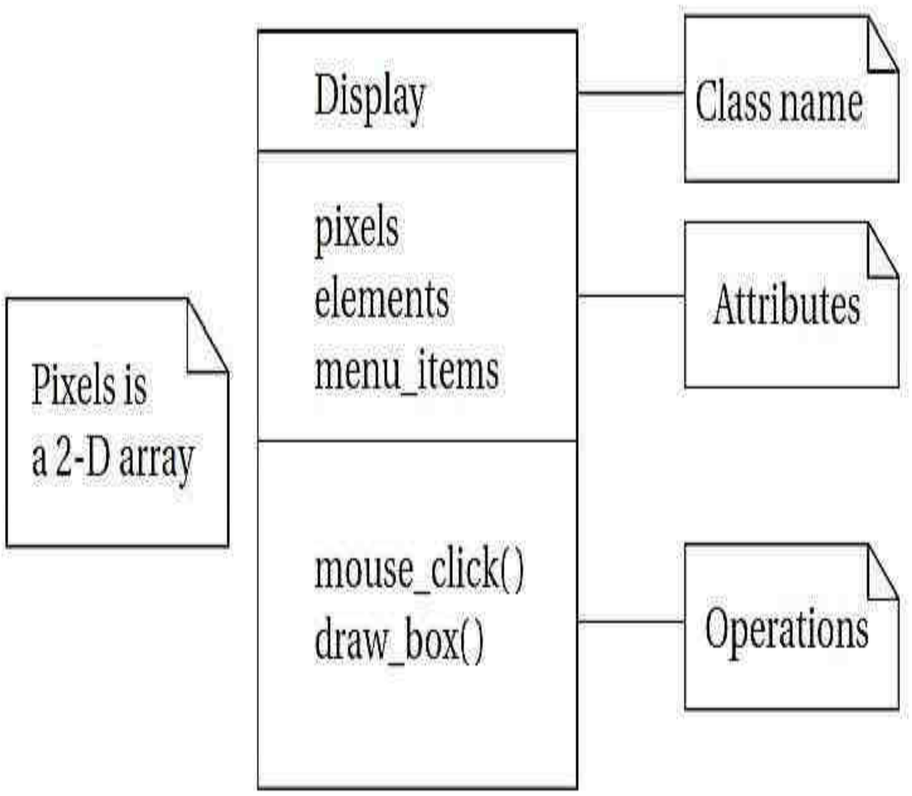
Structural Description

- By ***structural description***, we mean the basic components of the system.
- The principal component of an object-oriented design is the ***object.*** An object includes a set of ***attributes*** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure.

- In some cases, we will add the type of the attribute after A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values.

- A class defines the attributes that an object may have. It also defines the ***operations*** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object. The UML description of the ***Display*** class is shown below

Fig: A class in UML notation



- The class has the name that we saw used in the *d* 1 object since *d* 1 is an instance of class ***Display***.
- The ***Display*** class defines the *pixels* attribute seen in the object; remember that when we instantiate the class an object, that object will have its own memory so that different objects of the same class have their own values for the attributes.
- Other classes can examine and modify class attributes; if we have to do something more complex than use the attribute directly, we define a behavior to perform that function.
- A class defines both the ***interface*** for a particular type of object and that object's ***implementation.***
- When we use an object, we do not directly manipulate its attributes—we can only read or modify the object's state through the operations that define the interface to the object.

.

`

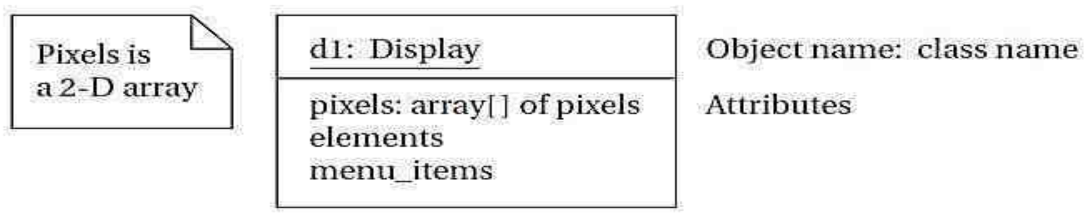


Fig: An object in UML notation.

There are several types of *relationships* that can exist between objects and classes:

■ ***Association*** occurs between objects that communicate with each other but have no ownership relationship between them.

■ ***Aggregation*** describes a complex object made of smaller objects.

■ ***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.

■ ***Generalization*** allows us to define one class in terms of another.

Unified Modeling Language allows us to define one class in terms of another. An example is shown below, where we ***derive*** two particular types of displays. The first, ***BW_display***, describes a black- and-white display. This does not require us to add new attributes or operations, but can specialize both to work on one-bit pixels. The second, ***Color_map_display***, uses a graphic device known as a color map to allow the user to select from behaviors—for example, large number of available colors even with a small number of bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors.

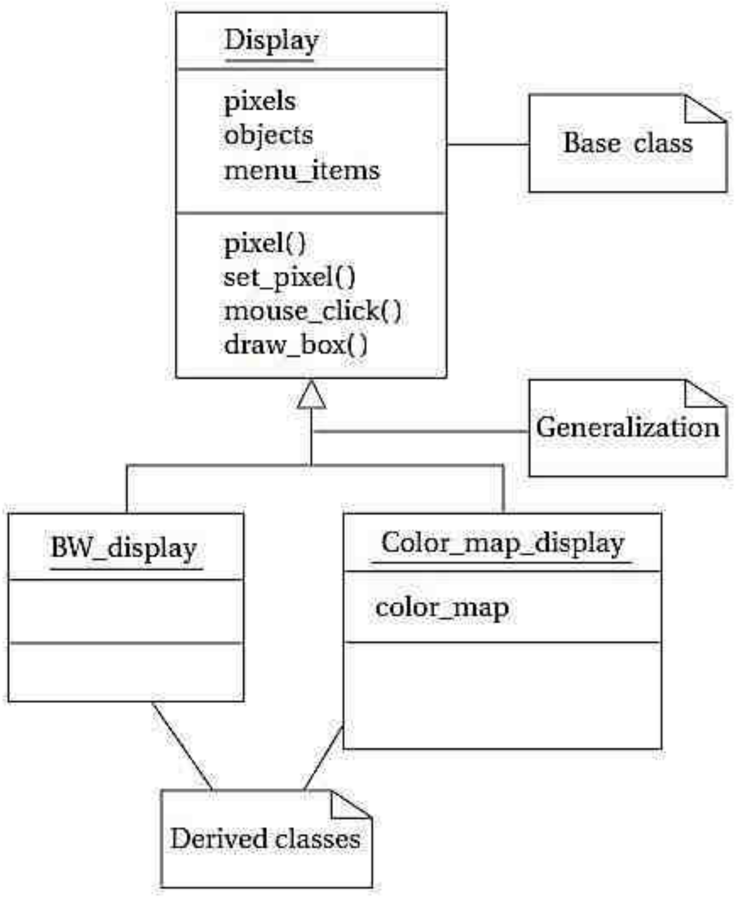


Fig: Derived classes as a form of generalization in UML.

- A ***derived class*** inherits all the attributes and operations from its ***base class***.
- In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class.
- This relation is transitive—if *Display* were derived from another class, both ***BW_display*** and ***Color_map_display*** would inherit all the attributes and operations of *Display's* base class as well.
- **Unified Modeling Language** considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead.
- UML also allows us to define ***multiple inheritances***, in which a class is derived from more than one base class.

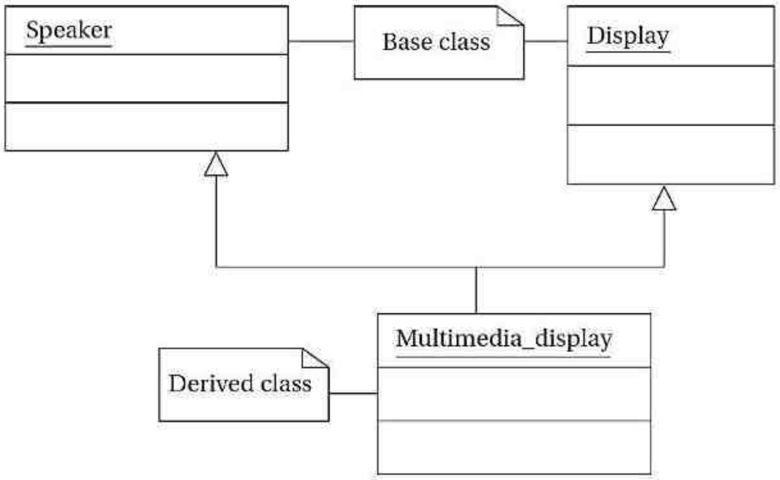
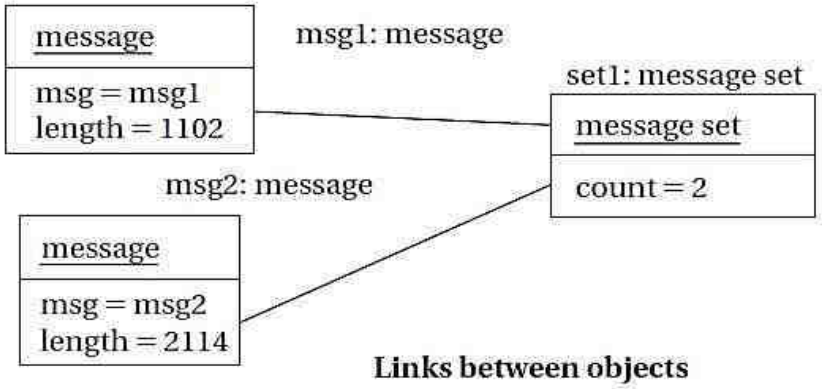
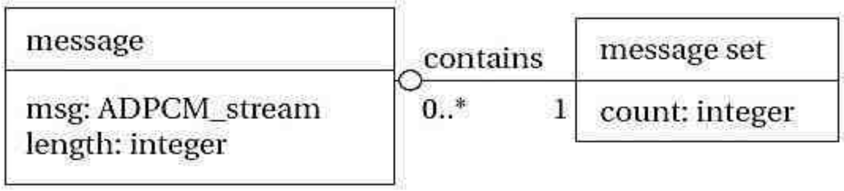


Fig: Multiple inheritances in UML.

- In the above figure we have created a ***Multimedia_display*** class by combining the ***Display*** class with a ***Speaker*** class for sound.
- The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*.
- A ***link*** describes a relationship between objects; association is to link as class is to object. Following figure shows examples of links and an association



Links between objects



Association between classes

- When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in the above example) and points to the active messages. In this case, the link defines the ***contains*** relation.
- The association is drawn as a line between the two labeled with the name of the association, namely, ***contains.***

Behavioral Description

- We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a ***state machine***. Following figure shows UML states; the transition between two states is shown by a skeleton arrow.

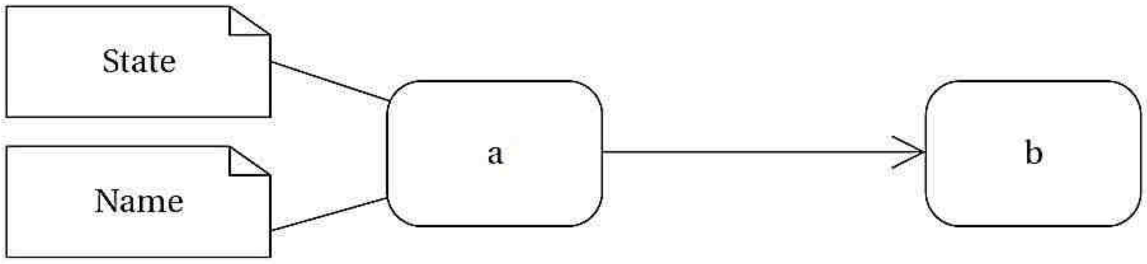


Fig: A state and transition in UML.

- The state machines will not rely on the operation of a clock, as in hardware; rather, changes from one state to another are triggered by the occurrence of ***events***.
- An event is some type of action. The event may originate outside the system, such as a user pressing a button. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.

- The three types of events defined by UML are

1. A ***signal*** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a <<*signal*>>. The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.
2. A ***call event*** follows the model of a procedure call in a programming language.
3. ***time-out event*** causes the machine to leave a state after a certain amount of time. The label *tm(time-value)* on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

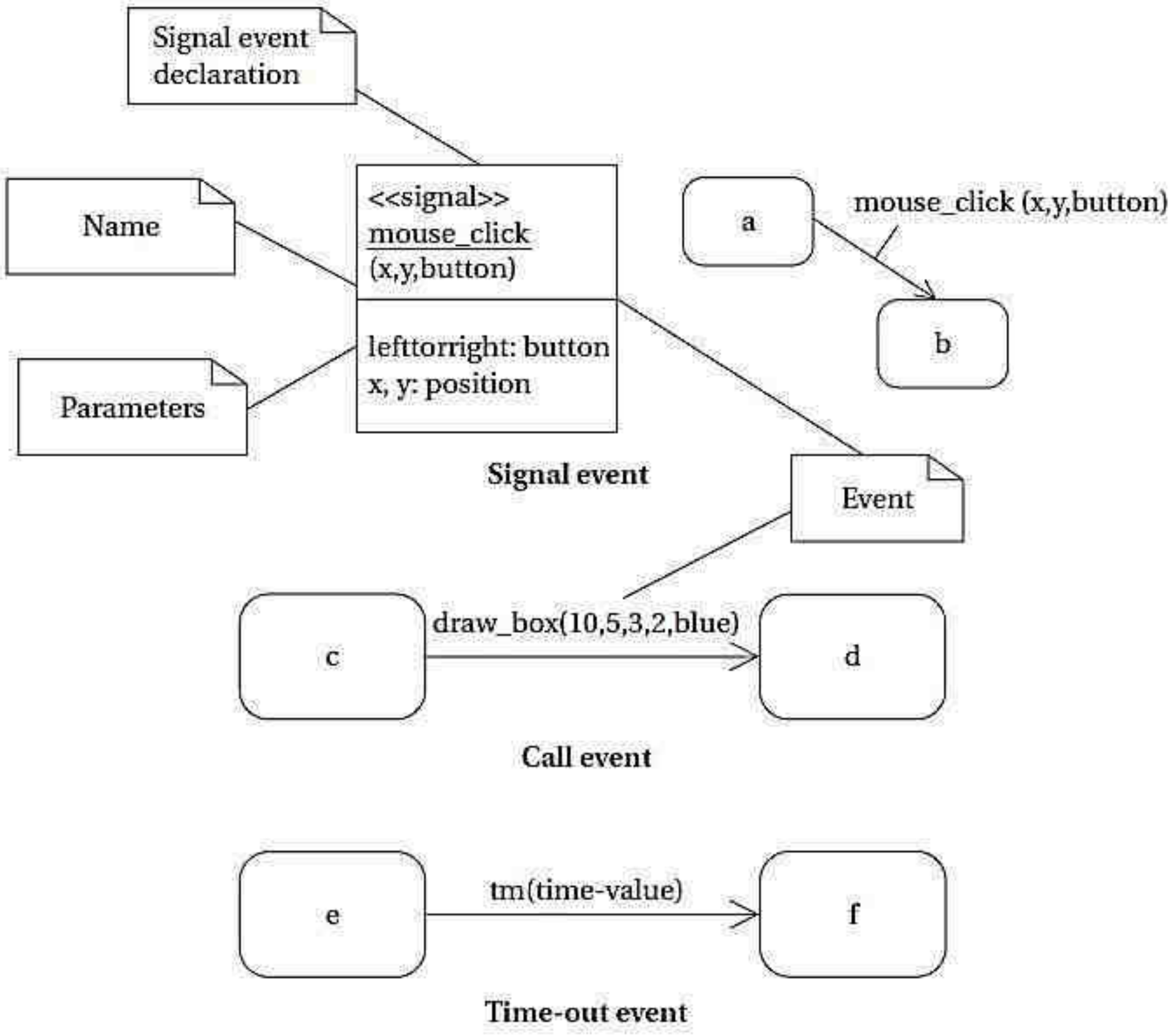


Fig: Signal, call, and time-out events in UML.

sequence diagram

- It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved. In this case, we can create a sequence diagram, like the one for a mouse click scenario shown below
- A ***sequence diagram*** is somewhat similar to a hardware timing diagram,
- The time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram.
- The sequence diagram is designed to show a particular scenario or choice of events
- In the following ***sequence diagram***, the sequence shows what happens when a mouse click is on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all the objects remain alive for the entire sequence, but in other cases objects may be created or destroyed during processing. The boxes along the

lifelines show the *focus of control* in the sequence, that is, when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually execute the menu call. The find_region( ) call is internal to the display object, so it does not appear as an event in the diagram.

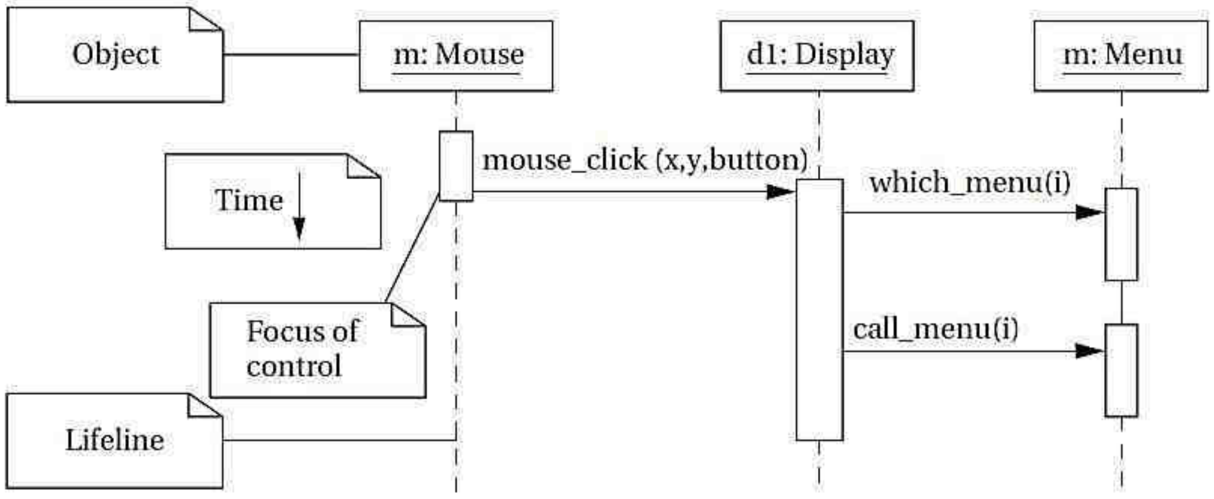


Fig: A sequence diagram in UML.

CHARACTERISTICS OF EMBEDDED SYSTEM

Following are some of the characteristics of an embedded system that make it different from a general purpose computer:

1. **Application and Domain specific**

   An embedded system is designed for a specific purpose only. It will not do any other task.

   Ex. A washing machine can only wash, it cannot cook

   Certain embedded systems are specific to a domain: ex. A hearing aid is an application that belongs to the domain of signal processing.

2. **Reactive and Real time**

   Certain Embedded systems are designed to react to the events that occur in the nearby environment. These events also occur real-time.

Ex. An air conditioner adjusts its mechanical parts as soon as it gets a signal from its sensors to increase or decrease the temperature when the user operates it using a remote control.An embedded system uses Sensors to take inputs and has actuators to bring out the required functionality.

3. **Operation in harsh environment**

   Certain embedded systems are designed to operate in harsh environments like very high temperature of the deserts or very low temperature of the mountains or extreme rains.

   These embedded systems have to be capable of sustaining the environmental conditions it is designed to operate in.

4. **Distributed**

   Certain embedded systems are part of a larger system and thus form components of a distributed system.

   These components are independent of each other but have to work together for the larger system to function properly.

   Ex. A car has many embedded systems controlled to its dash board. Each one is an independent embedded system yet the entire car can be said to function properly only if all the systems work together.

5. **Small size and weight**

   An embedded system that is compact in size and has light weight will be desirable or more popular than one that is bulky and heavy.

Ex. Currently available cell phones. The cell phones that have the maximum features are popular but also their size and weight is an important characteristic. For convenience users prefer mobile phones than phablets. (phone + tablet pc)

6. **Power concerns**

   It is desirable that the power utilization and heat dissipation of any embedded system be low.

   If more heat is dissipated then additional units like heat sinks or cooling fans need to be added to the circuit.

   If more power is required then a battery of higher power or more batteries need to be accommodated in the embedded system.

QUALITY ATTRIBUTES OF EMBEDDED SYSTEM

These are the attributes that together form the deciding factor about the quality of an embedded system.

There are two types of quality attributes are:-

Operational Quality Attributes.

These are attributes related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.

Non-Operational Quality Attributes.

These are attributes **not** related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.

These are the attributes that are associated with the embedded system before it can be put in operation.

Operational Attributes

1. **Response**

Response is a measure of quickness of the system.It gives you an idea about how fast your system is tracking the input variables.

Most of the embedded system demand fast response which should be real-time.

2. **Throughput**

Throughput deals with the efficiency of system.

It can be defined as rate of production or process of a defined process over a stated

period of time. In case of card reader like the ones used in buses, throughput means how much transaction the reader can perform in a minute or hour or day.

3. **Reliability**

   Reliability is a measure of how much percentage you rely upon the proper functioning of the system .

   Mean Time between failures and Mean Time To Repair are terms used in defining system reliability.

   Mean Time between failures can be defined as the average time the system is functioning before a failure occurs.

   Mean time to repair can be defined as the average time the system has spent in repairs.

4. **Maintainability**

   Maintainability deals with support and maintenance to the end user or a client in case of technical issues and product failures or on the basis of a routine system checkup

   It can be classified into two types :-

   Scheduled or Periodic Maintenance

   This is the maintenance that is required regularly after a periodic time interval.

   Example :

   Periodic Cleaning of Air Conditioners

   Refilling of printer cartridges.

Maintenance to unexpected failure

This involves the maintenance due to a sudden breakdown in the functioning of the system.

Example:

Air conditioner not powering on

Printer not taking paper in spite of a full paper stack

5. **Security**

Confidentiality, Integrity and Availability are three corner stones of information security.Confidentiality deals with protection data from unauthorized disclosure. Integrity gives protection from unauthorized modification.

Availability gives protection from unauthorized user

Certain Embedded systems have to make sure they conform to the security measures.

Ex. An Electronic Safety Deposit Locker can be used only with a pin number like a password.

Safety

Safety deals with the possible damage that can happen to the operating person and environment due to the breakdown of an embedded system or due to the emission of hazardous materials from the embedded products.

A safety analysis is a must in product engineering to evaluate the anticipated damage and determine the best course of action to bring down the consequence of damages to an acceptable level.

Non Operational Attributes

1. Testability and Debug-ability

- It deals with how easily one can test his/her design, application and by which mean he/she can test it.
- In hardware testing the peripherals and total hardware function in designed manner
- Firmware testing is functioning in expected way
- Debug-ability is means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system

2. Evolvability

- For embedded system, the qualitative attribute “Evolvability” refer to ease with which the embedded product can be modified to take advantage of new firmware or hardware technology.

3. Portability

- Portability is measured of “system Independence”.
- An embedded product can be called portable if it is capable of performing its operation as it is intended to do in variousenvironments irrespective of different processor and or controller and embedded operating systems.

4. Time to prototype and market

Time to Market is the time elapsed between the conceptualization of a product and time at which the product is ready for selling or use

Product prototyping help in reducing time to market.

Prototyping is an informal kind of rapid product development in which important feature of the under consider are develop.

In order to shorten the time to prototype, make use of all possible option like use of reuse, off the self component etc**.**

5. Per unit and total cost

Cost is an important factor which needs to be carefully monitored. Proper market study and cost benefit analysis should be carried out before taking decision on the per unit cost of the embedded product.

When the product is introduced in the market, for the initial period the sales and revenue will be low

There won't be much competition when the product sales and revenue increase.

During the maturing phase, the growth will be steady and revenue reaches highest point and at retirement time there will be a drop in sales volume.

MODULE 4

Integration of Hardware & Firmware

Integration of hardware and firmware deals with the embedding of firmware into the target hardware board. It is the process of '***Embedding Intelligence***' to the product. The embedded processors/controllers used in the target board may or may not have built in code memory. For non-operating system based embedded products, if the processor/controller contains internal memory and the total size of the firmware is fitting into the code memory area, the code memory is downloaded into the target controller/processor. If the processor/controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/controller, an external dedicated EPROM/FLASH memory chip is used for holding the firmware. This chip is interfaced to the processor/control-

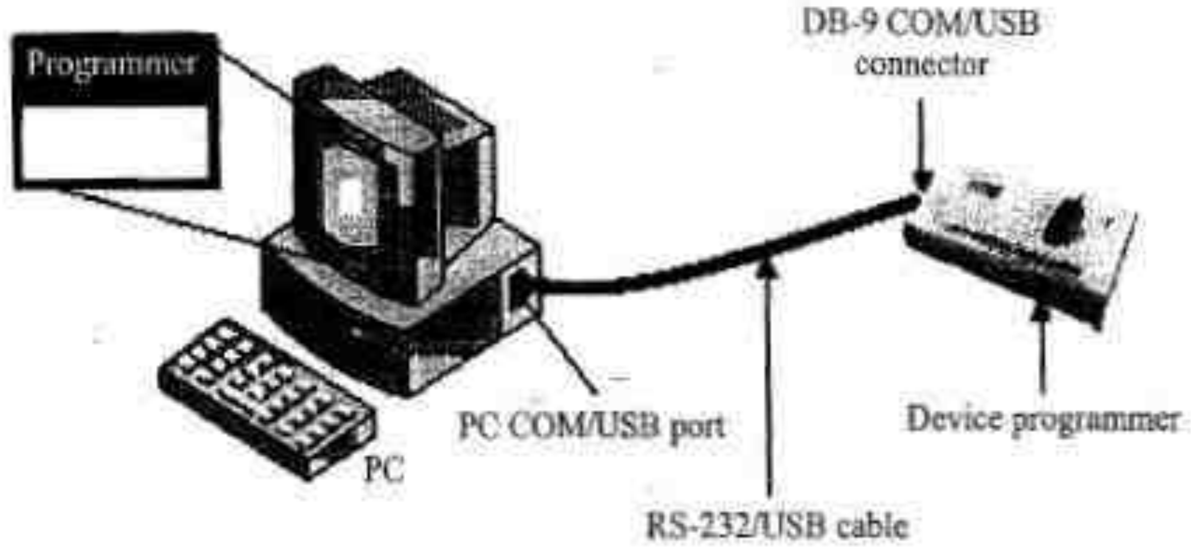
- Out-of-Circuit Programming
- In System Programming(ISP)
- In Application Programming
- Use of Factory programmed chip

Out-of-Circuit Programming

- Its performed outside the target board

- The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and its programmed with the programming device
- The programming device is a dedicated unit which contains the necessary hardware circuits generate the programming signals

- Most of this devices are capable to support different family of devices
- The programmer contains ZIF socket locking pin to hold the device to be programmed
- Programmer is interfaced to pc through RS232/USB/Parallel port interface



Interfacing of device programmer with pc

- The sequence of operations for embedded firmware with a programmer is

1. Connect the programming device to specified port(USB/COM/Parallel port)
2. Power up the device
3. Execute the programming utility on the pc and ensure proper connectivity between pc and programmer
4. Unlock the ZIF socket by turning the lock pin
5. Insert the device to be programmed into the open socket
6. Lock ZIP socket
7. Select the device name from the list of supported devices
8. Load the hex file which is to be embedded into the device

9. Program the device by program option of utility program
10. Wait till the completion of programming operation
11. Ensure that programming is successful by checking the status LED on the programmer
12. Unlock the ZIF socket & take device out of programmer

If security is required, enable the memory protection on the utility before programming the device

Only EEPROM/FLASH memory are erasable

Drawback

1. High development time
2. Not suitable for batch production(Option Gang programmer)
3. Very difficult to update firmware(Especially after the product is released)

In System Programming

- Firmware is embedded into the target device without removing it from the target board
- The target device must have ISP support
- Normally serial interface communication and protocols preferred
- ISP with SPI(Serial peripheral interface) protocol
  - The primary i/o lines in this methods are

    MOSI

    MISO

    SCK

    RST

    GND
  - PC acts as master and target device acts like slave in ISP
  - The key player behind ISP is a factory programmed memory called boot ROM.

  - It contains a set of low level instruction APIs and those APIs allows the mc/p to perform flash programming

- ISP with SPI protocol-Programming e.g for Atmel AT 89S

1. Apply supply voltage b/w VCC and GND pin of target chip
2. Set RST pin to HIGH
3. If crystal is not connected across pins XTAL1 and XTAL2,apply 3 to 24 Mhz clock to XLAL1 and wait for 10s
4. Enable serial programming by sending the programming enable serial instruction to pin MOSI/P1.5
5. The code or data array is programmed one byte at a time by supplying the address and data together with the appropriate write instruction
6. Any memory location is verified by using read instruction, which returns the content at the selected address at serial output MISO/P1.6
7. After successfully programming the device, set RST to low or turn off the chip power supply and turn it to ON to commence the normal operation

In Application Programming

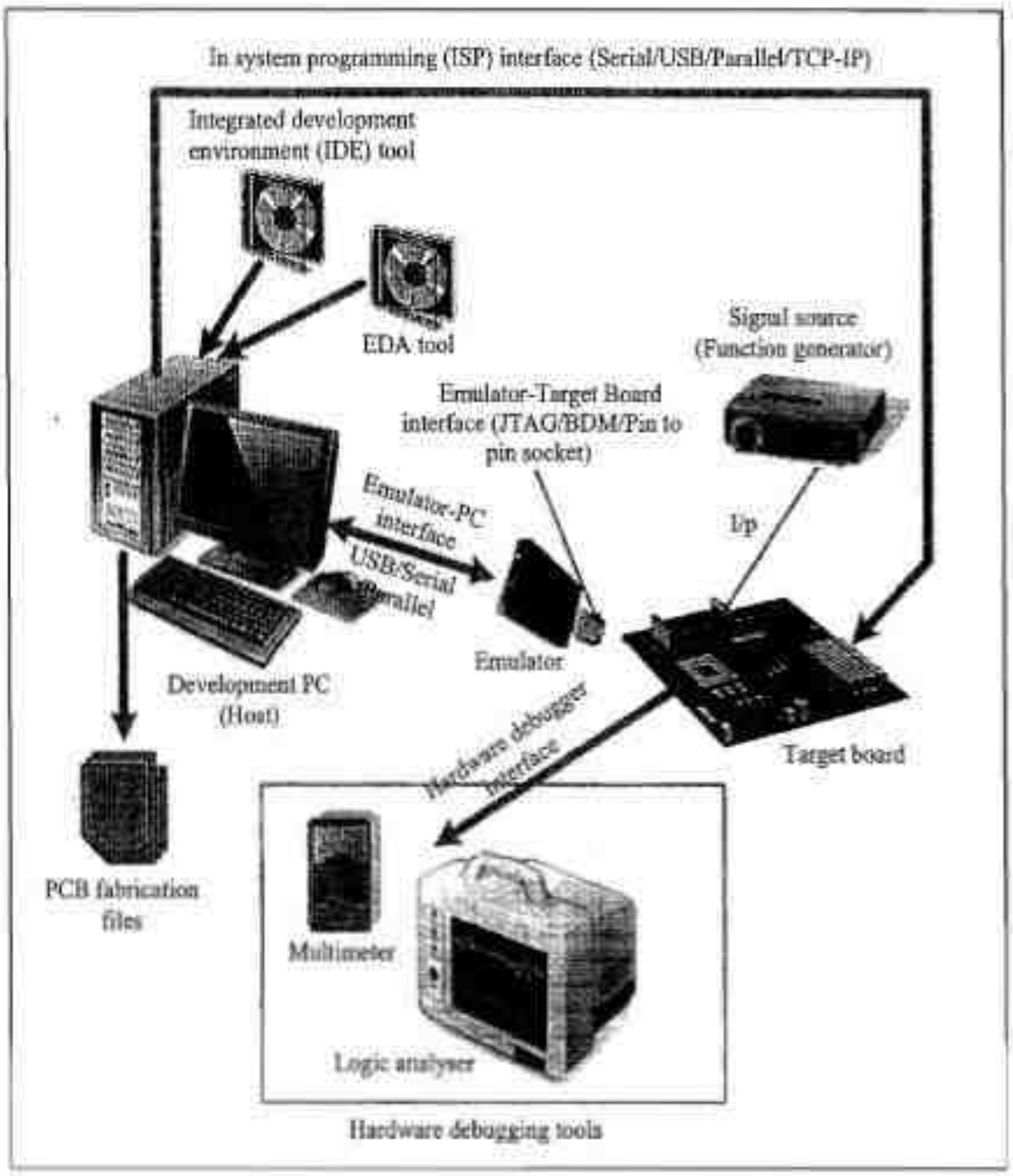
- It’s a technique used by firmware running on the target device for modifying a selected portion of the code memory
- It modifies the program code memory under the control of embedded application including updating calibration data, look up tables etc

Use of factory Programmed Chip

- Here embed the firmware into the target processor/controller memory at the time of chip fabrication itself
- Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory
- It reduces the product development time.

Firmware Loading for OS based devices

- Its programmed using ISP technique
- OS based system contain a special piece of code called *boot loader* program which takes control of the OS and application firmware embedding and copying of the OS images to the RAM of the system for execution
- Boot loader contains necessary driver initialization implementation for initializing the support interface like UART,TCP/IP etc
- E.g. Load from FLASH ROM, Load from network ,Load through UART etc


In system programming (ISP) interface (Serial/USB/Parallel/TCP-IP)
Integrated development environment (IDE) tool
EDA tool
Signal source (Function generator)
Emulator-Target Board interface (JTAG/BDM/Pin to pin socket)
Emulator-PC interface USB/Serial Parallel
I/p
Emulator
Development PC (Host)
Target board
Hardware debugger interface
PCB fabrication files
Multimeter
Logic analyser
Hardware debugging tools

The Embedded System Development Environment

EMBEDDED SYSTEM DEVELOPMENT ENVIRONMENT

•The most important characteristic of E.S is the cross-platform development technique.

•The primary components in the development environment are the host system, the target system and many connectivity solutions between the host and the target E.S.

•The development tools offered by the host system are the cross complier, linker and source-level debugger.

•The target embedded system offers a dynamic loader, link loader, a monitor and a debug agent.

•Set of connections are required between the source computer and the target system.

•These connections can be used for transmitting debugger information between the host debugger and the target debug agent.

IDE

- In E.S, IDE stands for an integrated environment for developing and debugging the target processor specific embedded firmware
- An IDE is also known as integrated design environment or integrated debugging environment.
- IDE is a software package which bundles a “Text Editor”, “Cross-compiler”, ”Linker” and a “Debugger”
- IDE is a software application that provides facilities to computer programmers for software development. IDEs can either command line based or GUI based

IDE Components

1.Text Editor or Source code editor

2.A compiler and an

interpreter

3.Build automation

tools 4.Debugger

5.Simulators

6.Emulators and logic analyzer

E.g. Turbo C/C++,Microsoft visual c++ etc

Keil uVersion3 IDE for 8051

- To start with IDE,execute the Keil uversion3 from desktop.ie similar to microsoft visual studio IDE.

- The µVision IDE combines project management, run-time environment, build facilities, source code editing, and program debugging in a single powerful environment.
- µVision is easy-to-use and accelerates your embedded software development.
- µVision supports multiple screens and allows you to create individual window layouts anywhere on the visual surface.

Step 1:`To start with IDE,execute the Keil uversion3 from desktop

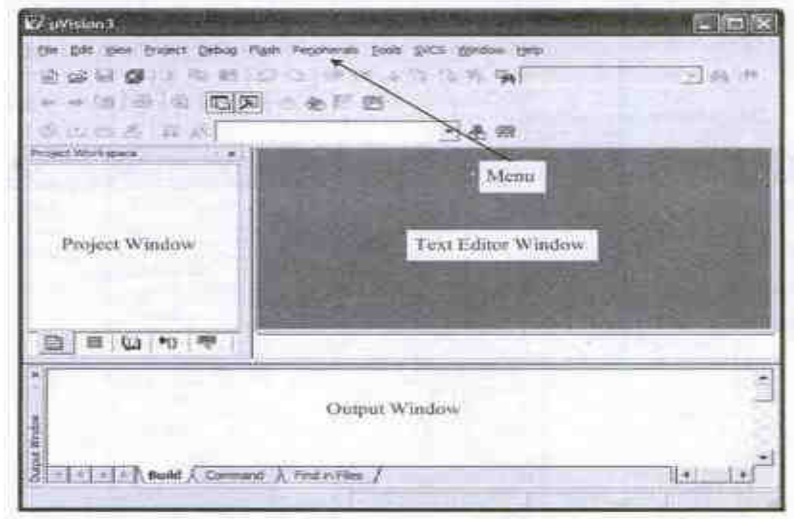
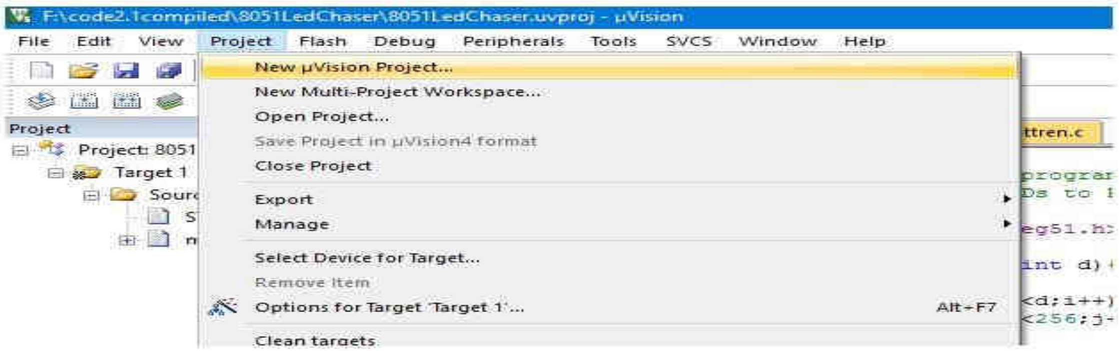


Fig. 13.2 Keil µVision3 Integrated Development Environment (IDE)

Step2: create a new project

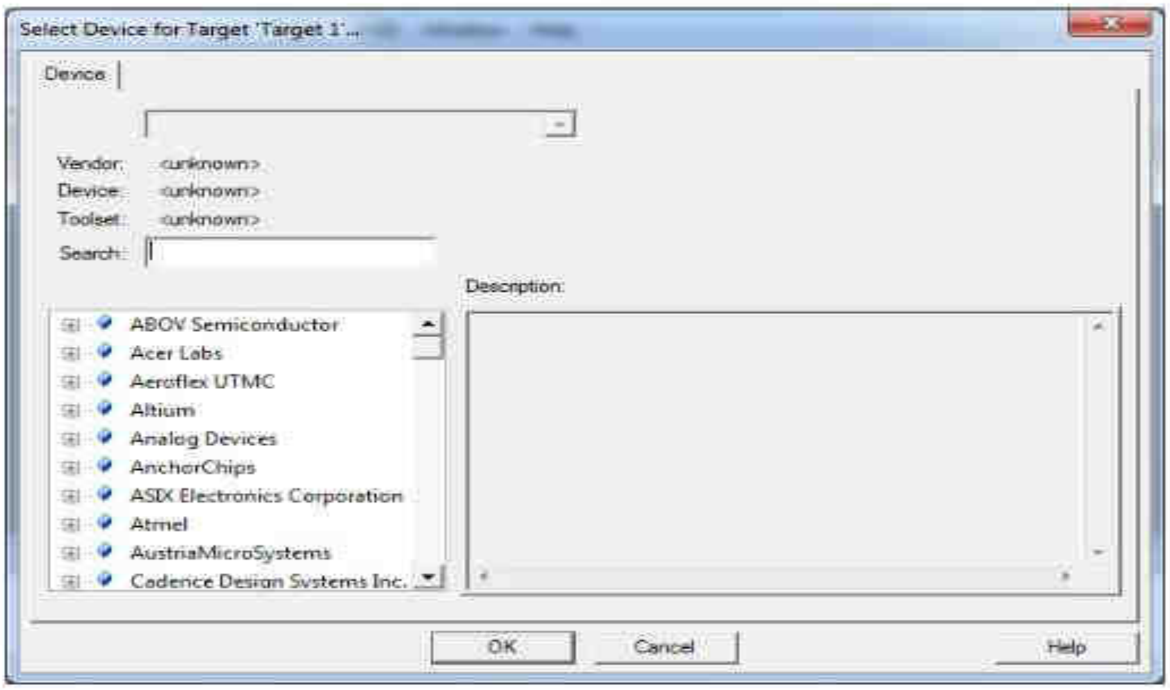
Creating Project in keil :

Select the project- > new uVision project


F:\code2.1compiled\8051LedChaser\8051LedChaser.uvproj - µVision
File
Edit
View
Project
Flash
Debug
Peripherals
Tools
SVCS
Window
Help
New µVision Project...
New Multi-Project Workspace...
Open Project...
Save Project in µVision4 format
Close Project
Export
Manage
Select Device for Target...
Remove Item
Options for Target 'Target 1'...
Alt+F7
Clean targets
Project
Project: 8051
Target 1

Once you select 'New uVision Project',it will prompt for a project name. Provide a suitable name for your project

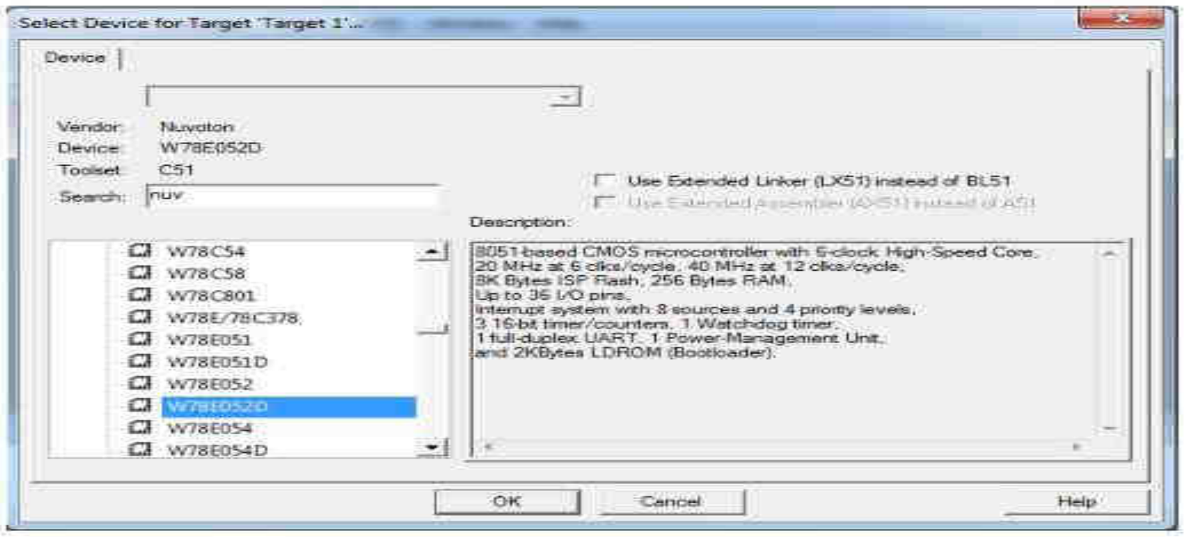
Step3:select CPU vendor,device and toolkit for keil IDE



Vendor: <unknown>
Device: <unknown>
Toolset: <unknown>
Search:

ABOV Semiconductor
Acer Labs
Aeroflex UTMC
Altium
Analog Devices
AnchorChips
ASIX Electronics Corporation
Atmel
AustriaMicroSystems
Cadence Design Systems Inc.

Here i am using **W78E052D from Nuvoton** for my project


Select Device for Target 'Target 1'...

Vendor: Nuvoton
Device: W78E052D
Toolset: C51
Search: nuv

Use Extended Linker (LX51) instead of BL51
Use Extended Assembler (AX51) instead of A51

Description:

8051-based CMOS microcontroller with 6-clock High-Speed Core,
20 MHz at 6 clks/cycle, 40 MHz at 12 clks/cycle,
8K Bytes ISP Flash, 256 Bytes RAM,
Up to 36 I/O pins,
Interrupt system with 8 sources and 4 priority levels,
3 16-bit timer/counters, 1 Watchdog timer,
1 full-duplex UART, 1 Power-Management Unit,
and 2KBytes LDROM (Bootloader).

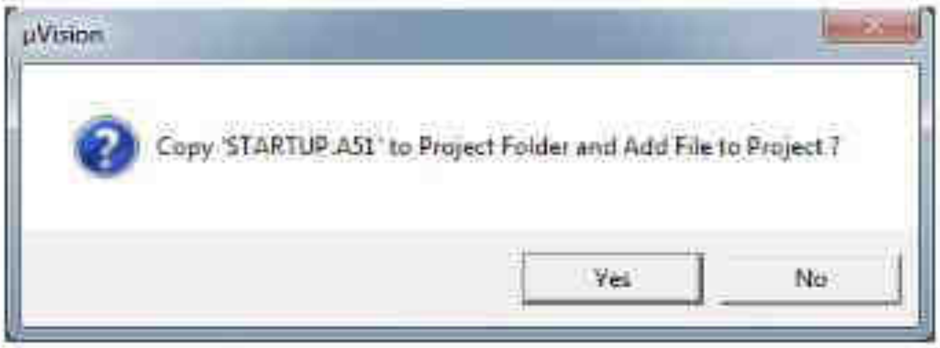
On selecting the particular microcontroller the Keil IDE also displays the features of the selected microcontroller on its left pane .You can Click *OK* to confirm your choice.

Keil has support for a wide variety of 8051 derivatives on its IDE.The 8051 derivatives are organised according to their manufacturer's.

For eg : Lets assume that you are developing code for ATMEL AT89S52 ,you can click on the ATMEL link on the bottom left pane and then browse for your choosen microcontroller here AT89S52.Alternatively you can also use the Search box on top to search for your particular part number.

Step 4: After selecting your 8051 derivative.

Asking to copy STARTUP.A51Click ' Yes '



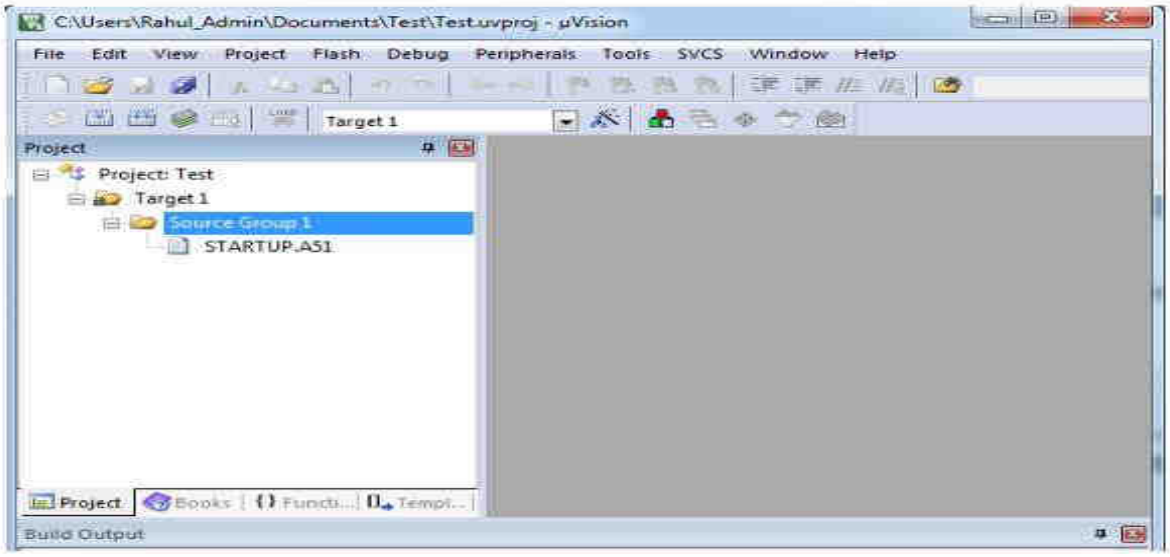
Copy 'STARTUP.A51' to Project Folder and Add File to Project ?

Click ' **Yes** '

Now your **Project pane** on the Kiel IDE would look something like this (below image)

Step5:

Now your **Project pane** on the Kiel IDE would look something like this (below image)



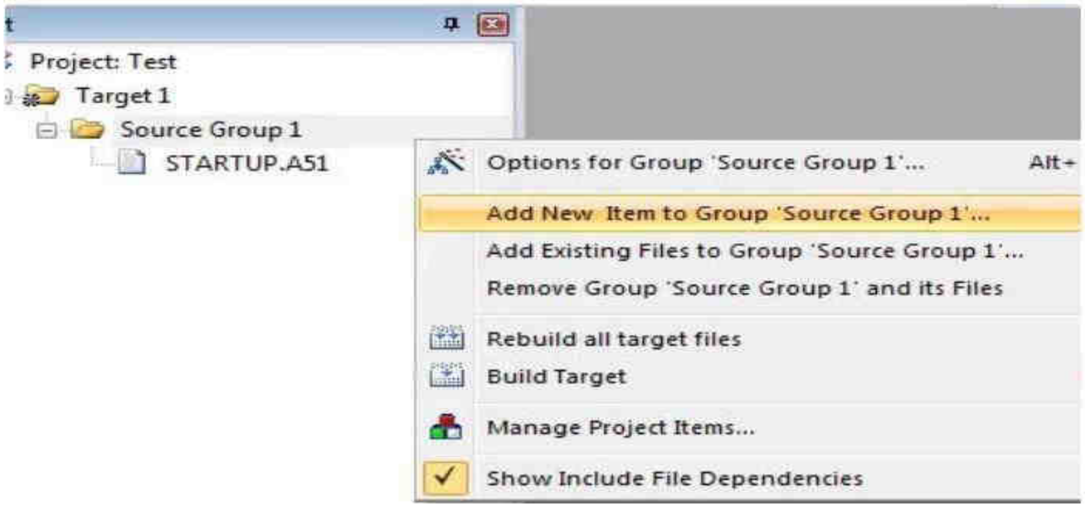
Now you can add C files to you Project.

Here Target 1 Is automatically generated under Files section of project window

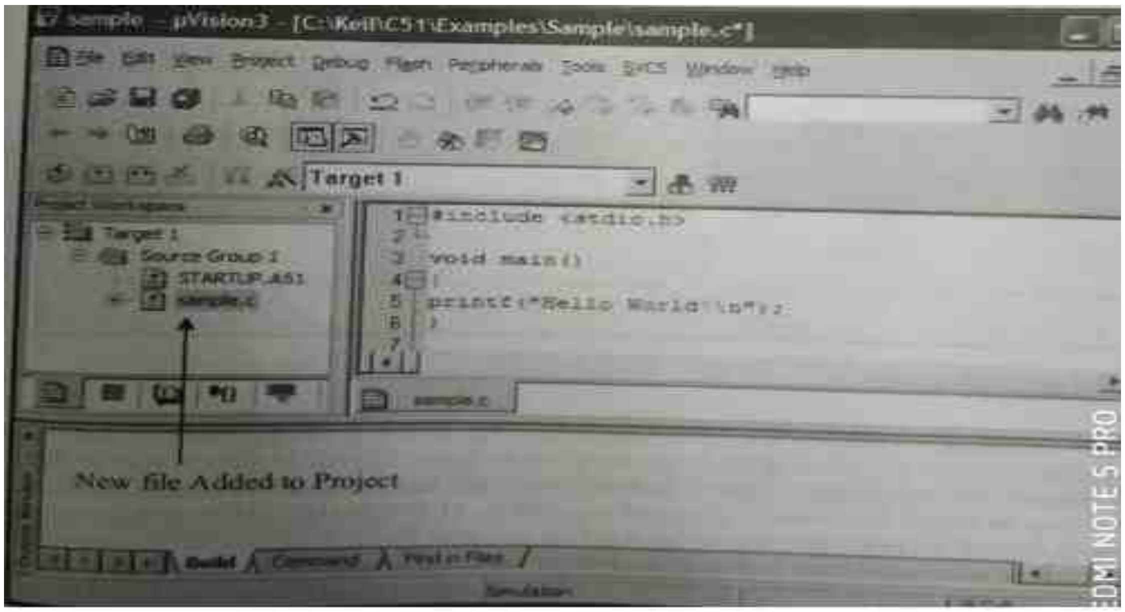
This contain source group with the name 'Source Group1'and 'start up' file is kept under this.

Step6: adding C file to the project

Step6: adding C file to the project


Project: Test
Target 1
Source Group 1
STARTUP.A51
Options for Group 'Source Group 1'... Alt+
Add New Item to Group 'Source Group 1'...
Add Existing Files to Group 'Source Group 1'...
Remove Group 'Source Group 1' and its Files
Rebuild all target files
Build Target
Manage Project Items...
Show Include File Dependencies

Step 7: Here I am adding Sample .C to my project

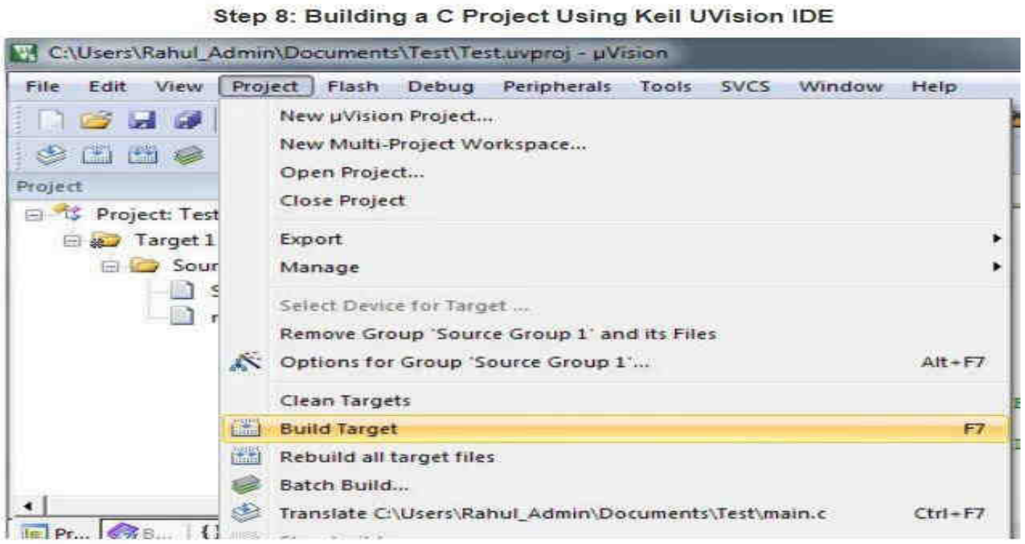

sample - µVision3 - [C:\Keil\C51\Examples\Sample\sample.c*]
Target 1
Source Group 1
STARTUP.A51
sample.c
New file Added to Project

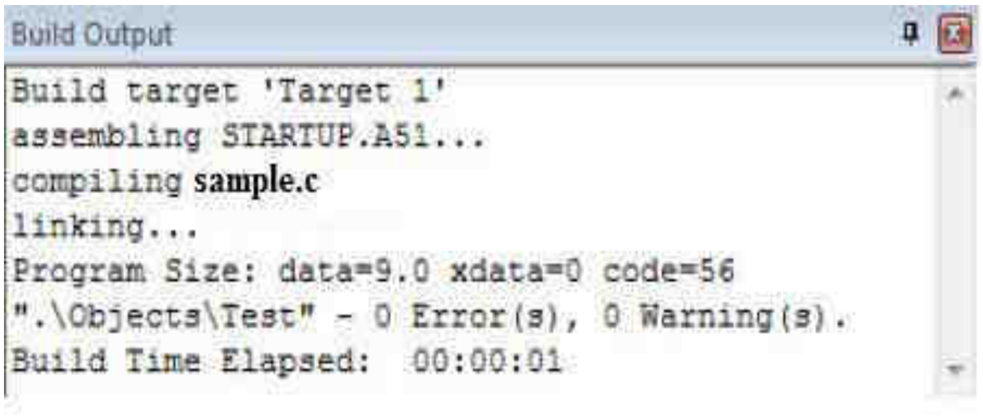
```
#include <stdio.h>

void main()
{
printf("Hello World!\n");
}
```

Step 8:

After you have typed out the above c program to your sample.c file,You can compile the C file by pressing **F7 key** or by going to ' **Project -> Build Target** ' on the IDE menu bar.


Step 8: Building a C Project Using Keil UVision IDE
C:\Users\Rahul_Admin\Documents\Test\Test.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target ...
Remove Group 'Source Group 1' and its Files
Options for Group 'Source Group 1'... Alt+F7
Clean Targets
Build Target F7
Rebuild all target files
Batch Build...
Translate C:\Users\Rahul_Admin\Documents\Test\main.c Ctrl+F7
Project
Project: Test
Target 1



```
Build target 'Target 1'
assembling STARTUP.A51...
compiling sample.c
linking...
Program Size: data=9.0 xdata=0 code=56
".\Objects\Test" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

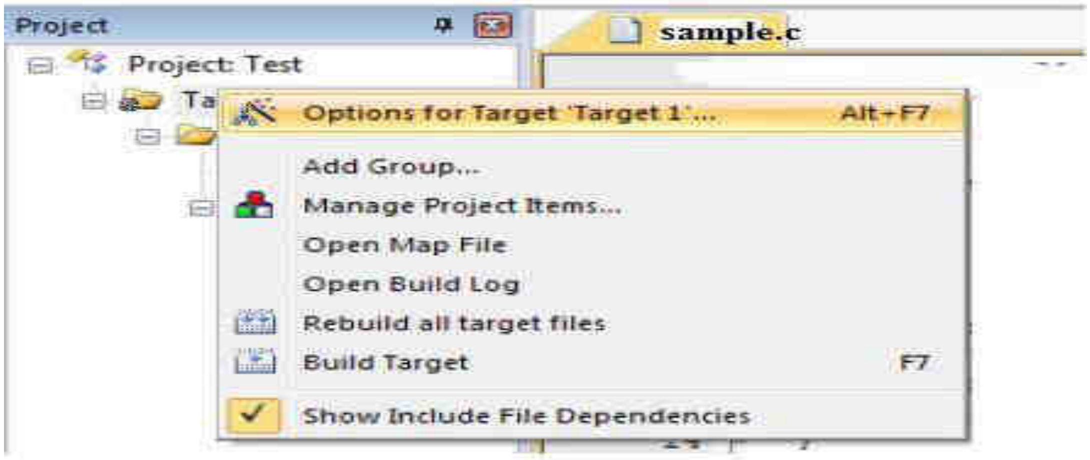
If there are no errors the code will compile and you can view the output on the ***Build Output*** pane

step 9:Generating 8051 HEX File using Kiel IDE

In the above example we have only compiled our sample .c file.Inorder to download the code into the 8051 microcontroller we have to generate the corresponding hexcode .

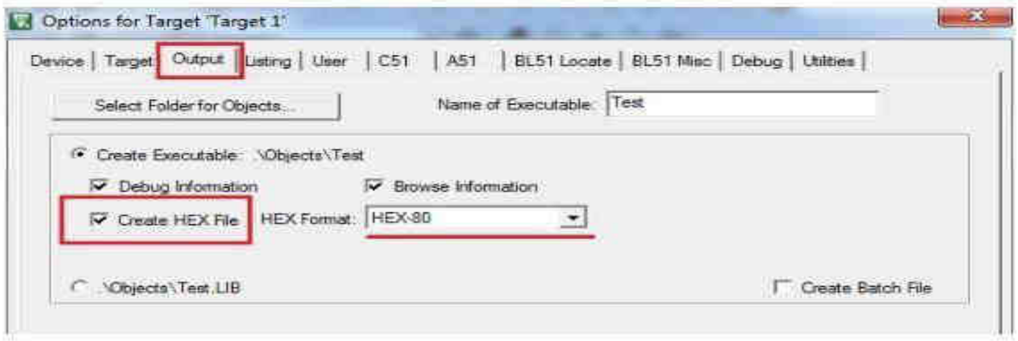
In Keil uVision IDE you can generate hexfile for your 8051 derivative by,

Right Clicking on the ***' Target 1 '*** Folder and Selecting ***Options for Target 'Target1'....***



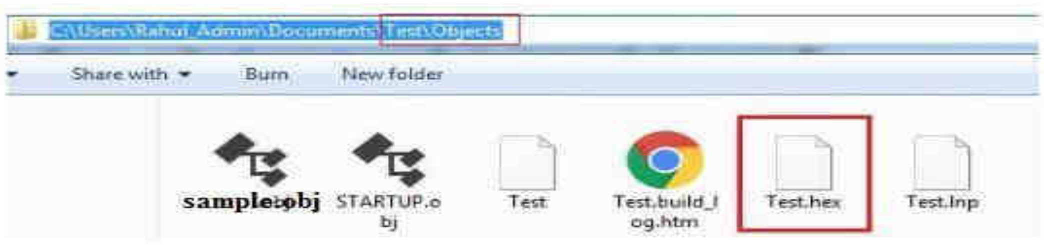
Then on the ***Options for Target ' Target 1'*** Dialog ,

Select the ***Output*** tab and check the **Create Hex File** option and Press ***OK***.

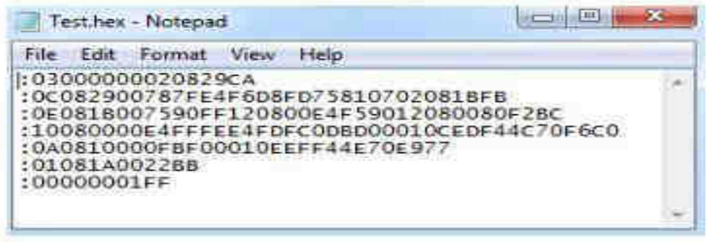


Now rebuild your project by pressing F7.

Kiel IDE would generate a hex file with same name (here Test.hex) as your project in the ***Objects folder*** .



You can also open the Test.hex file with notepad to view the contents.



```
:03000000020829CA
:0C082900787FE4F6D8FD75810702081BFB
:0E081800759 0FF120800E4F59012080080F2BC
:10080000E4FFFEE4FDFC0DBD00010CEDF44C70F6C0
:0A0810000FBF00010EEFF44E70E977
:01081A00022BB
:00000001FF
```

Cross Compilation

- Cross compilation is the process of converting a source code written in high level language to a target processor/controller understandable machine code
- The conversion of the code is done by software running on a processor/controller which is different from the target processor.
- The software performing this operation is referred as the Cross-compiler
- In other words cross-compilation the process of cross platform software/firmware development.
- A cross complier is a compiler that runs on one type of processor architecture but produces object code for a different type of processor architecture.

Cross Compiler-Advantages

- By using cross compliers we can not only develop complex E.S , but reliability can be improved and maintenance is easy.
- Knowledge of the processor instruction set is not required.
- Register allocation and addressing mode details are managed by the compiler.
- The ability to combine variable selection with specific operations improves program readability.

- Keywords and operational functions that more nearly resemble the human thought process can be changed.
- Program development and debugging time will be dramatically reduced when compared to assembly language programming
- The library files that are supplied provide may standard routines that may be incorporated into our application.
- Existing routine can be reused in new programs by utilizing the modular programming techniques available with C.
- The C language is very portable and very popular.

''////

TYPES OF FILE GENERATED ON CROSS COMPILATION

Following are some of the files generated upon cross compilation:

List file .lst

Hex file .hex

Preprocessor output file

Map file .map

Obj file .obj

1. **List File(.lst):-**

Listing file is generated during the cross-compilation process.

It contains an information about the cross compilation process like cross compiler details, formatted source text('C' code), assembly code generated from the source file, symbol tables, errors and warnings detected during the cross-compilation process.

The list file contain the following sections:

Page Header

It indicates the compiler version name, source file name, Date, Page No.

Example: C51 COMPILER V8.02 SAMPLE 05/23/2006 11:12:58 PAGE 1

Command Line

It represents the entire command line that was used for invoking the compiler.

C51 COMPILER V8.02, COMPILATION OF MODULE SAMPLE OBJECT MODULE PLACED IN sample.obj

COMPILER INVOKED BY: C:\Keil\C51\BIN\C51.EXE sample.c BROWSE DEBUG OBJECTTEXTEND CODE

LISTINCLUDE SYMBOLS

Source Code

It contains source code along with line numbers

Line level Source

```
1       //Sample.c for printing Hello World!

//Written by xyz

#include<stdio.h>

//Body part starts


//Body part end

void main()

{

printf("Hello World");

}

//Header part ends
```

Assembly listing

It contains the asembly code generated by compiler for even given 'C' code.

ASSEMBLY LISTING OF GENERATED OBJECT CODE;

FUNCTION main(BEGIN)

Symbol listing

It contains symbolic information about the various symbols present in the cross compiled source file.

FUNCTION main(BEGIN)

Eg: NAME, TYPE, SFR, SIZE.

;SOURCE LINE #5

Module Information

The module information provides the size of initialized and un-initialized memory areas defined by the source file.

Module Information		Static Overlayable
Code Size	9	-------------
Constant size	14	-------------
Bit size	-------	-------------

END OF MODULE INFORMATION

Warnings and Errors

Warnings and Errors section of list file records the errors encountered or any statement that may create issues in application(Warnings), during cross compilation.

- ie:- C51 COMPILATION COMPLETE, 0WARNING(S), 0 ERROR(S).

2. **Preprocessor Output File**

It contains preprocessor output for preprocessor instructions used in the source file.

This file is used for verifying the operation of Macros and preprocessor directive.

3. **Object File(.OBJ File)**

An **object file** is a file containing object code, meaning relocatable format machine code thatis usually not directly executable. Object files are produced by an assembler, compiler, or other language translator, and used as input to the linker, which in turn typically generates an executable or library by combining parts of object files. There are various formats for object files, and the same object code can be packaged in different object files. In addition to the object code itself, object files may contain metadata used for linking or debugging, including: information to resolve symbolic cross-references between different modules, relocation information, stack unwinding information, comments, program symbols, debugging or profiling information.
An **object file format** is a computer file format used for the storage of object code and related data. There are many different object file formats; originally each type of computer had its own unique format, but with the advent of Unix and other portable operating systems, some formats, such as COFF and ELF have been defined and used on different kinds of systems. It is possible for the same file format to be used both as linker input and output, and thus as the library and executable file format.

The design and/or choice of an object file format is a key part of overall system design. It affects the performance of the linker and thus programmer turnaround while developing. If the format is used for executables, the design also affects the time programs take to begin running, and thus the responsiveness for users. Most object file formats are structured as blocks of data, each block containing a certain type of data (see Memory segmentation). These blocks can be paged in as needed by the virtual memory system, needing no further processing to be ready to use.Debugging information may either be an integral part of the object file format, as in COFF, or a semiindependent format which may be used with several object formats, such as stabs or DWARF.
Object files are usually divided into segments or sections, not to be confused with memory segmentation. Segments in different object files may be combined by the linker according to rules specified when the segments are defined. Conventions exist for segments shared

between object files; for instance, in DOS there are different memory models that specify the names of special segments and whether or not they may be combined.[2] Types of data supported by typical object file formats:

• Header (descriptive and control information)

• Text segment (executable code)

• Data segment (static data)

• BSS segment (uninitialized static data)

• External definitions and references for linking

• Relocation information

• Dynamic linking information

• Debugging information

4. **Map File(.MAP)**

Also called as Linker List file. Map file contains information about the link/locate process and is composed of a number of sections described below:

Page Header

Each MAP file contains a header which indicates the linker version number, date, time and page number.

Command Line

Represents the entire command line that was used for invoking the linker.

CPU Details

It contains details about the target CPU and its memory model which includes information on internal data memory, external data memory, paged data memory, etc.

Input Modules

It includes the names of all the object files, library files and other files that are included in the linking process.

Memory Map

I
t lists the starting address, length, relocation type and name of each segment in the program

Symbol Table

It contains the name, value and type for all symbols from different input modules.

Inter Module Cross Reference

It includes the section name, memory type and module names in which it is defined and all modules where it is accessed.

Ex.

NAME……………………USAGE……………………..

MODULE NAMES ?CCCASE…………………CODE;… ....?

C?CCASE PRINTF ?C?CLDOPTR……………CODE;… ....?C?

CLDOPTR PRINTF ?C?CSTPTR………………CODE;…....................... ?C

?CSTPTR PRINTF

Program Size

It contains the size of various memory areas, constants and code space for the entire application Ex. Program Size: data=80.1 xdata=0 code 2000

Warnings and Errors

It contains the warnings and errors that are generated while linking a program. It is used in debugging link errors

5. **HEX FILE (.hex file)**

It is a binary executable file created from the source code.

The file created by linker/locater is converted into processor understandable binary code.

The tool used for converting an object file into a hex file is known as object to Hex converter.

Hex file have specific format and it varies for different processor and controller. Two commonly used hex file format are:

Intel Hex

Motorola Hex.

Both Intel and Motorola hex file format represent data in the form of ASCII codes.

The **Intel HEX file** is an ASCII text file with lines of text that follow the Intel HEX file format. Each line in an Intel HEX file contains one HEX record. These records are made up of hexadecimal numbers that represent machine language code and/or constant data. Intel HEX files are often used to transfer the program and data that would be stored in a ROM or EPROM. Most EPROM programmers or emulators can use Intel HEX files.

Record Format

An Intel HEX file is composed of any number of HEX records. Each record is made up of five fields that are arranged in the following format:

```
:llaaaatt[dd...]cc
```

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits-which make up a byte-as described below:

- **:** is the colon that starts every Intel HEX record.
- ***ll*** is the record-length field that represents the number of data bytes (**dd**) in the record.
- ***aaaa*** is the address field that represents the starting address for subsequent data in the record.
- ***tt*** is the field that represents the HEX record type, which may be one of the following:
  00 - data record
  01 - end-of-file record
  02 - extended segment address record
  04 - extended linear address record
  05 - start linear address record (MDK-ARM only)
- ***dd*** is a data field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the **ll** field.
- ***cc*** is the checksum field that represents the checksum of the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

Data Records

The Intel HEX file is made up of any number of data records that are terminated with a carriage return and a linefeed. Data records appear as follows:

```
:10246200464C5549442050524F46494C4500464C33
```

This record is decoded as follows:

```
:10246200464C5549442050524F46494C4500464C33
||||||||||                       CC->Checksum
||||||||DD->Data
||||||TT->Record Type
|||AAAA->Address
|LL->Record Length
:->Colon
```

where:

- **10** is the number of data bytes in the record.
- **2462** is the address where the data are to be located in memory.
- **00** is the record type 00 (a data record).
- **464C...464C** is the data.
- **33** is the checksum of the record.

Disassembler And Decompiler

- Disassembler is a utility program.
- Converts machine codes into target specific assembly instructions
- Process of converting machine code to assembly code –disassembling
- Operation is complementary to assembling.
- Utility program for translating machine code to high level instructions – decompiler.
- Decompiler performs reverse operation of compiler.
- They are deployed in reverse engineering.
- Reverse engineering – process of revealing technology behind working of a product.
- Disassemblers/Decompilers help reverse engineering process by translating embedded firmware into Assembly/high level instructions.

- Employed in embedded product development to find out the secret behind proprietary products.
- Powerful tools for analyzing the presence of malicious codes.
- Available as either freeware tools or as commercial tools.
- They generate a source code which is nearly similar to the original code.

SIMULATORS

- Simulator is a software tool for simulating various functionality of the application software
- IDE provides simulator support
- Simulator simulates target hardware and firmware execution can be simulate using simulators

FEATURES OF SIMULATOR

- Purely software based
- Doesn't require a real target system
- Very primitive
- Lack of real time behavior

LIMITATIONS:

- Deviation from real behavior
- Lack of real timeliness

Advantage of simulator based debugging

- No need of target board
    - Purely software oriented , IDE simulates the target board
    - Since real hardware is not needed we can start immediately after the device interface and memory maps are finalized this saved development time
- Simulated I/O peripherals

    - It eliminates the need for connecting IO devices for debugging the firmware

- Simulates abnormal conditions

    - Can input any parameter as input during debugging hence can check for abnormal conditions easily.

EMULATOR

- It is a piece of hardware that exactly behaves like the real microcontroller chip with all its integrated functionality.
- It is the most powerful debugging of all.
- A microcontroller's functions are emulated in real-time and non-intrusively.
- An emulator is a piece of hardware that looks like a processor, has memory like a processor, and executes instructions like a processor but it is not a processor.
- The advantage is that we can probe points of the circuit that are not accessible inside a chip.
- It is a combination of hardware and software.

All emulators contain 3 essential function:

1. The emulator control logic, including emulation memory
2. The actual emulation device
3. A pin adapter that gives the emulator's target connector the same "package" and pin out as the microcontroller to be emulated

DEBUGGERS

- Debugging in embedded application is the process of diagnosing the firmware execution, monitoring the target processor's registers and memory while the firmware is running
- Debugging is classified into two namely Hardware debugging and firmware debugging.
- Hardware debugging deals with the monitoring of various bus signals and checking the status lines of the target hardware.

- Firmware debugging deals with examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design.
- Debugger is a special program used to find errors or bugs in other programs.
- A debugger allows a programmer to stop a program at any point and examine and change the values of the variables.
- A debugger or debugging tool is a computer program that is used to test and debug other programs.
- Some of the debuggers offer two modes of operation like full or partial simulation.
- A crash happens when the program cannot normally continue because of a programming bug.
- Ex- The program might have tried to use an instruction not available on the current version of the CPU to access unavailable or protected memory.
- When program crashes or reaches a preset condition the debugger shows the position in the original code if it is a source-level debugger or symbolic debugger.

MODULE -6

Networks – Distributed Embedded Architectures, Networks for embedded systems, Network based design, Internet enabled systems. Embedded Product Development Life Cycle – Description – Objectives -Phases – Approaches1. Recent Trends in Embedded Computing.

Distributed Embedded Systems

1.INTRODUCTION

In a distributed embedded system, several processing elements (PEs) (either microprocessors or ASICs) are connected by a network that allows them to communicate. The application is distributed over the PEs, and some of the work is done at each node in the network.

1.1 Reasons to build network-based embedded systems.

1.When the processing tasks are physically distributed, it may be necessary to put some of the computing power near where the events occur.

2.Data reduction is another important reason for distributed processing. It may be possible to perform some initial signal processing on captured data to reduce its volume.

3.Modularity is another motivation for network-based design. For instance, when a large system is assembled out of existing components, those components may use a network port as a clean interface that does not interfere with the internal operation of the component in ways that using the microprocessor bus would.

4.A distributed system can also be easier to debug the microprocessors in one part of the network can be used to probe components in another part of the network. Finally, in some cases, networks are used to build fault tolerance into systems.

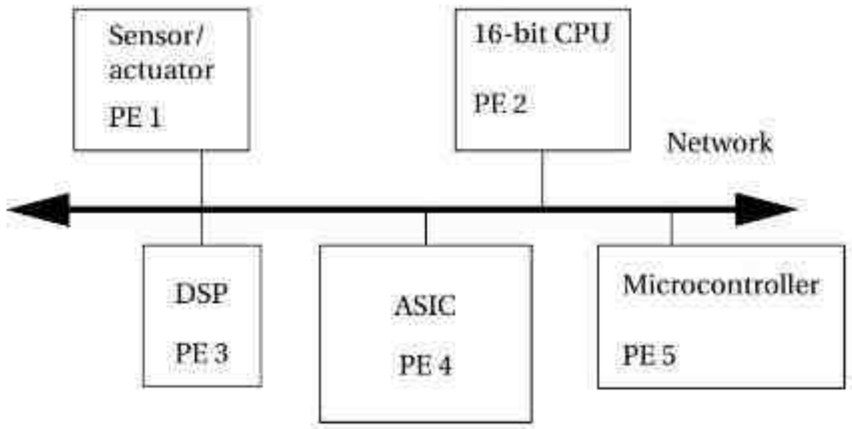
5.Distributed embedded system design is another example of hardware/software co-design, since we must design the network topology as well as the software running on the network.

2.DISTRIBUTED EMBEDDED ARCHITECTURE

A distributed embedded system can be organized in many different ways, but its basic units are the PE and the network as illustrated in Figure 8.1.

A PE may be an instruction set processor such as a DSP, CPU, or microcontroller, as well as a nonprogrammable unit such as the ASICs used to implement PE 4. An I/O device such as PE 1 (which we call here a sensor or actuator, depending on

whether it provides input or output) may also be a PE, so long as it can speak the network protocol to communicate with
other PEs. The network in this case is a bus, but other network topologies are also possible. It is also possible that the system can use more than one network, such as when relatively independent functions require relatively little communication among them. We often refer to the connection between PEs provided by the network as a communication link. The system of PEs and networks forms the hardware platform on which the application runs.



.

Fig 8.1 An example of a distributed embedded system.

2.1 Why Distributed?

In some cases,distributed systems are necessary because the devices that the PEs communicate with are physically separated. If the deadlines for processing the data are short, it may be more cost-effective to put the PEs where the data are located rather than build a higher-speed network to carry the data to a distant, fast PE.

An important advantage of a distributed system with several CPUs is that one part of the system can be used to help diagnose problems in another part.Whether you are debugging a prototype or diagnosing a problem in the field, isolating the error to one part of the system can be difficult when everything is done on a single CPU. If you have several CPUs in the system,you can use one to generate inputs for another and to watch its output.

2.2 Network Abstractions

Networks are complex systems. Ideally, they provide high-level services while hiding many of the details of data transmission from the other components in the system. In order to help understand (and design) networks, the International Standards Organization has developed a seven-layer model for networks known as Open Systems Interconnection (OSI) models

The seven layers of the OSI model, shown in Figure 8.2, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one or more layers because the higher layers may be totally missing or an intermediate layer may not be necessary. However, any data network should fit into the OSI model. The OSI layers from lowest to highest level of abstraction are described below.

■Physical: The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), electrical properties, basic functions of the electrical and physical components, and the basic procedures for exchanging bits.

■Data link: The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops, but only within a single hop.

■Network: This layer defines the basic end-to-end data transmission service.The network layer is particularly important in multihop networks.

■ Transport: The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links.This layer may also try to optimize network resource utilization.

■ Session: A session provides mechanisms for controlling the interaction of end-user services across a network, such as data grouping and checkpointing.

■ Presentation: This layer defines data exchange formats and provides transformation utilities to application programs.

■Application: The application layer provides the application interface between the network and end users.

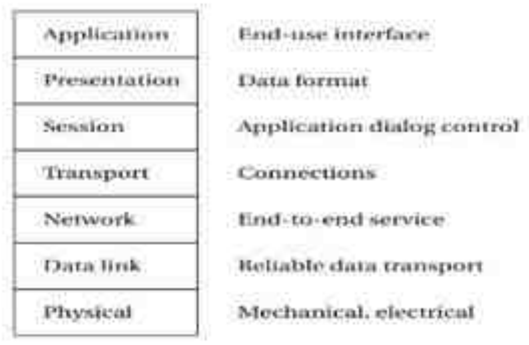


fig 8.2 The OSI Model layers

<u>2.3 Hardware and Software Architectures</u>

Distributed embedded systems can be organized in many different ways depending upon the needs of the application and cost constraints. One good way to understand possible architectures is to consider the different types of interconnection networks that can be used.

A point-to-point link establishes a connection between exactly two PEs. Point-to-point links are simple to design precisely because they deal with only two components.We do not have to worry about other PEs interfering with communication on the link

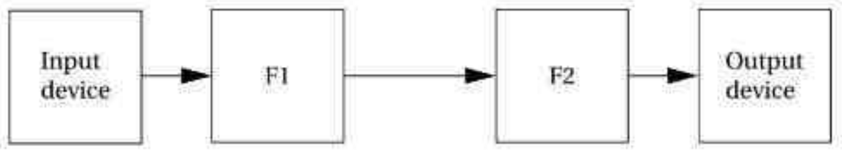


FIGURE 8.3

A signal processing system built from print-to-point links.

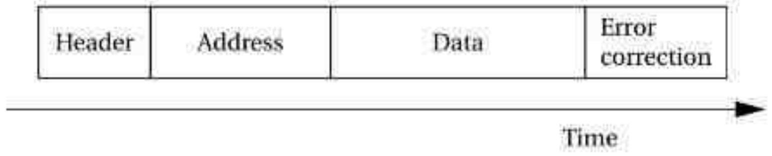


FIGURE 8.4

Format of a typical message on a bus.

Figure 8.3 shows a simple example of a distributed embedded system built from point-to-point links. The input signal is sampled by the input device and passed to the first digital filter, F1, over a point-to-point link. The results of that filter are sent through a second point-
to-point link to filter F2. The results in turn are sent to the output device over a third point-to-point link. A digital filtering system requires that its outputs arrive at strict intervals, which means that the filters must process their inputs in a timely fashion. Using point-to-point connections allows both F1 and F2 to receive a new sample

and send a new output at the same time without worrying about collisions on the communications network.

A bus is a more general form of network since it allows multiple devices to be connected to it. Like a microprocessor bus, PEs connected to the bus have addresses. Communications on the bus generally take the form of packets as illustrated in Figure 8.4. A packet contains an address for the destination and the data to be delivered. It frequently includes error detection/correction information such as parity. It also may include bits that serve to signal to other PEs that the bus is in use, such as the header shown in the figure. The data to be transmitted from one PE to another may not fit exactly into the size of the data payload on the packet. It is the responsibility of the transmitting PE to divide its data into packets; the receiving PE must of course reassemble the complete data message from the packets.

Distributed system buses must be arbitrated to control simultaneous access,just as with microprocessor buses. Arbitration scheme types are summarized below.

■Fixed-priority arbitration always gives priority to competing devices in the same way. If a high-priority and a low-priority device both have long data transmissions ready at the same time, it is quite possible that the low-priority device will not be able to transmit anything until the high-priority device has sent all its data packets.

■ Fair arbitration schemes make sure that no device is starved. Round-robin arbitration is the most commonly used of the fair arbitration schemes. The PCI bus requires that the arbitration scheme used on the bus must be fair, although it does not specify a particular arbitration scheme. Most implementations of PCI use round-robin arbitration. A bus has limited available bandwidth. Since all devices connect to the bus,communications can interfere with each other. Other network topologies can be used to reduce communication conflicts.At the opposite end of the generality spectrum from the bus is the crossbar network shown in Figure 8.5.

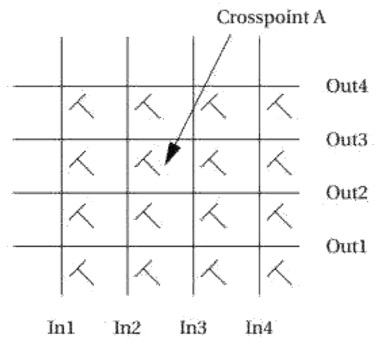


FIGURE 8.5

A crossbar network.

A crossbar not only allows any input to be connected to any output, it also allows all combinations of input/output connections to be made. Thus, for example, we can simultaneouslyconnect in1 to out4, in2 to out3, in3 to out2, and in4 to out1 or any other combinations of inputs. (Multicast connections can also be made from one input to several outputs.) A crosspoint is a switch that connects an input to an output. To connect an input to an output, we activate the crosspoint at the intersection between the corresponding input and output lines in the crossbar. For example, to connect in2 and out3 in the figure, we would activate crossbar A as shown. The major drawback of the crossbar network is expense: The size of the network grows as the square of the number of inputs (assuming the numbers of inputs and outputs are equal).

Many other networks have been designed that provide varying amounts of parallel communication at varying hardware costs. Figure 8.6 shows an example multistage network. The crossbar of Figure 8.5 is a direct network in which messages go from source to destination without going through any memory element. Multistage networks have intermediate routing nodes to guide the data packets.

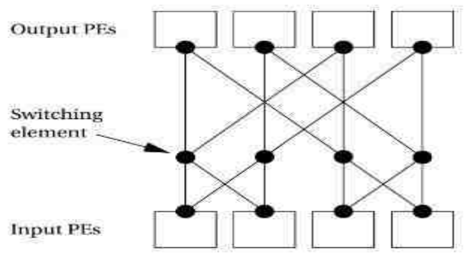


FIGURE 8.6
A multistage network.

2.4 Message Passing Programming

Distributed embedded systems do not have shared memory,so they must communicate by passing messages.We will refer to a message as the natural communication unit of an algorithm; in general, a message must be broken up into packets to be sent on the network. A procedural interface for sending a packet might look like the following:

```
send_packet(address,data);
```

The routine should return a value to indicate whether the message was sent successfully if the network includes a handshaking protocol. If the message to be sent is longer than a packet,it must be broken up into packet-size data segments as follows:

```
for (i = 0; i < message.length;i=i+ PACKET_SIZE)

send_packet(address,&message.data[i]);
```

The above code uses a loop to break up an arbitrary-length message into packet-size chunks. However, clever system design may be able to recast the message to take advantage of the packet format. For example, clever encoding may reduce the length of the message enough so that it fits into a single packet. On the other hand, if the message is shorter than a packet or not an even multiple of the packet data size, some extra information may be packed into the remaining bits of a packet.
Reception of a packet will probably be implemented with interrupts. The simplest procedural interface will simply check to see whether a received message is waiting in a buffer. In a more complex RTOS-based system, reception of a packetmay enable a process for execution.

Network protocols may encourage a **data-push design** style for the system built around the network. In a single-CPU environment,a program typically initiates a read whenever it wants data. In many networked systems, nodes send values out without any request from the intended user of the system. Data-push programming makes sense for periodic data—if the data will always be used at regular intervals, we can reduce data traffic on the network by automatically sending it when it is needed.

3.NETWORKS FOR EMBEDDED SYSTEMS

Networks for embedded computing span a broad range of requirements; many of those requirements are very different from those for general-purpose networks. Some networks are

used in safety-critical applications, such as automotive control. Some networks, such as those used in consumer electronics systems, must be very inexpensive. Other networks,such as industrial control networks,must be extremely rugged and reliable.
IO devices communicate with processor through an IO bus ,which is separate from memory bus that the processor used to communicate with the memory system. Embedded system communicate internally on the same IC or systems with very short and long distances and can be networked by using following bus each functioning according to specific protocols.

1. Using serial IO bus allows a computer or controller or embedded system to interface network with a wide range of IO devices without having to implement a specific interface for each IO device.when the IO devices in the distributed embedded system are networked at long distances of 25cm and above all can communicate through a common serial bus.A serial bus has very few lines
2. Using parellel IO bus allows a computer or controller or embedded system to interface with a number of internal systems at very short distances without having to implement a specific interface for each IO device
3. Using the internet or intranet a computer, controller or embedded system IO device can interface globally and can network with othersystems or computers and a wide range of devices in the distributed system.

4. Using wireless protocols allows a handheld computer ,controller or embedded system IO device to interface and network with a number of handheld system IO devices at short devices upto 100 m using a wireless personel area(WPAN) protocol without having to implement a specific wireless interface for each IO device

Embedded systems are distributed and networked using a serial or parallel bus or wireless protocol software and appropriate hardware.Several interconnect networks have been developed especially for distributed embedded computing:

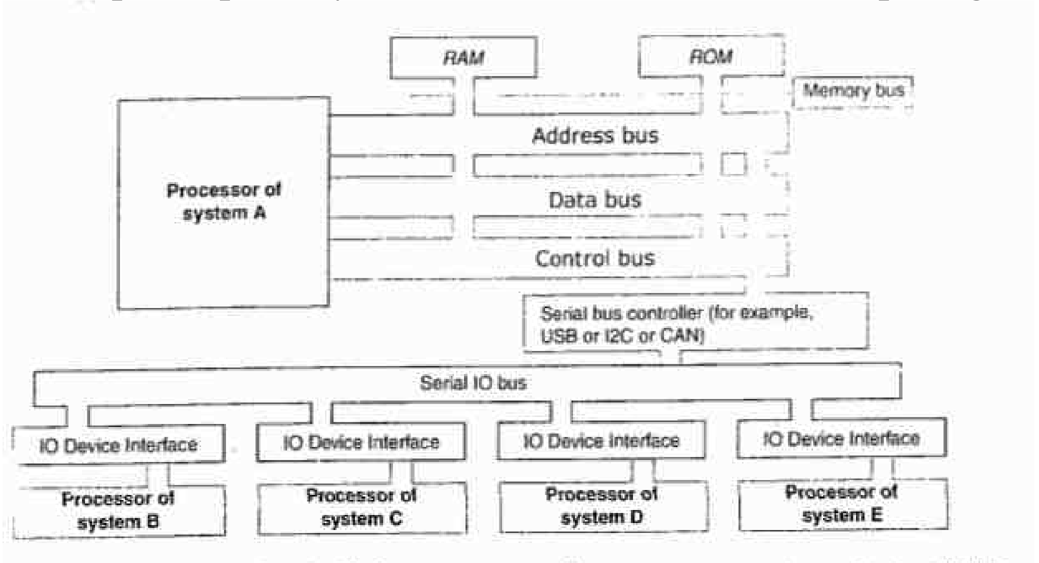


Fig. 3.9 A processor of embedded system connected to system memory bus and networked to other systems through a serial bus

3.1 Serial Bus Communication Protocols

The following describes most popular serial buses

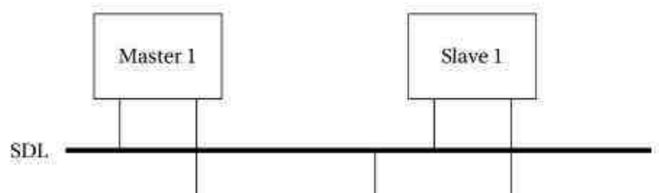
1. The I2 C bus is used in microcontroller-based systems.
2. The Controller Area Network (CAN) bus was developed for automotive electronics. It

   provides megabit rates and can handle large numbers of devices.

3. USB Bus
4. Firewire-IEEE 1394 bus standard
5. Advanced high speed buses
6. Ethernet and variations of standard Ethernet are used for a variety of control applications. In addition, many networks designed for general-purpose computing have been put to use in embedded applications as well.

3.1 .1 The I2C Bus

The ***I 2C bus*** is a well-known bus commonly used to link microcontrollers into systems. I 2C is designed to be low cost,easy to implement, and of moderate speed (up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus). As a result, it uses only two lines: the serial data line (SDL) for data and the serial clock line (SCL), which indicates when valid data are on the data line. Figure 8.7 shows the structure of a typical I2C bus system. Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus masters and the bus may have more than one master. Other nodes may act as slaves that only respond to requests from masters.



The I2C bus is designed as a multimaster bus—any one of several different devices may act as the master at various times. As a result, there is no global master to generate the clock signal on SCL. Instead, a master drives both SCL and SDL .when it is sending data.When the bus is idle, both SCL and SDL remain high.When two devices try to drive either SCL or SDL to different values, the open collector open drain circuitry prevents errors, but each master device must listen to the bus while transmitting to be sure that it is not interfering with another message—if the device receives a different value than it is trying to transmit, then it knows that it interfering with another message.

Every I2 C device has an address. The addresses of the devices are determined by the system designer, usually as part of the program for the I2 C driver. The addresses must of course be chosen so that no two devices in the system have the same address. A device address is 7 bits in the standard I2 C definition (the extended I2 C allows 10-bit addresses).

A bus transaction is initiated by a start signal and completed with an end signal as follows:

■ A start is signaled by leaving the SCL high and sending a 1 to 0 transition on SDL.

■ A stop is signaled by setting the SCL high and sending a 0 to 1 transition on SDL.

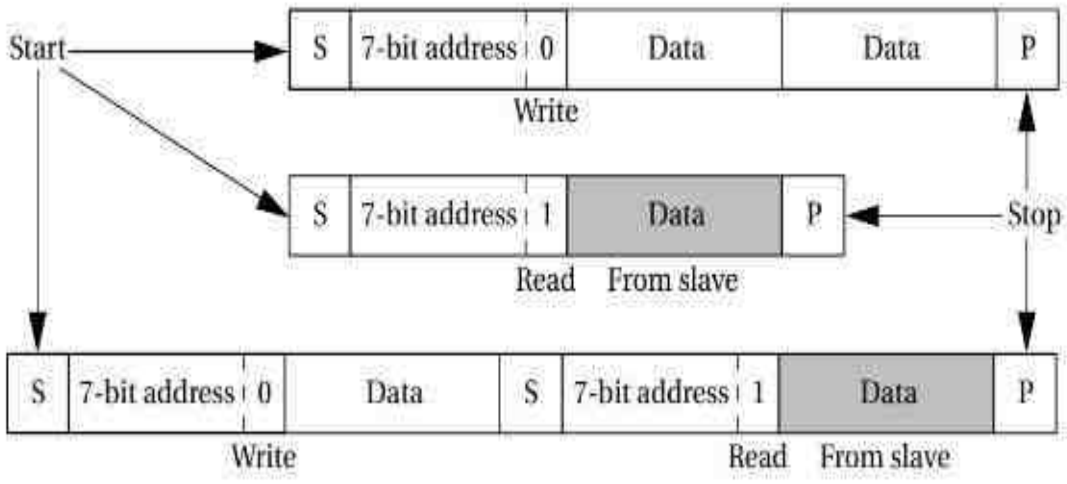


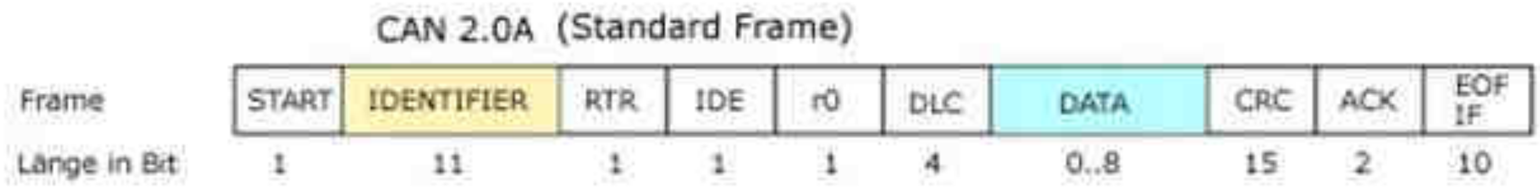
FIGURE 8.11

Typical bus transactions on the I^2C bus.

<u>3.1.2 CAN BUS</u>

- CAN or Controller Area Network or CAN-bus is an ISO standard computer network protocol and bus standard, designed for microcontrollers and devices to communicate with each other without a host computer.

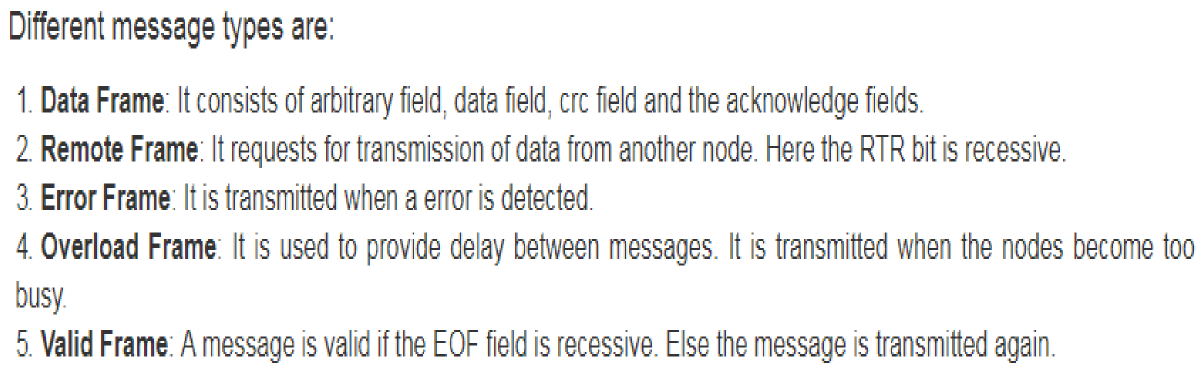
- Designed earlier for industrial networking but recently more adopted to automotive applications, CAN have gained widespread popularity for embedded control in the areas like industrial automation, automotives, mobile machines, medical, military and other harsh environment network applications.
- Development of the CAN-bus started originally in 1983 at Robert Bosch GmbH. The protocol was officially released in 1986. and the first CAN controller chips, produced by Intel and Philips, introduced in the market in the year of 1987.
- The CAN is a "broadcast" type of bus. That means there is no explicit address in the messages. All the nodes in the network are able to pick-up or receive all transmissions. There is no way to send a message to just a specific node. To be more specific, the messages transmitted from any node on a CAN bus does not contain addresses of either the transmitting node, or of any intended receiving node. Instead, an identifier that is unique throughout the network is used to label the content of the message. Each message carries a numeric value, which controls its priority on the bus, and may also serve as an identification of the contents of the message. And each of the receiving nodes performs an acceptance test or provides local filtering on the identifier to determine whether the message, and thus its content, is relevant to that particular node or not, so that each node may react only on the intended messages. If the message is relevant, it will be processed; otherwise it is ignored.
- How do they communicate?
  If the bus is free, any node may begin to transmit. But what will happen in situations where two or more nodes attempt to transmit message (to the CAN bus) at the same time. The identifier field, which is unique throughout the network helps to determine the priority of the message. A "non-destructive arbitration technique" is used to accomplish this, to ensure that the messages are sent in order of priority and that no messages are lost. The lower the numerical value of the identifier, the higher the priority. That means the message with identifier having more dominant bits (i.e. bit 0) will overwrite other nodes' less dominant identifier so that eventually (after the arbitration on the ID) only the dominant message remains and is received by all nodes.



CAN 2.0A (Standard Frame)

Frame	START	IDENTIFIER	RTR	IDE	r0	DLC	DATA	CRC	ACK	EOF IF
Länge in Bit	1	11	1	1	1	4	0..8	15	2	10

- **SOF- Start of Fr**ame. The message starts from this point.
- **Identifier**: It decides the priority of the message. Lower the binary value, higher is the priority. It is 11 bit.
- **RTR**– Remote Transmission Request. It is dominant when information is required from another node. Each node receives the request, but only that node whose identifier matches that of the message is the required node. Each node receives the response as well.z
- **IDE**– Single Identification Extension. If it is dominant, it means a standard CAN identifier with no extension is being transmitted.
- **R0**– reserved bit.
- **DLC**– Data Length Code. It defines the length of the data being sent. It is 4 bit
- **Data**– Up to 64 bit of data can be transmitted.
- **CRC**– Cyclic Redundancy Check. It contains the checksum (number of bits transmitted) of the preceding application data for error detection.
- **ACK**– Acknowledge. It is 2 bit. It is dominant if an accurate message is received.
- **EOF**– end of frame. It marks end of can frame and disables bit stuffing.
- **IFS**– Inter Frame Space. It contains the time required by the controller to move a correctly received frame to its proper position.



Different message types are:

1. **Data Frame**: It consists of arbitrary field, data field, crc field and the acknowledge fields.
2. **Remote Frame**: It requests for transmission of data from another node. Here the RTR bit is recessive.
3. **Error Frame**: It is transmitted when a error is detected.
4. **Overload Frame**: It is used to provide delay between messages. It is transmitted when the nodes become too busy.
5. **Valid Frame**: A message is valid if the EOF field is recessive. Else the message is transmitted again.

3.1.3 USB BUS

It connets flash memory cards, pen-like memory devices, digital camera, printer, mouse-device, PocketPC, video games, Scanner etc

USB allows Serial transmission and reception between host and serial devices . The data transfer is of four types:

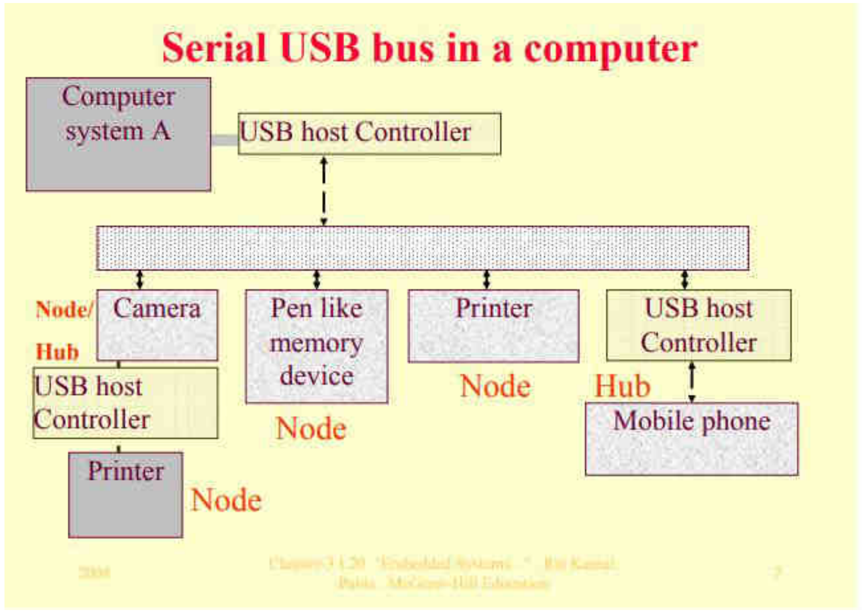
(a) Controlled data transfer, (b) Bulk data transfer, (c) Interrupt driven data transfer, (d) Iso-synchronous transfer

A bus between the host system and interconnected number of peripheral devices .Maximum 127 devices can connect a host.

Three standards:

- USB 1.1 (a low speed 1.5 Mbps 3 meter channel along with a high speed 12 Mbps 25 meter channel)
- USB 2.0 (high speed 480 Mbps 25 meter channel)
- wireless USB (high speed 480 Mbps 3 m)

Host connection to the devices or nodes is using USB port driving software and host controller, Host computer or system has a hostcontroller, which connects to a root hub. A hub is one that connects to other nodes or hubs. A tree- like topology is used



The root hub connects to the hub (s) and node (s) at level 1. A hub at level 1 connects to the hub (s) and node (s) at level 2 and so on. Root hub and each hub at a level have a star topology with the next level. Only the nodes are present at the last level.

USB Device features : Device Can be hot plugged (attached), configured and used, reset, reconfigured and used Bandwidth sharing with other devices: Host schedules the sharing of bandwidth among the attached devices at an instance. Can be detached (while others are in operation) and reattached. Attaching and detaching USB device or host without rebooting
USB device descriptor: Has data structure hierarchy as follows: It has device descriptor at the root, which has number of configuration descriptors, which has number of interface descriptor and which has number of end point descriptor.
Powering USB device : A device can be either bus-powered or self- powered. In addition, there is a power management by software at the host for USB ports
USB protocol :

- USB bus cable has four wires, one for +5V, two for twisted pairs and one for ground.Termination impedances at each end as per the device-speed.
- Electromagnetic Interference (EMI)- shielded cable for the 15 Mbps USB devices.
- Serial signals NRZI (Non Return to Zero (NRZI) The synchronization clock encoded by inserting synchronous code (SYNC) field before each USB packet Receiver synchronizes its bits recovery clock continuously
- A polled bus
- Host controller regularly polls the presence of a device as scheduled by the software. It sends a token packet. The token consists of fields for type, direction, USB device address and device end-point number. • The device does the handshaking through a handshake packet, indicating successful or unsuccessful transmission. A CRC field in a data packet permits error detection

USB supported three types of pipes

1. 'Stream' with no USB- defined protocol. It is used when the connection is already established and the data flow starts
2. 'Default Control' for providing access. •
3. 'Message' for the control functions for of the device.

Host configures each pipe with the data bandwidth to be used, transfer service type and buffer sizes.

Wireless USB:

Wireless extension of USB 2.0 and it operates at UWB (ultra wide band) 3.1 GHZ to 10.6 GHz frequencies. For short-range personal area network (high speed 480 Mbps 3 meter or 110 Mbps 10 meter channel)

3.1.4 FireWire-IEEE 1395 Bus Standard

Firewire connecting

• FireWire IEEE 1394a port up to 400 Mbps
• 1394b up to 800 Mbps
• Serial isosynchronous data transfer
• Transfers data at a guaranteed rate
• Also used in real time devices, such as video device data transfers

Applications

Multimedia streaming devices

• digital video cameras,
• digital camcorders,

• digital video disk (DVD),
• set-top boxes,
• music systems multimedia peripherals,
• latest hard disk drives,
• latest high speed printers

FireWire IEEE 1394 Protocol Features

- A single 1394 port can interface up to 63 external FireWire devices.
- Supports both plug and play and hot plugging.
- Provides self-powered and buspowered support on the bus

3.1.5 Advanced Serial High Speed Bus 3

Are for handheld devices.The following protocols were used

1. IEE 802.3 -2000[Gbps bandwidth Gigabit Ethernet MAC for 125Mhz performance]
2. IEEP802.3oe draft 4.1[10mbps Ethernet performance]
3. IEEP802.3oe draft 4.1[12.5mbps Ethernet performance]
4. XAUI(10 Gigabit attachment unit)
5. XSBI(10 Gigabit Serial Bus Interchange)
6. SONET OC-48
7. SONET OC-192
8. SONET OC-768
9. ATM OC-12/46/192F

3.2 Parallel bus device protocols

Parallel bus interconnects IO devices and peripherals over very short distances and at high speed. ISA, PCI and ARM buses are the examples of parallel buses. A parallel bus interfaces the system memory bus through a bridge or switching circuit.

3.2.1. ISA BUS

An Industry Standard Architecture bus (ISA bus) is a computer bus that allows additional expansion cards to be connected to a computer's motherboard. It is a standard bus architecture for IBM compatibles. Introduced in 1981, the ISA bus was designed to support the Intel 8088 microprocessor for IBM's first-generation PC. The ISA bus has evolved from its original 8-bit standard to 16-bit standard available in most PCs today. It operates at 8.33MHz. Its data transfer data rate is 8MB/s. It has 24 address lines and 16 data lines. It is used to connect printer, scanner, modem, sound card, CD-ROM etc. In the late 1990s the faster peripheral component interconnect (PCI). Soon afterwards, use of the ISA bus began to diminish, and most IBM motherboards were designed with PCI slots.The ISA bus provides direct memory access using multiple expansion cards on a memory channel allowing separate interrupt request transactions for each card. Depending on the version, the ISA bus can support a network card, additional serial ports, a video card and other processors and architectures, including:

- IBM PC with Intel 8088 microprocessor
- IBM AT with Intel 80286 processor (1984)
- Extended Industry Standard Architecture (1988)

EISA bus is the 32 bit data and address line version of ISA and devices also are supported

3.2.2 PCI and PCI/X Buses

Parallel bus enables a host computer or system to communicate simultaneously 32-bit or 64-bit with other devices or systems, for example, to a network interface card (NIC) or graphic card.
When the I/O devices in the distributed embedded subsystems are networked all can communicate through a common parallel bus. • PCI connects at high speed to other subsystems having a range of I/O devices at very short distances (<25cm) using a parallel bus without having to implement a specific interface for each I/O device.
PCI bus Applications: PCI bus connects display monitor, printer, character devices, network subsystems, video card, modem card, hard disk controller, thin client, digital video capture card, streaming displays, 10/100 Base T card, Card with 16 MB Flash ROM with a router gateway for a LAN and Card using DEC 21040 PCI Ethernet LAN controller.

PCI Bus Features

- 32- bit data bus extendible to 64 bits.

- PCI protocol specifies the ways of interaction between the different components of a computer.
- A specification version 2.1 ─ synchronous/asynchronous throughput is up to 132/ 528 MB/s [33M × 4/ 66M × 8 Byte/s], operates on 3.3V to 5V signals.
- PCI driver can access the hardware automatically as well as by the programmer assigned addresses. Automatically detects the interfacing systems and assigns new addresses Thus, simplified addition and deletion (attachment and detachment) of the system peripherals.
- Each device may use a FIFO controller with a FIFO buffer for maximum throughput.

Identification Numbers : A device identifies its address space by three identification numbers, (i) I/O port (ii) Memory locations and (iii) Configuration registers of total 256B with a four 4-byte unique ID. Each PCI device has address space allocation of 256 bytes to access it by the host computer

PCI device identification A sixteen16-bit register in a PCI device identifies this number to let that device auto- detect it. Another sixteen16-bit register identifies a device ID number. These two numbers let allow the device to carry out its auto-detection by its host computer.

PCI Standards

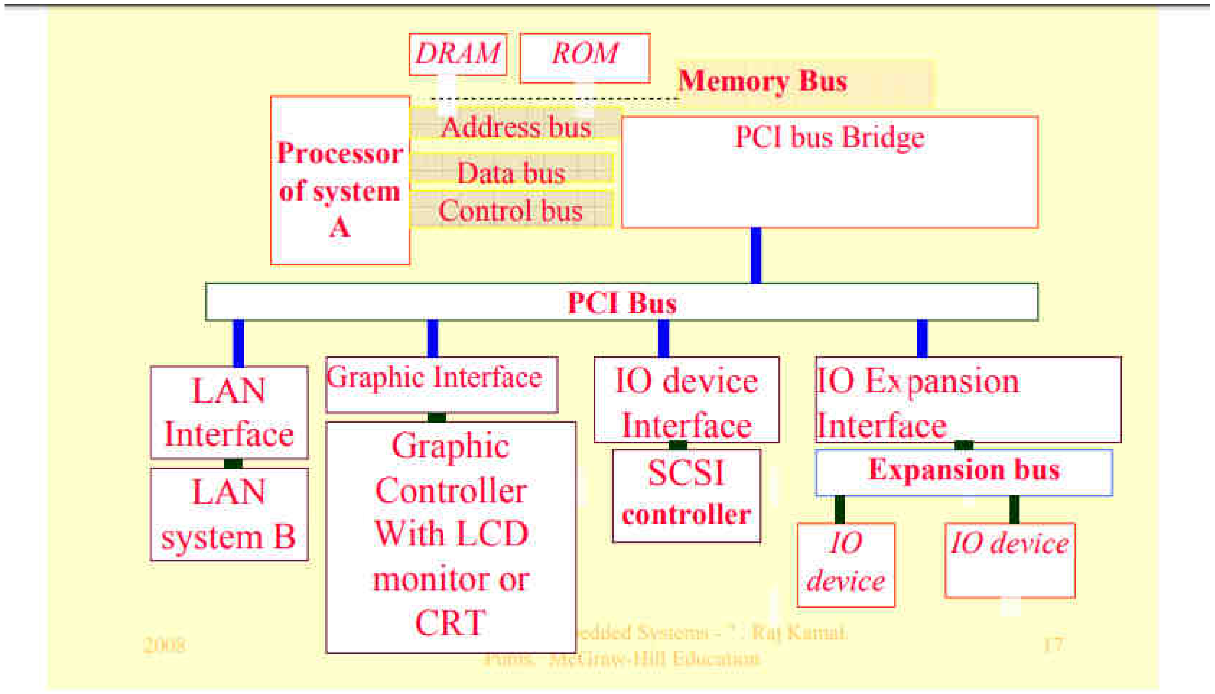
- PCI 32bit/33 MHz, and 64bit/66 MHz
- PCI Extended (PCI/X) 64 bit/100 MHz
- Compact PCI (cPCI) Bus

Two super speed versions

- PCI Super V2.3 264/528 MBps 3.3V (on 64- bit bus), and 132/264 (on 32-bit bus)
- PCI-X Super V1.01a for 800MBps 64- bit bus 3.3Volt.

PCI bridge : PCI bus interface switches a processor communication with the memory bus to PCI bus. In most systems, the processor has a single data bus that connects to a switch module PCI bridge Some processors integrate the switch module onto the same integrated circuit as the processor to reduce the number of chips required to build a system and thus the system cost.

PCI bridge/switch Communicates with the memory through a memory bus (a set of address, control and data buses), a dedicated set of wires that transfer data between these two systems. A separate I/O bus connects the PCI switch to the I/O devices.



PCI/X(PCI Extended)

- 33 MBps to as much as 1 GBps
- Backward compatible with existing PCI cards
- Used in high bandwidth devices (Fiber Channel, and processors that are part of a cluster and Gigabit Ethernet)
- Maximum 264 MBps throughput, uses 8, 16, 32, or 64 bit transfers
- 6U cards contain additional pins for user defined I/Os
- Live insertion support (Hot-Swap),
- Supports two independent buses on the back plane (on different connectors)
- Supports Ethernet, Infiniband, and Star Fabric support (Switched fabric based systems) Compact PCI (cPCI)

Each PCI device on Bus Each PCI device on Bus Perform a specific function, May contain a processor and software to perform a specific function. Each device has the specific memory address-range, specific interrupt-vectors (pre-assigned or auto configured) and the device I/O port addresses. A bus of appropriate specifications and protocol interfaces these to the host computer system or compute

Unique feature of PCI bus unique feature is its configuration address space.

PCI Controller features

• Accesses one device at a time

• All the devices within host device or system can share the I/O port and memory addresses, but cannot share the configuration registers

• Device cannot modify other configuration registers but can access other device resources or share the work or assist the other device

PCI device initilization

• A device can initialize at booting time

• Avoids any address collision

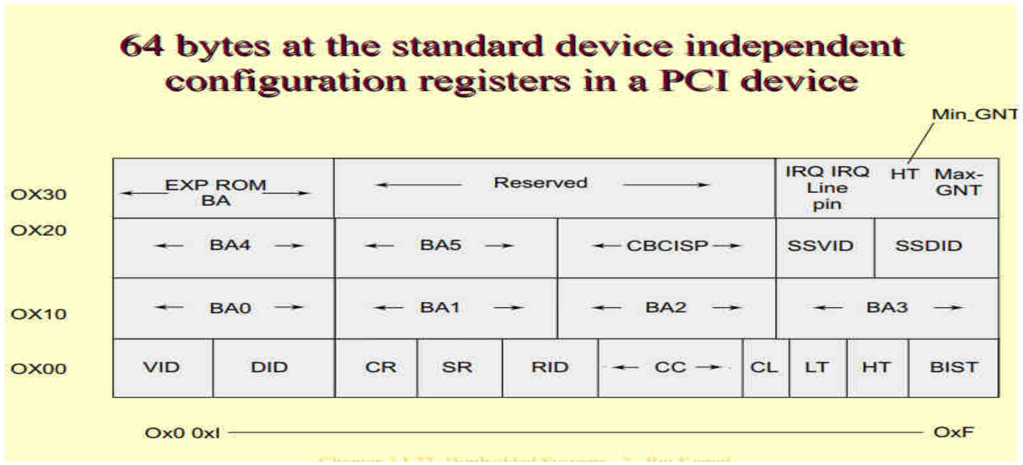
• Device on boot up disables its interrupt and closes its door to its address space except to the configuration registers space

PCI BIOS

Performs the configuration transactions and then, memory and address spaces automatically map to the address space in the device hosting system

PCI device Interrupt Handling : A uniquely assigned interrupt type (a number) handles an interrupt. For example, interrupt type 3 has the interrupt vector address 0x0000C and four bytes at the address specify the interrupt service routine address. Interrupt type can be a number between 0x00 and 0xFF.

Configuration register number 60 :Stores the one byte for the interrupt type n (pci) The PCI device when interrupted handles the interrupt of type n(pci)



Meaning of Terms in Figure
VID: Vendor ID.
DID: Device ID.
RID: Revision ID.
CR: Common Register.
CC: Class Code.
SR: Status Register.
CL: Cache Line.
LT: Latency Timer.
BIST: Base Input Tick.
HT: Header Type.
BA: Base Address. C
BCISB: Card Base CIS Pointer.
SS: Sub System.
ExpROM: Expansion ROM.
MIN_GNT: Minimum Guaranteed time
MAX_GNT: Maximum Guaranteed Time.

3.2.3 ARM Bus

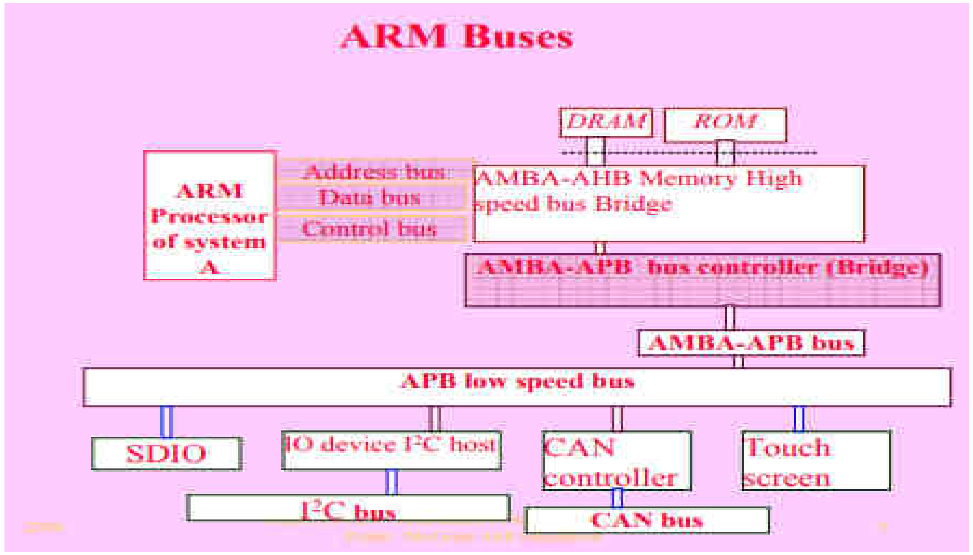
ARM processor interfaces the memory ,external DRAM using AMBA .

AMBA (ARM Main Memory Bus Architecture) AMBA (ARM Main Memory Bus Architecture) AHB (ARM High Performance Bus)

- AMBA-AHB interfaces the memory, external DRAM (dynamic RAM controller and on-chip I/O devices
- AMBA-AHB connects to 32-bit data and 32-bit address lines at high speed
- AHB maximum bps bandwidth─ sixteen times ARM processor clock

3.2.4 AMBA (ARM Main Memory Bus Architecture) AMBA (ARM Main Memory Bus Architecture) APB (ARM Peripheral Bus)

AMBA -APB interfaces ARM processor with the memory AMBAAHB and external -chip I/O devices, which operate at low speed using a bridge (AMBA-APB bridge)
AMBA -APB bridge :Switches ARM CPU communication with the AMBA bus to APB bus. ARM processor based microcontroller has a single data bus in AMBA-AHB that connects to the bridge, which integrate the bridge onto the same integrated circuit as the processor to reduce the number of chips required to build a system and thus the system cost. The bridge communicates with the memory through a AMBA-AHB, a dedicated set of wires that transfer data between these two systems. A separate APB I/O bus connects the bridge to the I/O devices.



APB bus
connects I ^{2} C , touch screen, SDIO ,MMC (multimedia card) , USB , CAN bus and other required interfaces to an ARM microcontroller

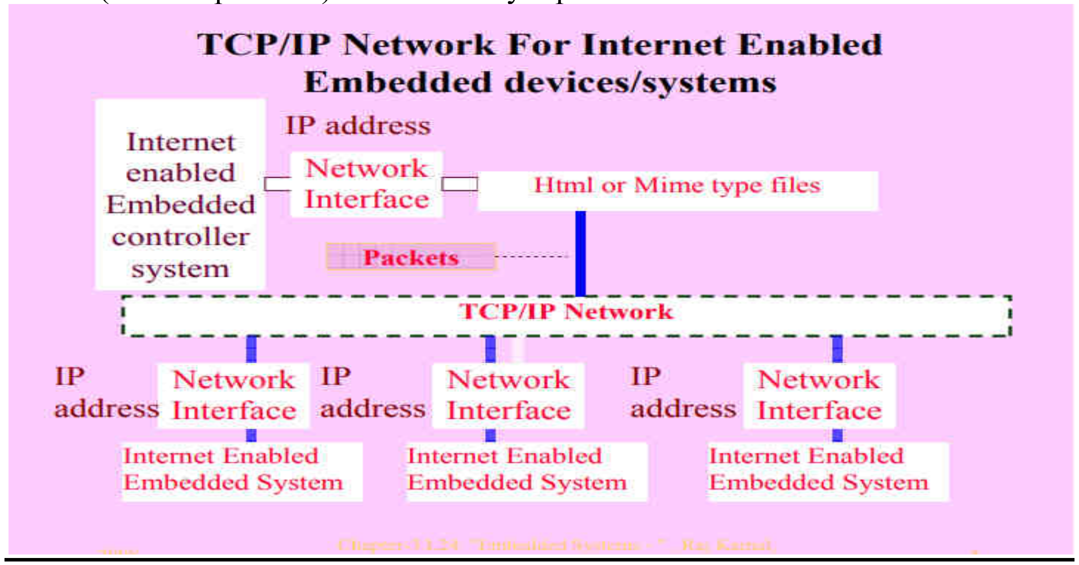
3.2.5 Advanced parallel high speed buses
An embedded system may need to connect IO system using gigabit parallel synchronous interfaces. The following are advanced bus standard and proprietary protocols developed recently

1. GMII-Gigabit Ethernet MAC internet Interface
2. XGMI-10 Gigabit Ethernet MAC internet Interface
3. CSIX-1.6.6 Gbps 32bit HSTL with 200Mhz performance
4. Rapidio TM Interconnect specification v1.1 at 8Gbps with 500Mbps performance

4. Internet enabled systems-Network protocols

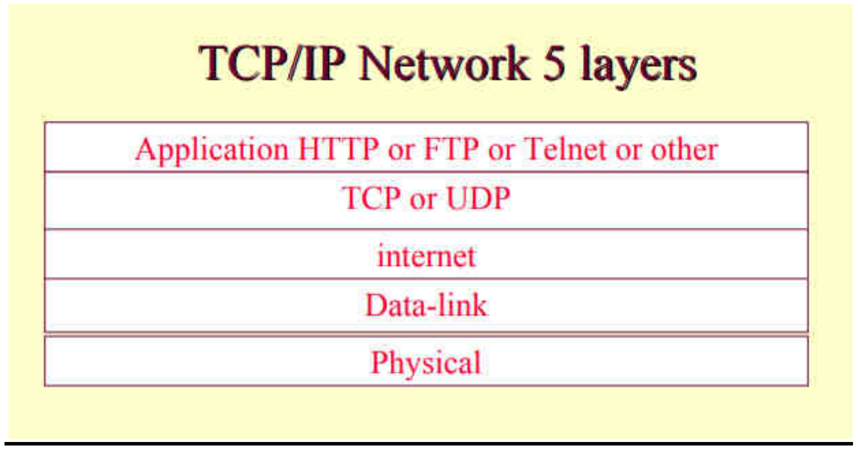
- Internet enabled system have Communication to other on the Internet.
- Use html (hyper text markup language) or MIME (Multipurpose Internet Mail Extension) type files

- Use TCP (transport control protocol) or UDP (user datagram protocol) as transport layer protocol addressed by an IP address
- Use IP (internet protocol) at network layer protocol


TCP/IP Network For Internet Enabled Embedded devices/systems
Internet enabled Embedded controller system
IP address
Network Interface
Html or Mime type files
Packets
TCP/IP Network
IP address
Network Interface
Internet Enabled Embedded System
IP address
Network Interface
Internet Enabled Embedded System
IP address
Network Interface
Internet Enabled Embedded System

MIME Format to enable attachment of multiple Format to types of files txt (text file) , doc (MSOFFICE Word document file) , gif (graphic image format file) jpg (jpg format image file) , wav format voice or music file

A system at one IP address Communication with other system at another IP address using the physical connections on the Internet and routers . Since Internet is global network, the system connects to remotely as well as short range located system.
There are five layers in a TCP/IP network ─ Application, transport, network, data-link and physical Application layer protocol also specifies presentation ways. Transport layer protocol also specifies provide session establishment and termination ways. Each layer has a protocol, which specifies the way in which the data or message from previous layer transfer to next layer



4.1 Hyper-Text Transfer protocol(HTTP)

This Layer accepts the data, for example, in HTML or text format and puts the header words as per the protocol and sends application header and data to transport layer. A port number specifies the application in the header. A port assigned number supports multiple logical connections using a socket, and a socket has an IP address and port number. A registered port number is between 0 and 1023. Registration is done by IANA (Internet Assigned Number Authority. Port number 0 means host itself.

Following are the important application layer protocols that support TCP/IP networking

1. HTTP (Port 80) enables Internet connectivity by Hyper-Text Transfer Protocol (HTTP).
2. FTP (Port 21 for control, 20 for data) enables file transfer connectivity by File Transfer Protocol.
3. TFTP (Port 69) for Trivial FTP.
4. NFS (Network File System) is used for sharing files on a network.
5. TELNET (Port 23) enables remote login to remote terminals by Terminal Access Protocol.
6. SMTP (Port 25) enables e-mail transfer, store and forward by Simple Mail Transfer Protocol.

7. PoP3 (Port 110) enables e-mail retrieval.
8. NNTP (Port 119) by (Network News Transfer Protocol).
9. DNS (Port 53) for Domain Name Service.
10. SNMP (Port 161) is Simple Network Management Protocol.
11. Bootps and Bootpc (Ports 67 and 68) for Bootstrap Protocol Server and Client, respectively.
12. DHCP (Dynamic Host Configuration Protocol) is used for remote booting as well as for configuring a system.

HTTP (Port 80) features HTTP (Port 80) features

- ➢ standard protocol for requesting for a URL (universal resource locator, for example, http://www.mcgraw-hill.com)

- ➢ stateless protocol. For HTTP request, the protocol assumes a fresh request. It means there is no session or sequence number field or no field that is retained in the next exchange. This makes a current exchange by an HTTP request independent of the previous exchanges

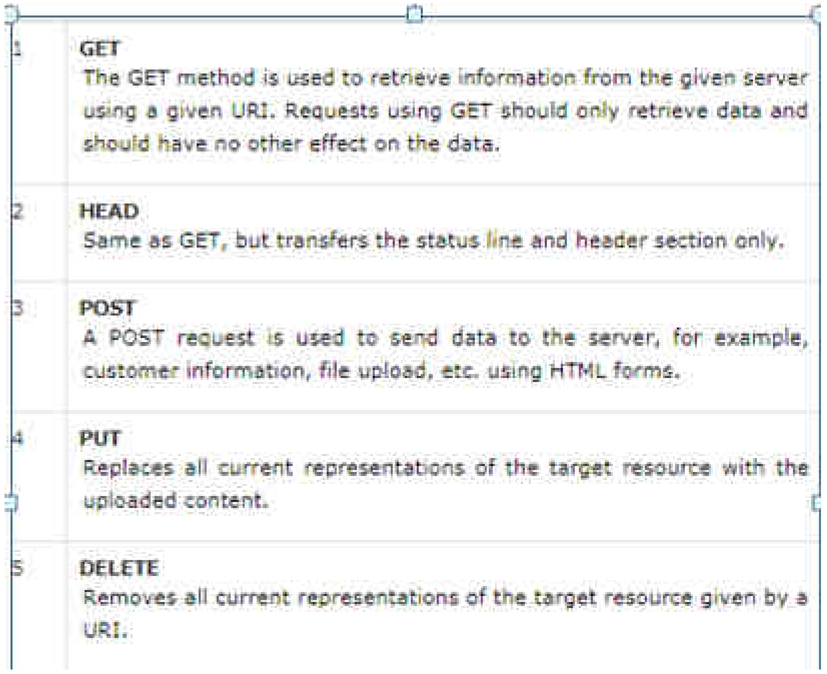
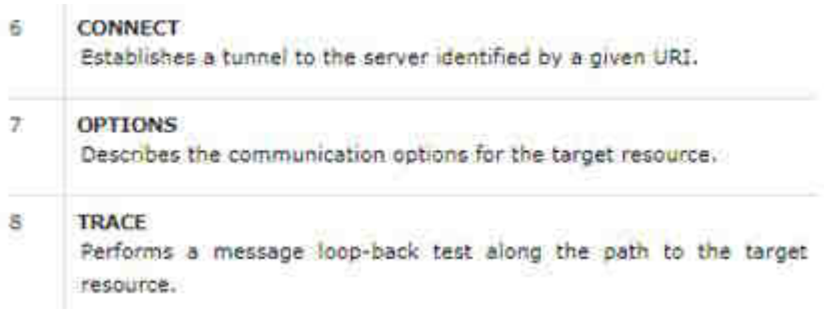
  .

- ➢ A file–transfer like protocol for HTML files. This makes it easy to explore a web site URL. A request (from a client) is sent and reply (response from a server) is received.

- ➢ The HTTP protocol is very light (a small format) and thus speedy as compared to other existing protocols. HTTP is able to transfer any type of data to a browser (a client) provided it is capable of handling that data.

- ➢ HTTP is flexible

- ➢ HTTP protocol is based on Object Oriented Programming System. Methods are applied to objects identified by URL
  HTTP specific methods

1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but transfers the status line and header section only.
3	**POST** A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.
4	**PUT** Replaces all current representations of the target resource with the uploaded content.
5	**DELETE** Removes all current representations of the target resource given by a URI.
6	**CONNECT** Establishes a tunnel to the server identified by a given URI.
7	**OPTIONS** Describes the communication options for the target resource.
8	**TRACE** Performs a message loop-back test along the path to the target resource.

HTTP interaction scheme A client requests a server directly or through proxy- or gateway An HTTP message therefore either a request or a response

4.2 Transport control protocol(TCP)

TCP protocol use in transport layer Accepts the message from the upper layer (application layer) on a transmission by an application or session layer. Accepts a data stream from the network layer at the receiving end. Before communicating a message to the next network layer, it may add a header.

TCP message : The message may communicate in parts or segments or fragments. The header generally has the additional bits for the source and destination addresses. Also there are bits in it for the sequence and the acknowledge management, flow and error controls, etc.

TCP feature : Specifies a format of byte streams at the transport layer of the TCP/IP suite. TCP is used for a full duplex acknowledged flow. Its format has a TCP header of five plus (n −5) words for options and padding and data of maximum l words.

4.3 User datagram protocol(UDP)

When message is connection less and stateless, then transport layer protocol in TCP/IP suite specifies a protocol called UDP. UDP supports the broadcast networking mode. Example ─ application for communicating a header before a data stream. UDP header specifies the bits for the source and destination ports, the total length of message including header and check sum (optional). During reception, UDP message to the upper layer flows after deleting the header bits from the received transport layer header. Header bits added at the transmitting time from the application or session layers are thus strip from the message

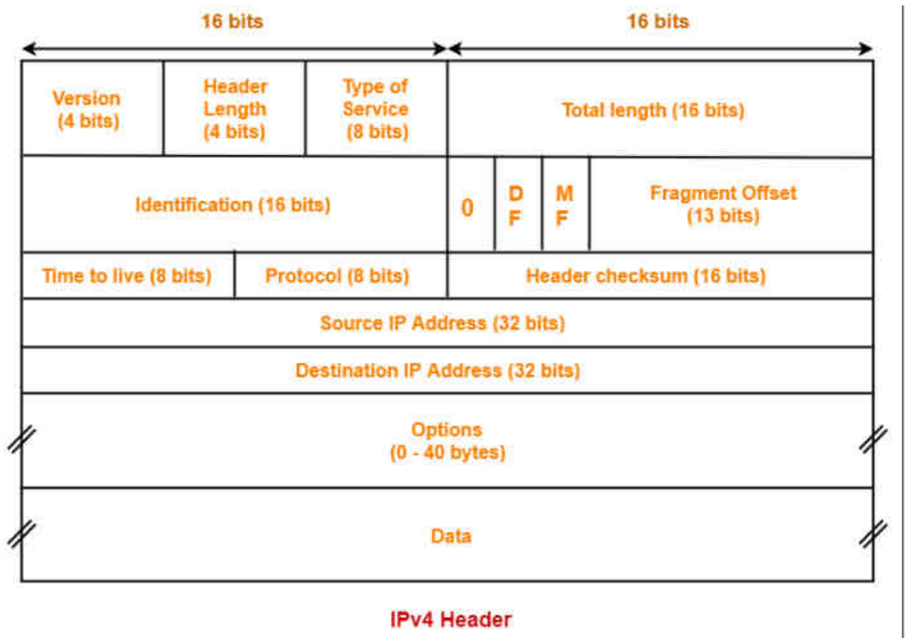
4.4 Internet protocol

All internet enabled devices communicate using internet protocol.Transport layer data in TCP or UDP message format transmits on the network after first division into the packets at the network layer, called internet layer.

IP Packet transmission : Each packet transmits through a chain of routers on the Internet. Packet a minimum unit of data, which transmit on Internet through the routers. Several packets form a source can reach a destination using different routes and can have different delays.

IP packet and routing of packets: The packet consists of an IP header plus data or an IP header plus a routing protocol with the routing messages. The packet has a maximum of 216 bytes (214 words, 1 word = 32 bits = 4 bytes).

The network routing is as per the standard IPv4 (Internet protocol version 4) or IPv6 (Internet protocol version 6). IPv6 broadband protocol.



1. **Version-** is a 4 bit field that indicates the IP version used.The most popularly used IP versions are version-4 (IPv4) and version-6 (IPv6).

2. **Header length** -is a 4 bit field that contains the length of the IP header.It helps in knowing from where the actual data begins.

3. **Type of service**- is a 8 bit field that is used for Quality of Service (QoS).The datagram is marked for giving a certain treatment using this field.
4. **Total length**- is a 16 bit field that contains the total length of the datagram (in bytes).
5. **Identification**- is a 16 bit field.It is used for the identification of the fragments of an original IP datagram.
6. **DF bit-** stands for Do Not Fragment bit.Its value may be 0 or 1. When DF bit is set to 0,It grants the permission to the intermediate devices to fragment the datagram if required.When DF bit is set to 1,It indicates the intermediate devices not to fragment the IP datagram at any cost.
7. **Fragment Offset** is a 13 bit field.It indicates the position of a fragmented datagram in the original unfragmented IP datagram.
8. **Time to live** (TTL) is a 8 bit field.-It indicates the maximum number of hops a datagram can take to reach the destination.
9. **Protocol** is a 8 bit field.It tells the network layer at the destination host to which protocol the IP datagram belongs to.
10. **Header checksum** is a 16 bit field.It contains the checksum value of the entire header.

11. **Source IP Address** is a 32 bit field.it contains the logical address of the sender of the datagram.
12. **Destination IP Address** is a 32 bit field it contains the logical address of the receiver of the datagram.
13. Options is a field whose size vary from 0 bytes to 40 bytes.This field is used for several purposes such as Record route,Source routing &Padding
14. **Padding**-Addition of dummy data to fill up unused space in the transmission unit and make it conform to the standard size is called as padding.

4.5 Ethernet

Ethernet is very widely used as a local area network for general-purpose computing. The physical organization of an Ethernet is very simple, as shown in Figure 8.14. The network is a bus with a single signal path; the Ethernet standard allows for several different implementations such as twisted pair and coaxial cable. Unlike the I2 C bus, nodes on the Ethernet are not synchronized—they can send their bits at any time. I2 C relies on the fact that a collision can be detected and quashed within a single bit time thanks to synchronization. But since Ethernet nodes are not synchronized, if two nodes decide to transmit at the same time, the message will be ruined.

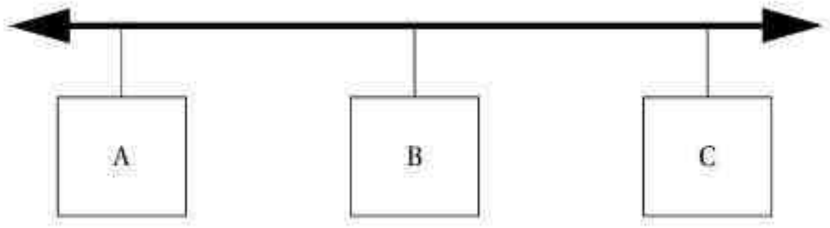


FIGURE 8.14

Ethernet organization.

The Ethernet arbitration scheme is known as ***Carrier Sense Multiple Access*** with ***Collision Detection (CSMA/CD)***. The algorithm is outlined in Figure 8.15. A node that has a message waits for the bus to become silent and then starts transmitting. It simultaneously listens, and if it hears another transmission that interferes with its transmission, it stops transmitting and waits to retransmit. The waiting time is random, but weighted by an exponential function of the number of times the message has been aborted. Figure 8.16 shows the expo- nential backoff function both before and after it is modulated by the random wait time. Since a message may be interfered with several times before it is successfully transmitted, the ***exponential backoff*** technique helps to ensure that the network does not become overloaded at high demand factors. The random factor in the wait time minimizes the chance that two messages will repeatedly interfere with each other. The maximum length of an Ethernet is determined by the nodes' ability to detect collisions.

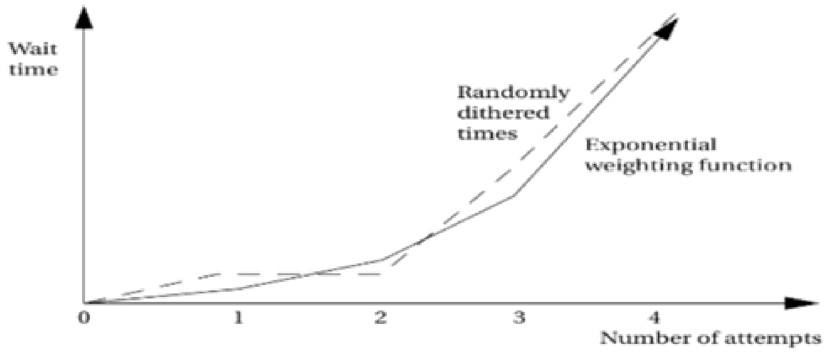
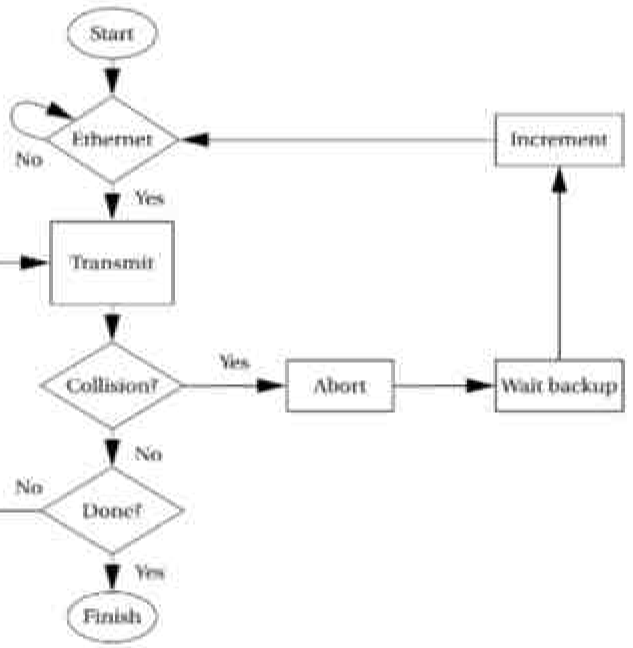


FIGURE 8.16

Exponential backoff times.

FIGURE 8.15

The Ethernet CSMA/CD algorithm.



Preamble	Start frame	Destination address	Source address	Length	Data	Padding	CRC

FIGURE 8.17

Ethernet packet format.

5 . Wireless and Mobile network protocols

5.1 IrDA (Infrared Data Association)
Used in mobile phones, digital cameras, keyboard, mouse, printers to communicate to laptop computer and for data and pictures download and synchronization. Used for control TV, air-conditioning, LCD projector, VCD devices from a distance.Use infrared (IR) after suitable modulation of the data bits. Communicates over a line of sight Phototransistor receiver for infrared rays

5.2 Bluetooth 2.4 GHz
Bluetooth is a wireless technology standard for exchanging data between fixed and mobile devices over short distances using short-wavelength UHF radio waves in the industrial, scientific and medical radio bands, from 2.400 to 2.485 GHz, and building personal area networks (PANs). It was originally conceived as a wireless alternative to RS-232 data cables.

5.3 802.11

In IEEE 802.11 wireless local area networking standards (including Wi-Fi), a **service set** is a group of wireless network devices that are operating with the same networking parameters.

Service sets are arranged hierarchically,: **basic service sets** (**BSS**) are units of devices operating with the same medium access characteristics (i.e. radio frequency, modulation scheme etc.), while **extended service sets** (**ESS**) are logical units of one or more basic service sets on the same logical network segment (i.e. IP subnet, VLAN etc.). There are two classes of basic service sets: those that are formed by infrastructure mode redistribution points (access points or mesh nodes), and those that are formed by independent stations in a peer-to-peer ad hoc topology. Basic service sets are identified by **BSSIDs** (**basic service set identifiers**), which are 48-bit labels that conform to MAC-48 conventions. Logical networks (including extended service sets) are identified by **SSIDs** (**service set identifiers**), which serve as "network names" and are typically natural language labels.

5.4 ZigBee

Zigbee is an IEEE 802.15.4-based specification for a suite of high-level communication protocols used to create personal area networks with small, low-power digital radios, such as for home automation, medical device data collection, and other low-power low-bandwidth needs, designed for small scale projects which need wireless connection. Hence, Zigbee is a low-power, low data rate, and close proximity (i.e., personal area) wireless ad hoc network.

The technology defined by the Zigbee specification is intended to be simpler and less expensive than other wireless personal area networks (WPANs), such as Bluetooth or more general wireless networking such as Wi-Fi. Applications include wireless light switches, home energy monitors, traffic management systems, and other consumer and industrial equipment that requires short-range low-rate wireless data transfer.

6. EMBEDDED PRODUCT DEVELOPMENT LIFE CYCLE (EDLC)

Just like the SDLC used in Software Development, there is EDLC used in Embedded product development. This chapter explains what is the EDLC, its objectives, the phases that are involved in the EDLC.
EDLC is Embedded Product Development Life Cycle

It is an Analysis – Design – Implementation based problem solving approach for embedded systems development.

Analysis involves understanding what product needs to be developed

Design involves what approach to be used to build the product

Implementation is developing the product by realizing the design.

Need for EDLC

- EDLC is essential for understanding the scope and complexity of the work involved in embedded systems development

- It can be used in any developing any embedded product
- EDLC defines the interaction and activities among various groups of a product development phase including project management, system design ,System testing, Release management and quality assurance

  ,

Objectives of EDLC

- The ultimate aim of any embedded product in a commercial production setup is to produce Marginal benefit
- Marginal is usually expressed in terms of Return On Investment
- The investment for product development includes initial investment, manpower, infrastructure investment etc.
- EDLC has three primary objectives are:

  1. Ensuring that high quality products are delivered to user

  2. Risk minimization and defect prevention in product development through project management

  3. Maximize productivity

1. Ensuring that high quality products are delivered to user

➢ The primary definition of quality in any embedded product development is return on investment achieved by the product. The expenses incurred for developing the product the product are:-

Initial investment

Developer recruiting

Training

Infrastructure requirement related

- In order to survive in market, quality is very important factor to be taken care of while developing the product.

- Qualitative attributes depends on the budget of the product so budget allocation is very important.

- Budget allocation might have done after studying the market, trends & requirements of product, competition .etc.

- ELDC must ensure that the development of the product has taken account of all the quality attributes of the embedded system

2. Risk minimization and defect prevention in product development through project management

- There are projects in embedded product development which requires loose or tight management. If the development is a simple one a senior developer itself can take the charge of a management activity and there is no need for a skilled project manager to look after this with dedicated effort. But there will be an overall supervision form the skilled project management team for ensuring that the development process is going in the right direction. Project which are complex and requires timeliness should have a dedicated and a skilled project management part and hence they are said to be tightly bounded to project management.
- Project management -

  -Adds an extra cost on budget

  -But essential for ensuring the development process is going in right direction

  - Projects in EDLC requires Loose project management or tight project management.

  - PM is required for

Predictability

| || || || |Analyze the time to finish the product (PDS = no of person days).this is estimated on the basis of past experience

Co-ordination

Resources (developers) needed to do the job

Risk management

- Backup of resources to overcome critical situation
- Ensuring defective product is not developed

- ➢ project management is essential for ' predictability co-ordination and risk minimization
- ➢ Resource allocation is critical and it is having a direct impact on investment
- ➢ Microsoft @ Project Tool is a typical example for CASE tools for project management

3. Increased Productivity

- ➢ Productivity is a Measure of efficiency as well as ROI This productivity measurement is based on total manpower efficiency
- ➢ Different ways to improve the productivity are

- Saving the manpower
    - X members – X period
    - X/2 members – X period(the productivity is doubled)
- Use of automated tools where ever is required
- Re-usable effort – work which has been done for the previous product can be used if similarities present b/w previous and present product.
- Use of resources with specific set of skills which exactly matches the requirements of the product, which reduces the time in training the resource. Recruiting people with desired skill set for current project is another option ,this is worth only if you expect to have more work on the near future on the same technology or skill sets.

➢ ELDC should take all this aspects into consideration to provide maximum productivity

7.. DIFFERENT PHASES OF EDLC

- A life cycle of product development is commonly referred as the “model”
- A simple model contains five phases
    - Requirement analysis
    - Design
    - Development and test

- Deployment and maintenance

- In real product development the phases given in this model may vary and can have sub models or model contain only certain important phases of classic model
- The no of phases involved in EDLC model depends on the complexity of the product.
- The following section describes each phase of classic ELDC model in detail

The following figure depicts the different phases in EDLC:

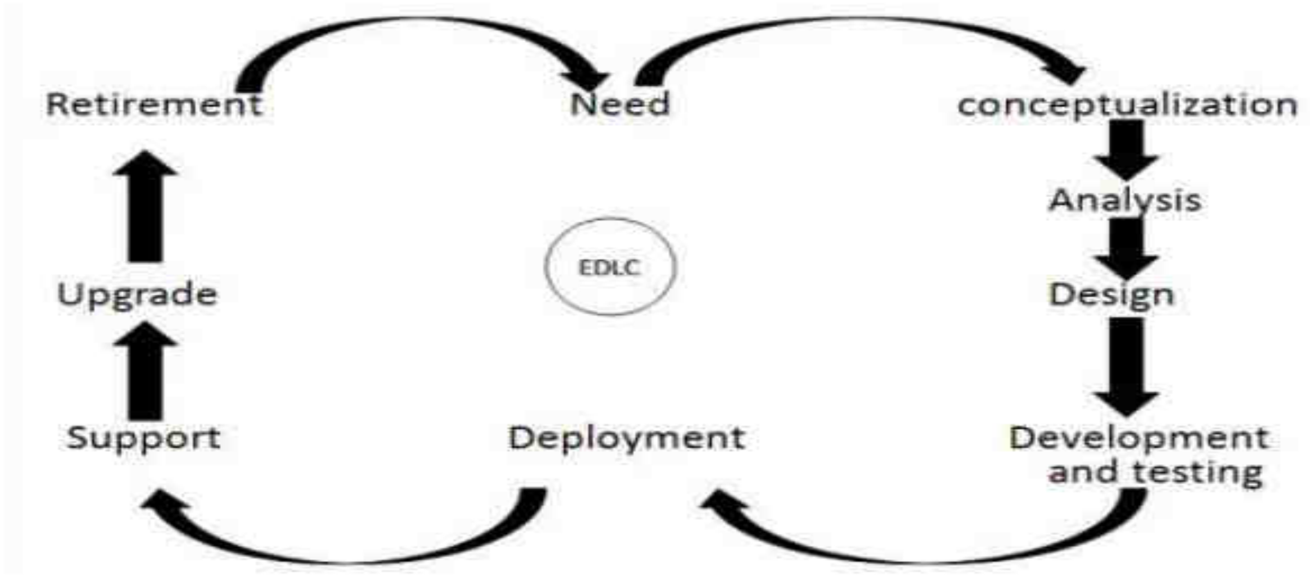


Figure : Phases of EDLC

7.1 Need

- ➢ The embedded product involves the output of a need

- The need may come from an individual or from the public or from a company.
- 'Need' should be articulated to initiate the Development Life Cycle; a 'Concept Proposal' is prepared which is reviewed by the senior management for approval.

Product development Need can be visualized in any one of the following three needs:

a. **New or Custom Product Development**.-The need for a product that doesn't exist in the market or a product which act as a competitor to an existing product in the current market will need to the development of a completely new product. Example of an embedded product which act as a competitor in market is Mobile handset

b. **Product Re-engineering.-**The embedded product market is competitive and dynamic. Re-engineering an existing product comes as a result of following needs.

   I. Change in Business requirement
   II. User interface Enhancements
   III. Technology Upgrades

c. **Product Maintenance.-**This need deals with providing technical support to the end user for an existing product in the market. The maintenance request may comes as the result of product nonfunctioning or failure

7.2 Conceptualisation

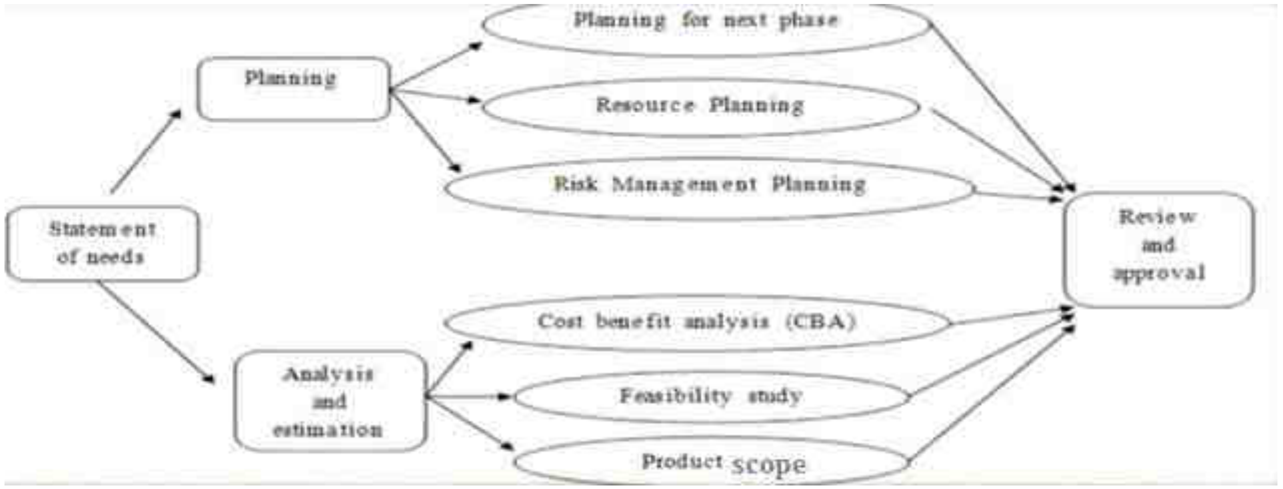
- Is the ***product concept development phase*** and it begins immediately after the concept proposal is formally approved.
- These phases define the scope of concept, performs cost benefit analysis and feasibility study and prepare project management and risk management plans.
- The conceptualization involves two types of activities, *planning activity and analysis and study activity*. This are performed to understand the opportunity of the product in the market.

- **Planning activity** Requires various plans to be developed first before development like
  - Planning for next phase
  - Resource planning
  - risk management planning
- **Analysis and Study activities i**nvolves

Feasibility Study : Examine the need and suggest possible solutions.

Cost Benefit Analysis (CBA): Revealing and assessing the total development cost and profit expected from the product. It is somewhat related to loss gain analysis in business. Some Common underlying principles are given below like Common Unit of measurement, market choice based benefit measurement, target end users etc..

Product Scope: Deals with what is scope(functionalities need to be considered) and what is not in scope.Should be properly documented for future reference



7.3. Analysis

- Requirement analysis phase start immediately after the documents submitted during the conceptualization phase is approved by the client / sponser of the project
- This phase is performed to develop a detailed functional model of the product under consideration
- The product is defined in detail with respect to the inputs, processes, outputs, and interfaces at a functional level.
- Emphasize on what functions must be performed by the product rather than how to perform those functions

The various activities performed during this phase..

I. **Analysis and Documentations:** This activity consolidates the business needs of the product under development.
Requirements that need to be addressed during this phase are

1. Functional Capabilities like performance
2. Operational and non-operational quality attribute
3. Product external interface requirements
4. User manuals θ Data requirements
5. Operational requirements
6. Maintenance requirements
7. General assumptions

II. Interface definition and documentation
This activity should clearly analyse on the physical interfaces as well as data exchange through this interfaces and should document it.

III. **Defining Test Plan and Procedures**: The various type of testing performed in a product development are:

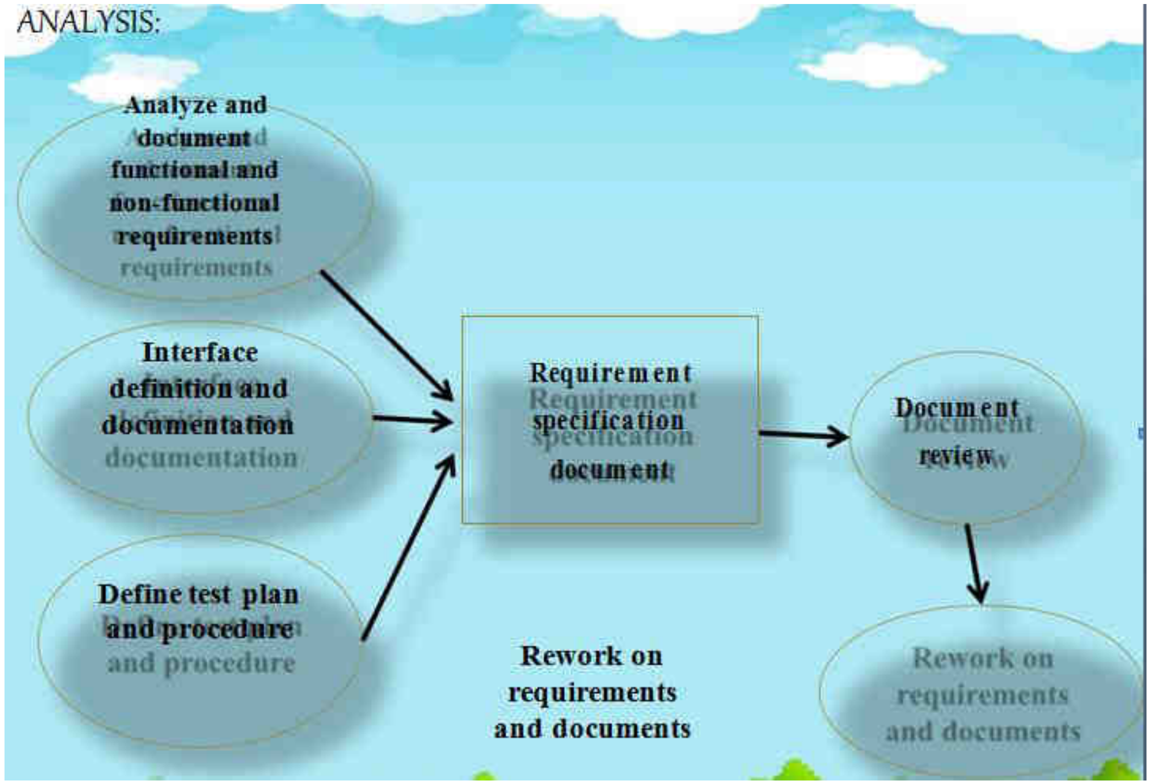
1. Unit testing – Testing Individual modules

2. Integration testing – Testing a group of modules for required functionality

3. **System testing-** Testing functional aspects or functional requirements of the product after integration. Different tests are
    - **Usability testing**: test the usability of the product
    - **Load testing** :test the behavior of the product under different loading conditions
    - **Security testing**: test the security aspects of the product
    - **Scalability** testing: test the scalability aspects of the product
    - **Sanity testing**: Superficial testing is performed to ensure that the product is functioning properly
    - **Smoke testing**: ensures the crucial requirement of the product are functioning properly
    - **Performance testing**: test the performance aspects of the product after integration
    - **Endurance testing**:durability test of the product

4. User acceptance testing- Testing the product to meet the end user requirements.

At the end all requirements are put in a template suggested by a standard process model


ANALYSIS:
Analyze and document functional and non-functional requirements
Interface definition and documentation
Define test plan and procedure
Requirement specification document
Document review
Rework on requirements and documents
Rework on requirements and documents

7.4 Design

- The design phase identifies application environment and creates an overall architecture for the product.
- It starts with the Preliminary Design. It establishes the top level architecture for the product. On completion it resembles a ‘black box’ that defines only the inputs and outputs. The final product is called Preliminary Design Document (PDD).
- Once the PDD is accepted by the End User the next task is to create the ‘Detailed Design’.

- It encompasses the Operations manual design, Maintenance Manual Design and Product Training material Design and is together called the 'Detailed Design Document'.

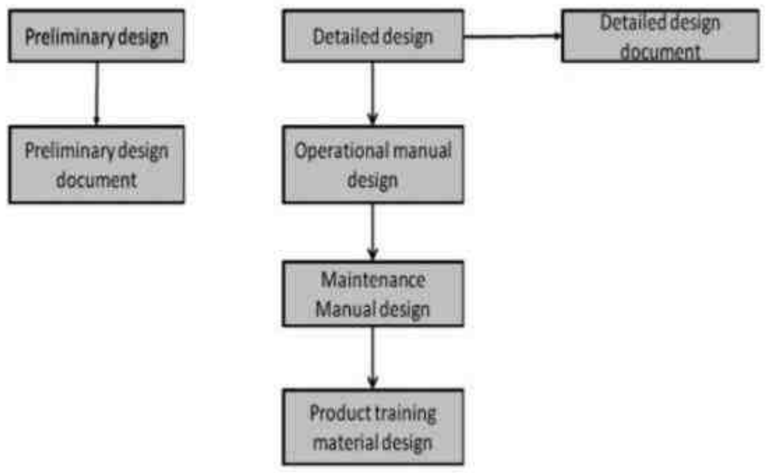
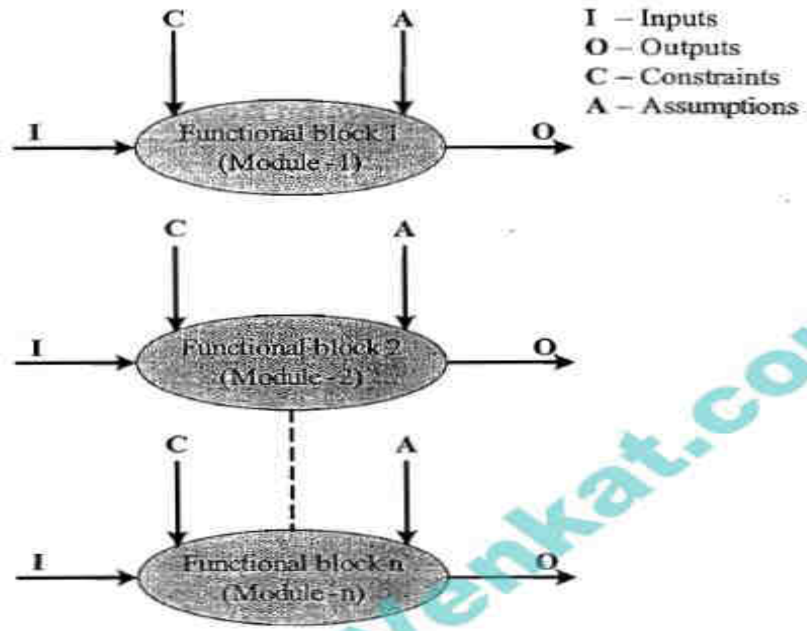

Preliminary design
Preliminary design document
Detailed design
Detailed design document
Operational manual design
Maintenance Manual design
Product training material design

Fig.various activities involved in design phase



ig. 15.5 **Preliminary Design Illustration**

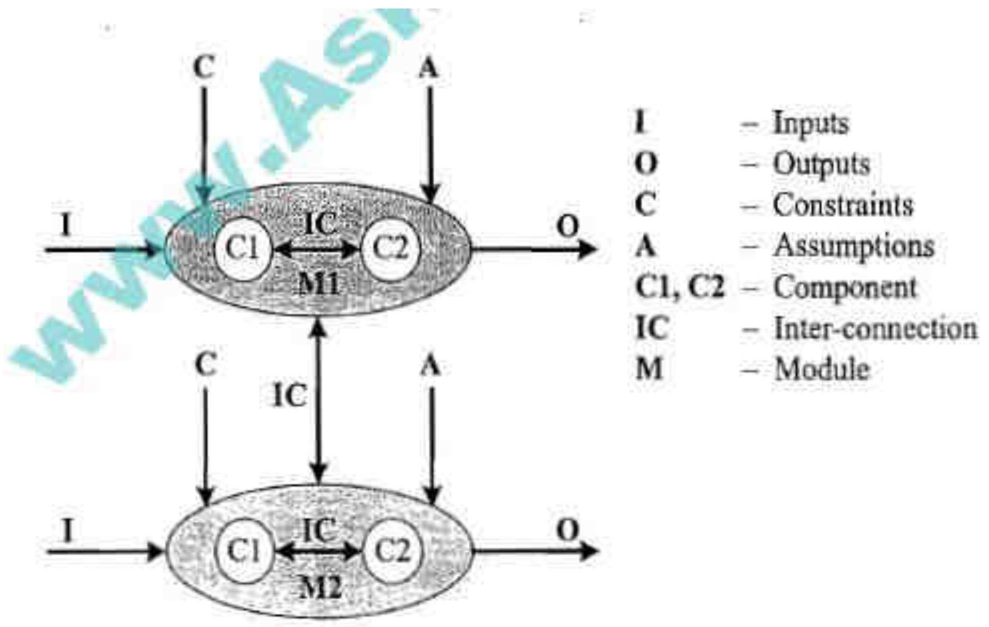


Fig. 15.6 **Detailed Design Illustration**

7.5 Development and Testing

- Development phase transforms the design into a realizable product.
- The detailed specification generated during the design phase is translated into hardware and firmware.
- The Testing phase can be divided into independent testing of firmware and hardware that is:

Unit testing

Integration testing

System testing

User acceptance testing

7.6 **Deployment**

- Deployment is the process of launching the first fully functional model of the product in the market.It is also known as First Customer Shipping (FCS).

Tasks performed during this phase are:

I. **Notification of Product Deployment:** Tasks performed here include:

Deployment schedule

Brief description about the product

Targeted end user

Extra features supported

Product support information

II. **Execution of training plan**

Proper training should be given to the end user top get them acquainted with the new product.

III. **Product installation**

Install the product as per the installation document to ensure that it is fully functional.

IV. **Product post Implementation Review**

After the product launch, a post implementation review is done to test the success of the product

7.7 Support

- The support phase deals with the operational and maintenance of the product in the production environment
- Bugs in the product may be observed and reported.
- The support phase ensures that the product meets the user needs and it continues functioning in the production environment.

Activities involved under support are

Setting up of a dedicated support wing: Involves providing 24 x 7 supports for the product after it is launched.

Identify Bugs and Areas of Improvement: Identify bugs and take measures to eliminate them.

7.8 Upgrades

- Deals with the development of upgrades (new versions) for the product which is already present in the market.
- Product upgrade results as an output of major bug fixes.
- During the upgrade phase the system is subject to design modification to fix the major bugs reported.

7.9 Retirement/Disposal

- The retirement/disposal of the product is a gradual process.
- This phase is the final phase in a product development life cycle where the product is declared as discontinued from the market.
- The disposal of a product is essential due to the following reasons
  - Rapid technology advancement
  - Increased user needs

8. ELDC APPROACHES(modeling the eldc)

Following are some of the different types of approaches that can be used to model embedded products.

1. Waterfall or Linear Model
2. Iterative/ Incremental or Fountain Model
3. Prototyping Model/Evolutionary Model
4. Spiral Model

Waterfall or Linear Model

- Linear or waterfall model is the one adopted in most of the olden systems.
- In this approach each phase of EDLC (Embedded Development Product Lifecycle) is executed in sequence.

- It establishes analysis and design with highly structured development phases.
- The execution flow is unidirectional.
- The output of one phase serves as the input of the next phase
- All activities involved in each phase are well planned so that what should be done in the next phase and how it can be done.
- The feedback of each phase is available only after they are executed.
- It implements extensive review systems To ensure the process flow is going in the right direction.
- One significant feature of this model is that even if you identify bugs in the current design the development process proceeds with the design.
- The fixes for the bug are postponed till the support phase.

Advantages

Product development is rich in terms of:

Documentation

Easy project management

Good control over cost & Schedule

Drawbacks

It assumes all the analysis can be done without doing any design or implementation

The risk analysis is performed only once.

The working product is available only at the end of the development phase

Bug fixes and correction are performed only at the maintenance/support phase of the life cycle.

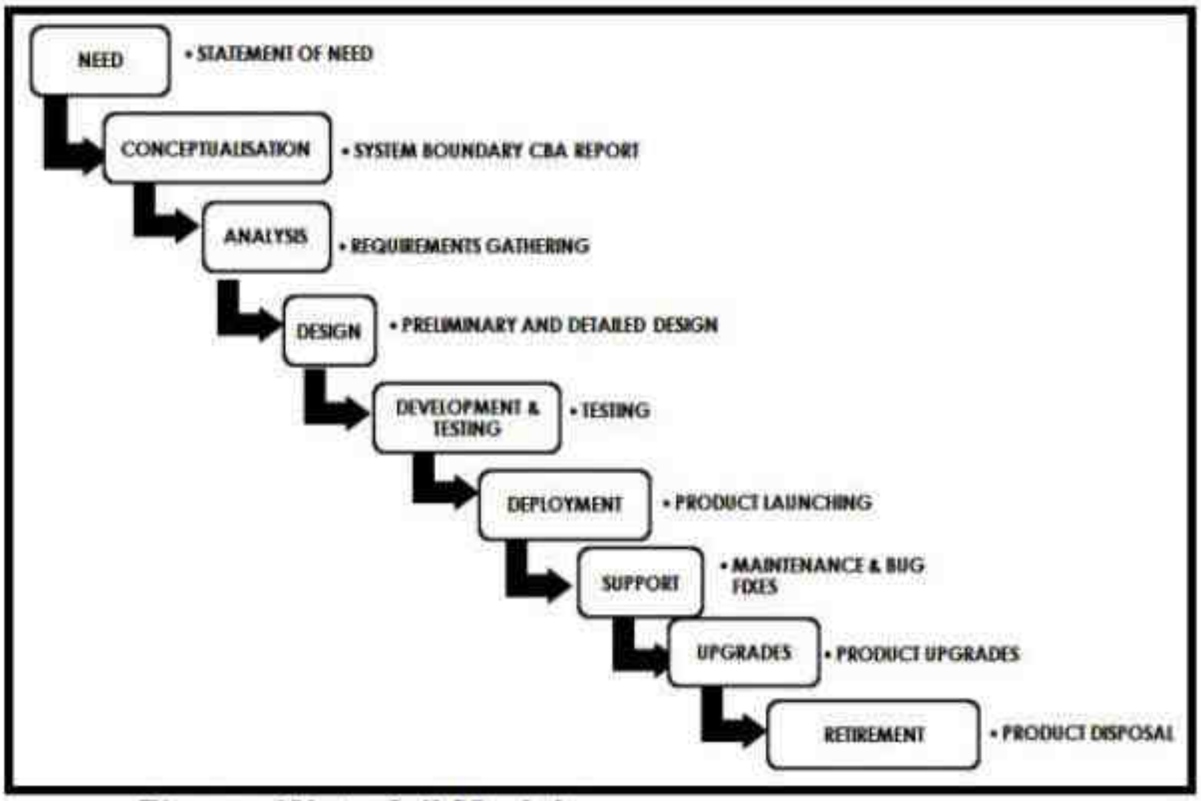
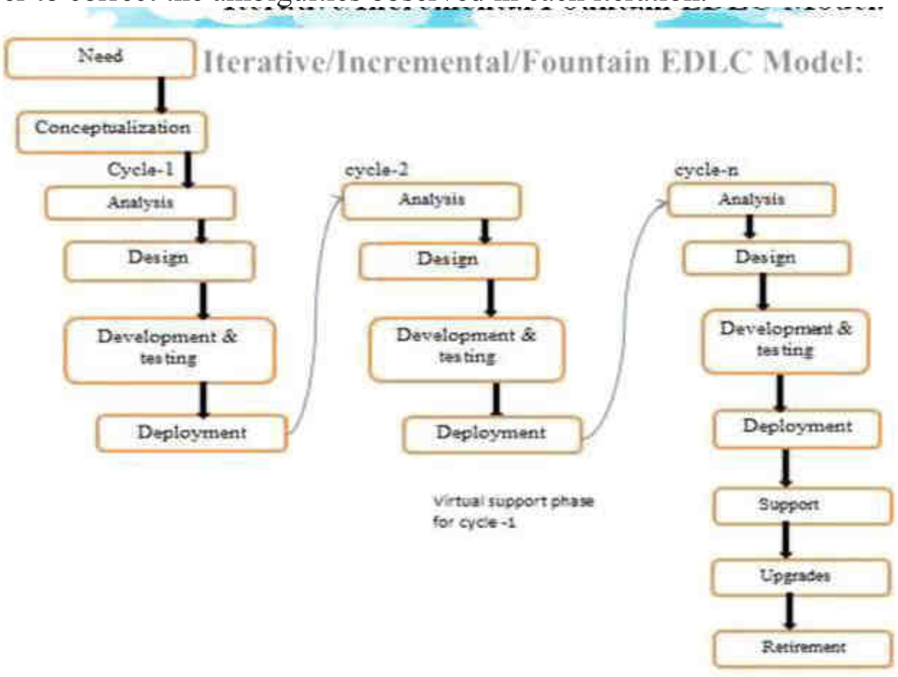


Figure: Waterfall Model

2. ITERATIVE/ INCREMENTAL OR FOUNTAIN MODEL

- Iterative and Incremental development is at the heart of a cyclic software development process developed in response to the weaknesses of the waterfall model.
- The iterative model is the repetitive process in which the Waterfall model is repeated over and over to correct the ambiguities observed in each iteration.



- The above figure illustrates the repetitive nature of the Iterative model.
- The core set of functions for each group is identified in the first cycle, it is then built, deployed and release. This release is called as the first release.
- Bug fixes and modification for first cycle carried out in second cycle.
- Process is repeated until all functionalities are implemented meeting the requirements.

Advantages

- Good development cycle feedback at each function/feature implementation
- Data can be used as reference for similar product development in future.
- More responsive to changing user needs.
- Provides working product model with at least minimum features at the first cycle.
- Minimized Risk
- Project management and testing is much simpler compared to linear model.
- Product development can be stopped at any stage with a bare minimum working product.

Disadvantages

- Extensive review requirement each cycle.
- Impact on operations due to new releases.
- Training requirement for each new deployment at the end of each development cycle.
- Structured and well documented interface definition across modules to accommodate changes

3. PROTOTYPING MODEL

- It is similar to iterative model and the product is developed in multiple cycles.
- The only difference is that, Prototyping model produces a refined prototype of the product at the end of each cycle instead of functionality/feature addition in each cycle as performed by the iterative model.
- There won't be any commercial deployment of the prototype of the product at each cycle's end.
- The shortcomings of the proto-model after each cycle are evaluated and it is fixed in the next cycle.
- After the initial requirement analysis, the design for the first prototype is made, the development process is started.
- On finishing the prototype, it is sent to the customer for evaluation.
- The customer evaluates the product for the set of requirements and gives his/her feedback to the developer in terms of shortcomings and improvements needed.
- The developer refines the product according to the customer's exact expectation and repeats the proto development process.
- After a finite number of iterations, the final product is delivered to the customer and launches in the market/operational environment
- In this approach the product undergoes significant evolution as a result of periodic shuttling of product information between the customer and developer

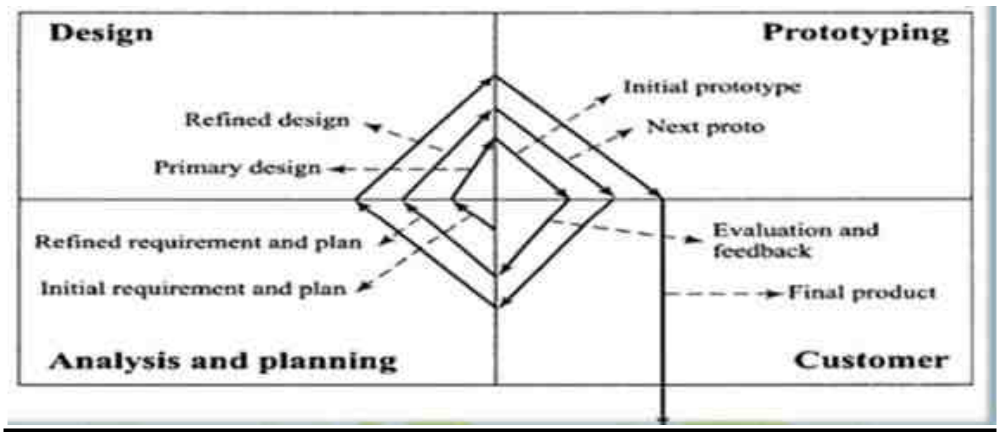
The prototyping model follows the approach-

Requirement definition

Proto-type development

Proto-type evaluation

Requirements refining



4. SPIRAL MODEL

- **Spiral model is developed by Barry Boehm in 1988.**
- The Product development starts with project definition and traverse through all phases of EDLC(Embedded Product Development Life Cycle).

The activities involved are:

I. Determine objectives, alternatives, constraints
II. Evaluate alternatives, identify and resolve risks
III. Develop and test
IV. Plan

It is a combines the concept of Linear Model and iterative nature of Prototyping Model.

Prototyping Model

- In prototyping after the requirement analysis the design for the prototype is made and development process is started.
- On finishing the prototype it is send to the customer for evaluation ie. Judgment.
- After customer evaluation for the product the feedback is taken from the customer in term of what improvement is needed.
- Then developer refines the product according to the customer expectation.

Linear Model

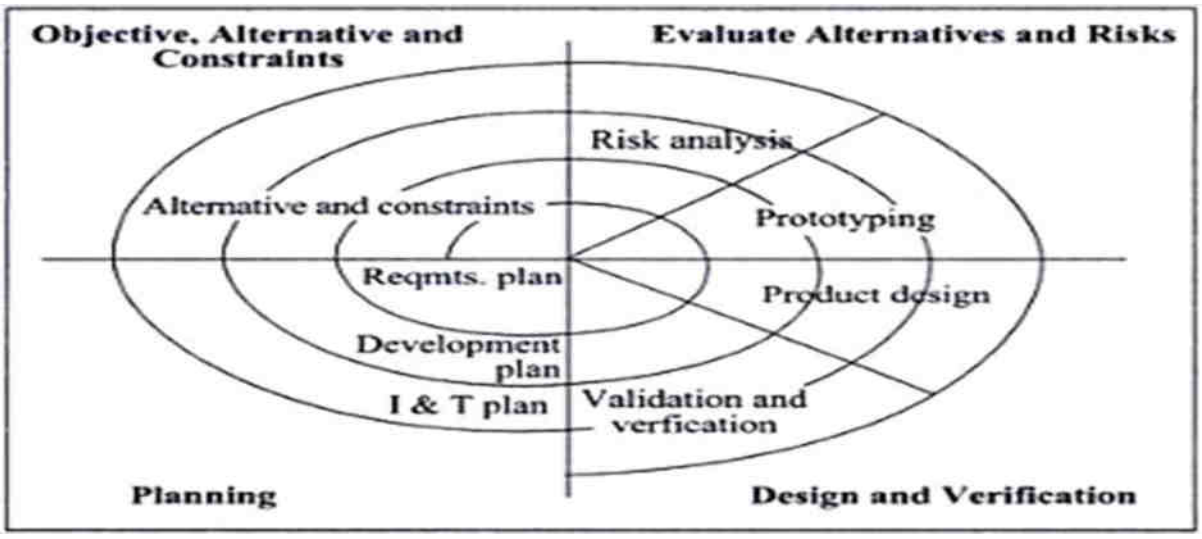
- Spiral Model contains the concept of linear model, having following type.

Requirement

Analysis

Design

Implementation


Objective, Alternative and Constraints
Evaluate Alternatives and Risks
Risk analysis
Alternative and constraints
Prototyping
Reqmts. plan
Product design
Development plan
I & T plan
Validation and verfication
Planning
Design and Verification

Requirement:

- This process is focused specifically on embedded software, to understand the nature of the software to be build and what are the requirement for the software.
- And the requirement for both the system & the software is documented & viewed to customer.

Analysis:

- Analysis is performed to develop a detailed functional module under consideration.
- The product is defined in detailed with respect to the input, processing & output.
- This phase emphasis on determining 'what function must be performed by the product' & how to perform those function.

Design:

- Product design deals with the entire design of the product taking the requirement into consideration.
- The design phase translates requirement into representation.

Implementation:

- In this process the launching of first fully functional model of the product in the market is done or handing over the model to an end user/client

- In this product modifications are implemented & product is made operational in production environment.

9. TRENDS IN EMBEDDED SYSTEMS

1. Processor Trends
2. Operating System Trends
3. Development Language Trends
4. Open Standards, Frameworks and alliances
5. Bottlenecks faced by Embedded Industry

9.1. PROCESSOR TRENDS

There have been tremendous advancements in the area of processor design.

Following are some of the points of difference between the first generation of processor/controller and today's processor/ controller.

Number of ICs per chip: Early processors had a few number of IC/gates per chip. Today's processors with Very Large Scale Integration (VLSI) technology can pack together ten of thousands of IC/gates per processor.

Need for individual components: Early processors need different components like brown out circuit, timers, DAC/ADC separately interfaced if required to be used in the circuit. Today's processors have all these components on the same chip as the processor.

Speed of Execution: Early processors were slow in terms of number of instructions executed per second. Today's processor with advanced architecture support features like instruction pipeline improving the execution speed.

Clock frequency: Early processors could execute at a frequency of a few MHz only. Today's processors are capable of achieving execution frequency in rage of GHz.

Application specific processor: Early systems were designed using the processors available at that time. Today it is possible to custom create a processor according to a product requirement.

Following are the major trends in processor architecture in embedded development.

9.1.1 System on Chip (SoC)

This concept makes it possible to integrate almost all functional systems required to build an embedded product into a single chip.

SoC are now available for a wide variety of diverse applications like Set Top boxes, Media Players, PDA, etc.

SoC integrate multiple functional components on the same chip thereby saving board space which helps to miniaturize the overall design.

9.1.2 Multicore Processors/ Chiplevel Multi Processor

This concept employs multiple cores on the same processor chip operating at the same clock frequency and battery.

Based on the number of cores, these processors are known as:

- Dual Core – 2 cores
- Tri Core – 3 cores
- Quad Core – 4 cores

These processors implement multiprocessing concept where each core implements pipelining and multithreading.

9.1.3 Reconfigurable Processors

It is a processor with reconfigurable hardware features.

Depending on the requirement, reconfigurable processors can change their functionality to adapt to the new requirement. Example: A reconfigurable processor chip can be configured as the heart of a camera or that of media player.

These processors contain an Array of Programming Elements (PE) along with a microprocessor. The PE can be used as a computational engine like ALU or a memory element.

9.2 EMBEDDED OPERATING SYSTEM TRENDS

The advancements in processor technology have caused a major change in the Embedded Operating System Industry.

There are lots of options for embedded operating system to select from which can be both commercial and proprietary or Open Source.

Virtualization concept is brought in picture in the embedded OS industry which replaces the monolithic architecture with the microkernel architecture.

This enables only essential services to be contained in the kernel and the rest are installed as services in the user space as is done in Mobile phones.

Off the shelf OS customized for specific device requirements are now becoming a major trend.

9.3 DEVELOPMENT LANGUAGE TRENDS

There are two aspects to Development Languages with respect to Embedded Systems Development

Embedded Firmware

It is the application that is responsible for execution of embedded system.

It is the software that performs low level hardware interaction, memory management etc on the embedded system.

Embedded Software

It is the software that runs on the host computer and is responsible for interfacing with the embedded system.

It is the user application that executes on top of the embedded system on a host computer.

Early languages available for embedded systems development were limited to C & C++ only. Now languages like Microsoft C$, ASP.NET, VB, Java, etc are available.

9.3.1 **Java for embedded development**

Java is not a popular language for embedded systems development due to its nature of execution.

Java programs are compiled by a compiler into bytecode. This bytecode is then converted by the JVM into processor specific object code.

During runtime, this interpretation of the bytecode by the JVM makes java applications slower that other cross compiled applications.

This disadvantage is overcome by providing in built hardware support for java bytecode execution.

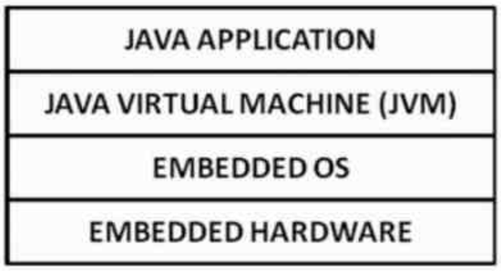


Figure: Java based Embedded Application Development

Another technique used to speed up execution of java bytecode is using Just In Time (JIT) compiler. It speeds up the program execution by caching all previously executed instruction.

Following are some of the disadvantage of Java in Embedded Systems development:

- For real time applications java is slow
- Garbage collector of Java is non-deterministic in behavior which makes it not suitable for hard real time systems.
- Processors need to have a built in version of JVM
- Those processors that don’t have JVM require it to be ported for the specific processor architecture.
- Java is limited in terms of low level hardware handling compared to C and C++
- Runtime memory requirement of JAVA is high which is not affordable by embedded systems.

9.3.2 .NET CF for embedded development

It stands for .NET Compact Framework.

.NET CF is a replacement of the original .NET framework to be used on embedded systems.

The CF version is customized to contain all the necessary components for application development.

The Original version of .NET Framework is very large and hence not a good choice for embedded development.

The .NET Framework is a collection of precompiled libraries.

Common Language Runtime (CLR) is the runtime environment of .NET. It provides functions like memory management, exception handling, etc.

Applications written in .NET are compiled to a platform neutral language called Common Intermediate Language (CIL).

For execution, the CIL is converted to target specific machine instructions by CLR.

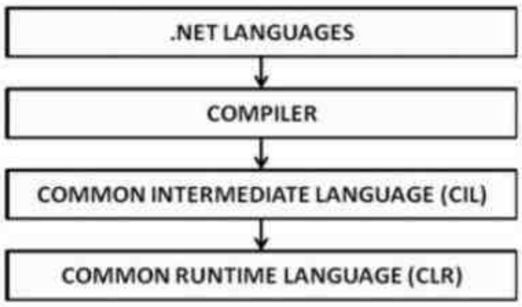

.NET LANGUAGES
COMPILER
COMMON INTERMEDIATE LANGUAGE (CIL)
COMMON RUNTIME LANGUAGE (CLR)

Figure: .NET based Embedded Application Development

9.4.OPEN STANDARDS, FRAMEWORKS AND ALLIANCES

Standards are necessary for ensuring interoperability. With diverse market it is essential to have formal specifications to ensure interoperability.

Following are some of the popular strategic alliances, open source standards and frameworks specific to the mobile handset industry.

9.4.1 Open Mobile Alliance (OMA)

- ➢ It is a standard body for creating open standards for mobile industry.
- ➢ OMA is the Leading Industry Forum for Developing Market Driven – Interoperable Mobile Service Enablers
- ➢ OMA was formed in June 2002 by the world's leading mobile operators, device and network suppliers, information technology companies and content and service providers.
- ➢ OMA delivers open specifications for creating interoperable services that work across all geographical boundaries, on any bearer network. OMA's specifications support the billions of new and existing fixed and mobile terminals across a variety of mobile

networks, including traditional cellular operator networks and emerging networks supporting machine-to-machine device communication.

- OMA is the focal point for the development of mobile service enabler specifications, which support the creation of interoperable end-to-end mobile services.

Goals of OMA

- Deliver high quality, open technical specifications based upon market requirements that drive modularity, extensibility, and consistency amongst enablers to reduce industry implementation efforts.
- Ensure OMA service enabler specifications provide interoperability across different devices, geographies, service providers, operators, and networks; facilitate interoperability of the resulting product implementations.
- Be the catalyst for the consolidation of standards activity within the mobile data service industry; working in conjunction with other existing standards organizations and industry fora to improve interoperability and decrease operational costs for all involved.

- Provide value and benefits to members in OMA from all parts of the value chain including content and service providers, information technology providers, mobile operators and wireless vendors such that they elect to actively participate in the organization.

9.4.2 Open Handset Alliance (OHA)

- The Open Handset Alliance is a group of 84 technology and mobile companies who have come together to accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience. Together they have developed Android™, the first complete, open, and free mobile platform and are committed to commercially deploy handsets and services using the Android Platform.

- Members of OHA include mobile operators, handset manufacturers, semiconductor companies, software companies, and commercialization companies.

9.4.3 Android

- Android is an operating system based on the Linux kernel, and designed primarily for touchscreen mobile devices such as smartphones and tablet computers.

- Initially developed by Android, Inc., which Google supported financially and later bought in 2005, Android was unveiled in 2007 along with the founding of the Open Handset Alliance: a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices.

- The first publicly-available Smartphone to run Android, the HTC Dream, was released on October 18, 2008

9.4.4Openmoko

Openmoko is a project to create a family of open source mobile phones, including the hardware specification and the operating system.

The first sub-project is Openmoko Linux, a Linux-based operating system designed for mobile phones, built using free software.

The second sub-project is developing hardware devices on which Openmoko Linux runs.

10 . Bottlenecks faced by Embedded Industry

Following are some of the problems faced by the embedded devices industry:

Memory Performance

The rate at which processors can process may have increased considerably but rate at which memory speed is increasing is slower.

Lack of Standards/ Conformance to standards

Standards in the embedded industry are followed only in certain handful areas like Mobile handsets.

There is growing trend of proprietary architecture and design in other areas.

Lack of Skilled Resource

Most important aspect in the development of embedded system is availability of skilled labor. There may be thousands of developers who know how to code in C, C++, Java or .NET but very few in embedded software.

Hardware Software Co-Design and Program Modelling – Fundamental Issues, Computational Models- Data Flow Graph, Control Data Flow Graph, State Machine, Sequential Model, Concurrent Model, Object oriented model, UML

Traditional Embedded System Development Approach

- The hardware software partitioning is done at an early stage
- Engineers from the software group take care of the software architecture development and implementation, and engineers from the hardware group are responsible for building the hardware required for the product
- There is less interaction between the two teams and the development happens either serially or in parallel and once the hardware and software are ready, the integration is performed

Fundamental issues in H/w S/w Co-design

➢ **Model Selection**

- A Model captures and describes the system characteristics
- A model is a formal system consisting of objects and composition rules
- It is hard to make a decision on which model should be followed in a particular system design.
- Most often designers switch between a variety of models from the requirements specification to the implementation aspect of the system design
- The objectives vary with each phase

➢ **Architecture Selection**

- A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'
- The architecture specifies how a system is going to implement in terms of the number and types of different components and the interconnection among them
- Controller architecture, Datapath Architecture, Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), Very long Instruction Word Computing (VLIW), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD) etc are the commonly used architectures in system design

➢ **Language Selection**

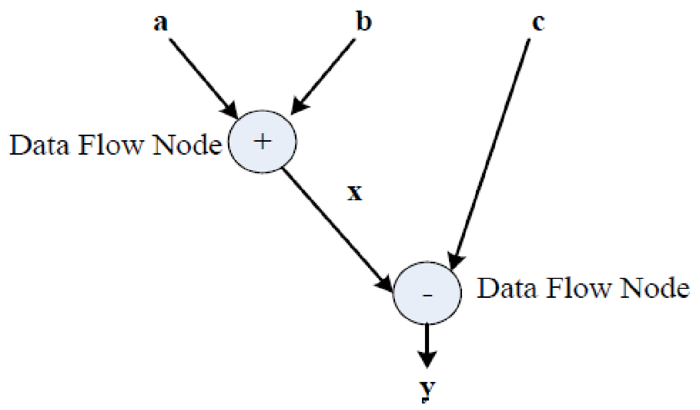
- A programming Language captures a 'Computational Model' and maps it into architecture
- A model can be captured using multiple programming languages like C, C++, C#, Java etc for software implementations and languages like VHDL, System C, Verilog etc for hardware implementations
- Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model.
- The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily

- **Partitioning of System Requirements into H/w and S/w**
  - Implementation aspect of a System level Requirement
  - It may be possible to implement the system requirements in either hardware or software (firmware)
  - Various hardware software trade-offs like performance, re-usability, effort etc are used for making a decision on the hardware-software partitioning

Computational Models in Embedded Design

- **Data Flow Graph/Diagram (DFG) Model**
  - Translates the data processing requirements into a data flow graph
  - A data driven model in which the program execution is determined by data.
  - Emphasizes on the data and operations on the data which transforms the input data to output data.
  - A visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation
  - Best suited for modeling Embedded systems which are computational intensive (like DSP applications)

    E.g. Model the requirement x = a + b; and y = x - c;

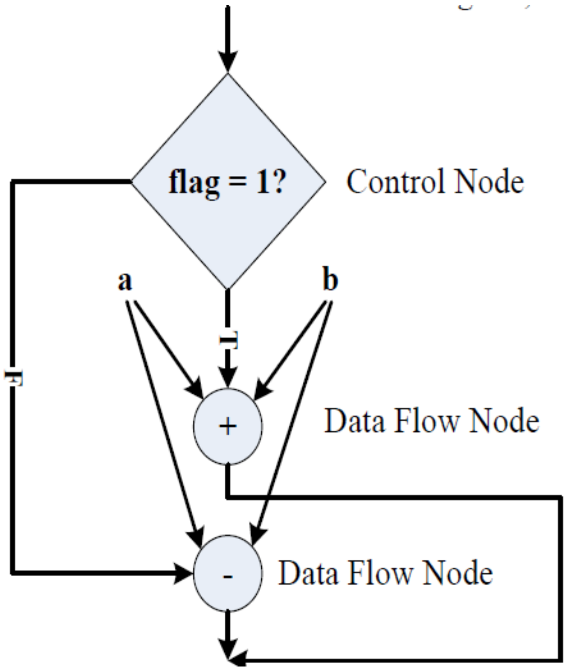


Data path: The data flow path from input to output
A DFG model is said to be acyclic DFG (ADFG) if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s). Feedback inputs (Output is feed back to Input), events etc are examples for non-acyclic inputs. A DFG model translates the program as a single sequential process execution.

Control Data Flow Graph/Diagram (CDFG) Model

- Translates the data processing requirements into a data flow graph
- Model applications involving conditional program execution
- Contains both data operations and control operations
- Uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers.
- CDFG contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes
- A visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.
- The control node is represented by a 'Diamond' block which is the decision making element in a normal flow chart based design
- Translates the requirement, which is modeled to a concurrent process model
- The decision on which process is to be executed is determined by the control node
- Capturing of image and storing it in the format selected (bmp, jpg, tiff, etc.) in a digital camera is a typical example of an application that can be modeled with CDFG

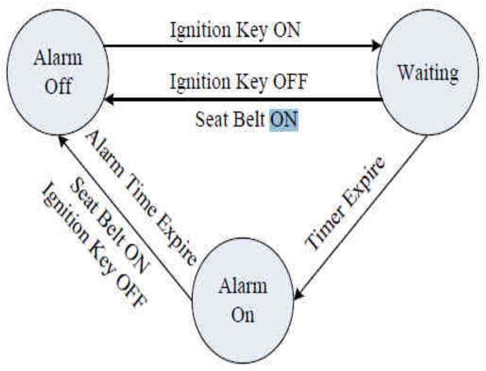
E.g. Model the requirement If flag = 1, x = a + b; else y = a-b;



State Machine Model

- Based on 'States' and 'State Transition'
- Describes the system behavior with 'States', 'Events', 'Actions' and 'Transitions'
- *State* is a representation of a current situation.
- An *event* is an input to the *state*. The *event* acts as stimuli for state transition.
- *Transition* is the movement from one state to another.
- *Action* is an activity to be performed by the state machine.
- A Finite State Machine (FSM) Model is one in which the number of states are finite. In other words the system is described using a finite number of possible states
- Most of the time State Machine model translates the requirements into sequence driven program
- The Hierarchical/Concurrent Finite State Machine Model (HCFSM) is an extension of the FSM for supporting concurrency and hierarchy
- HCFSM extends the conventional state diagrams by the AND, OR decomposition of States together with inter level transitions and a broadcast mechanism for communicating between concurrent processes
- HCFSM uses statecharts for capturing the states, transitions, events and actions. The Harel Statechart, UML State diagram etc are examples for popular statecharts used for the HCFSM modeling of embedded systems
- E.g. Automatic 'Seat Belt Warning' in an automotive

Requirements for automatic seat belt warning:



- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
- The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Sequential Program Model

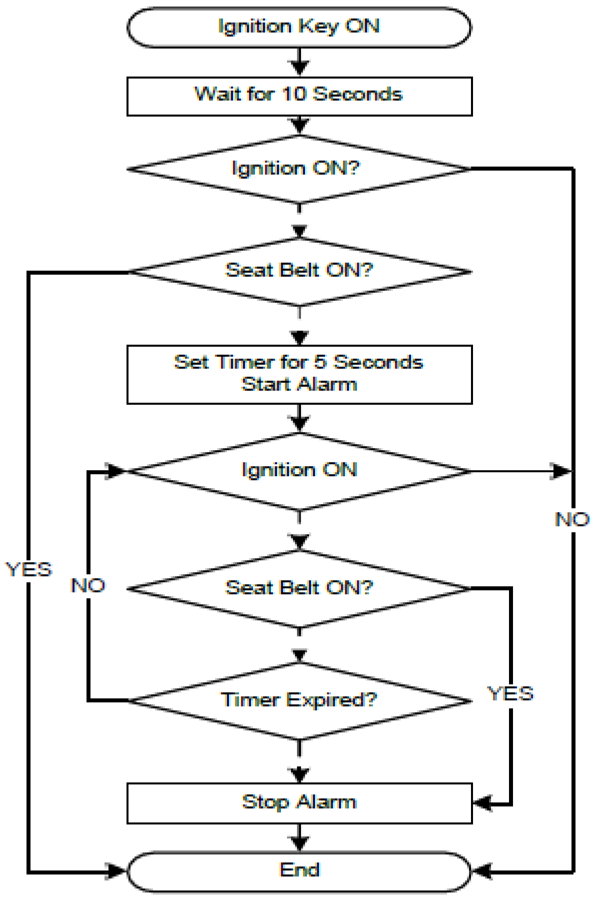
- The functions or processing requirements are executed in sequence
- The program instructions are iterated and executed conditionally and the data gets transformed through a series of operations
- FSMs are good choice for sequential Program modeling.
- Flow Charts is another important tool used for modeling sequential program
- The FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow
- E.g. Automatic 'Seat Belt Warning' in an automotive

Requirement for Seat Belt Warning:

- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
- The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.
E.g. Automatic 'Seat Belt Warning' in an automotive

▪When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds. The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

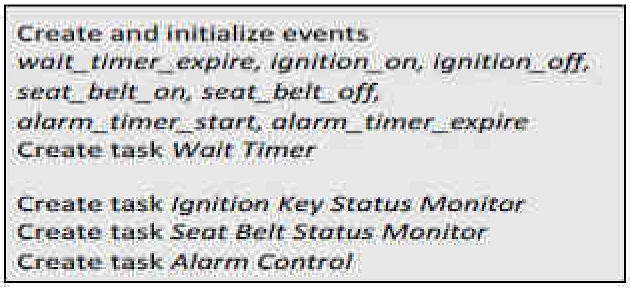
```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn()
{
wait_10sec();
if (check_ignition_key()==ON)
{
if (check_seat_belt()==OFF)
{
set_timer(5);
start_alarm();
while ((check_seat_belt()==OFF )&&(check_ignition_key()==OFF )&&
(timer_expire()==NO));
stop_alarm();
}
}
}
```



Concurrent/Communicating Process Model

- Models concurrently executing tasks/processes. The program instructions are iterated and executed conditionally and the data gets transformed through a series of operations
- Certain processing requirements are easier to model in concurrent processing model than the conventional sequential execution.
- Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilization, when the task involves I/O waiting, sleeping for specified duration etc.
- If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively, when the subtask under execution goes to a wait or sleep mode, by switching the task execution.
- Concurrent processing model requires additional overheads in task scheduling, task synchronization and communication

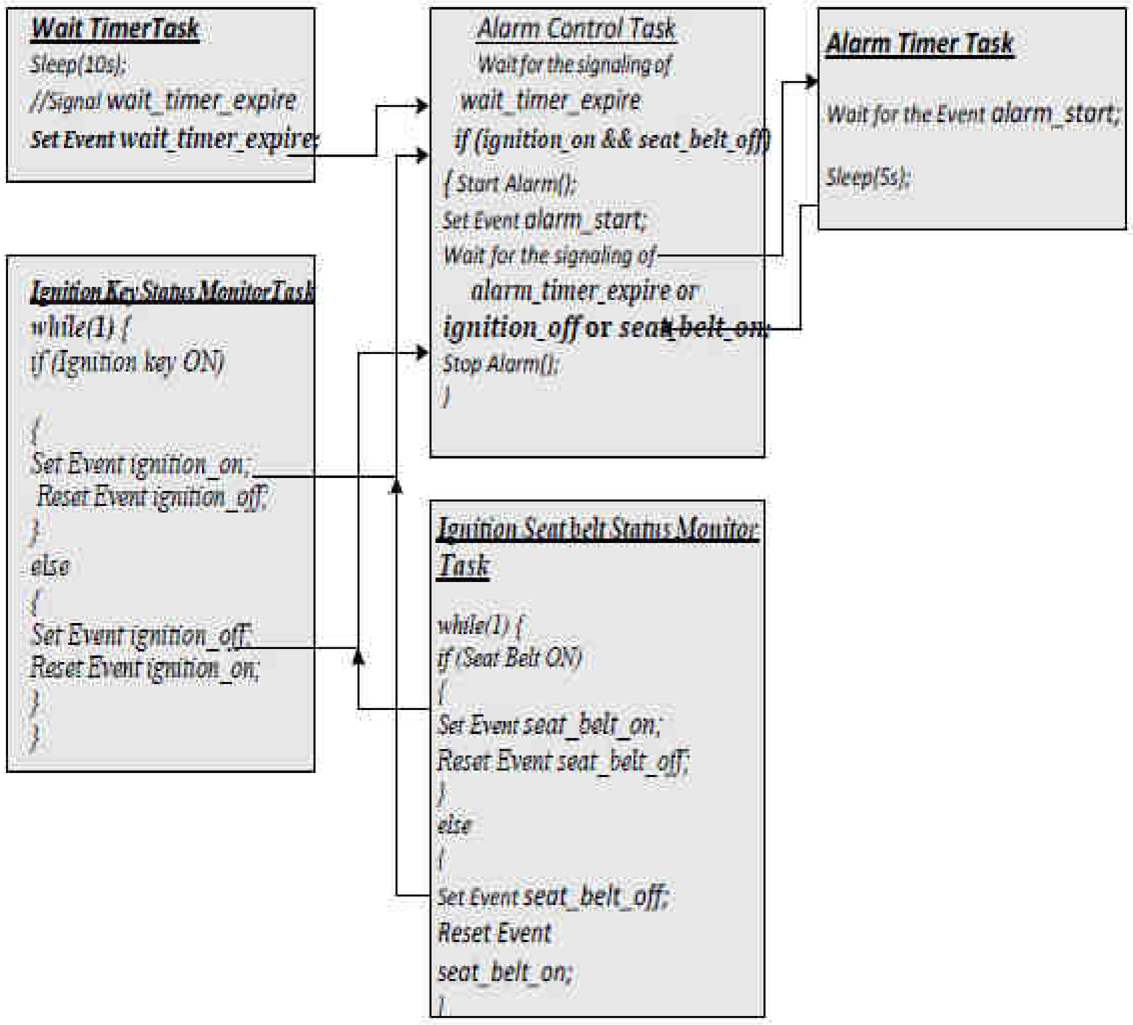
E.g. Automatic 'Seat Belt Warning' in an automotive



```
Create and initialize events
wait_timer_expire, ignition_on, ignition_off,
seat_belt_on, seat_belt_off,
alarm_timer_start, alarm_timer_expire
Create task Wait Timer

Create task Ignition Key Status Monitor
Create task Seat Belt Status Monitor
Create task Alarm Control
```

- The processing requirements are split in to multiple tasks
- Tasks are executed concurrently
- 'Events' are used for synchronizing the execution of tasks



```
Wait TimerTask
Sleep(10s);
//Signal wait_timer_expire
Set Event wait_timer_expire;
```

```
Alarm Control Task
 Wait for the signaling of
 wait_timer_expire
 if (ignition_on && seat_belt_off)
{ Start Alarm();
Set Event alarm_start;
Wait for the signaling of
   alarm_timer_expire or
ignition_off or seat_belt_on;
Stop Alarm();
}
```

```
Alarm Timer Task
Wait for the Event alarm_start;
Sleep(5s);
```

```
Ignition Key Status MonitorTask
while(1) {
if (Ignition key ON)
{
Set Event ignition_on;
 Reset Event ignition_off;
}
else
{
Set Event ignition_off;
Reset Event ignition_on;
}
}
```

```
Ignition Seat belt Status Monitor
Task
while(1) {
if (Seat Belt ON)
{
Set Event seat_belt_on;
Reset Event seat_belt_off;
}
else
{
Set Event seat_belt_off;
Reset Event
seat_belt_on;
}
```

Introduction to Unified Modeling Language (UML)

Unified Modeling Language (UML) is a visual modeling language for Object Oriented Design (OOD). UML helps in all phases of system design through a set of unique diagrams for requirements capturing, designing and deployment.

UML Building Blocks

The following are the main building block of UML diagrams.

1. Things
   An abstraction of the UML Model

2. Relationships
   An entity which express the type of relationship between UML elements (objects, classes etc)

3. Diagrams
   UML Diagrams give a pictorial representation of the static aspects, behavioral aspects and organization and management of different modules (classes, packages etc) of the system

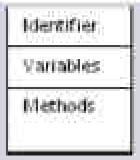
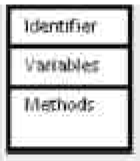
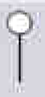
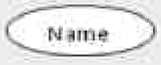
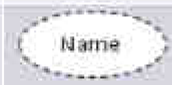
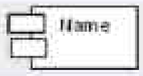
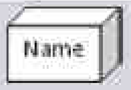
1.Things

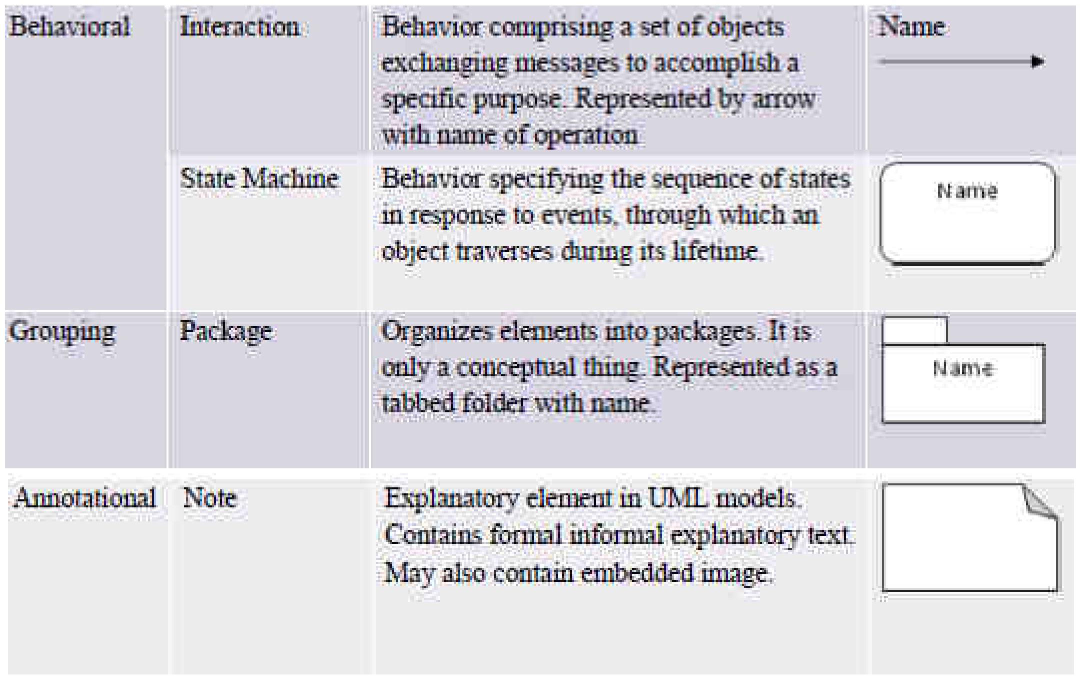
- **Structural things:** Represents mostly the static parts of a UML model. They are also known as 'classifiers'. Class, interface, use case, use case realization (collaboration), active class, component and node are the structural things in UML.

- **Behavioral things:** Represents mostly the dynamic parts of a UML model. Interaction, state machine and activity are the behavioral things in UML.

- **Grouping things:** Are the organizational parts of a UML model. Package and sub-system are the grouping things in UML.

- **Annotational things:** Are the explanatory parts of a UML model. *Note* is the Annotational thing in UML.

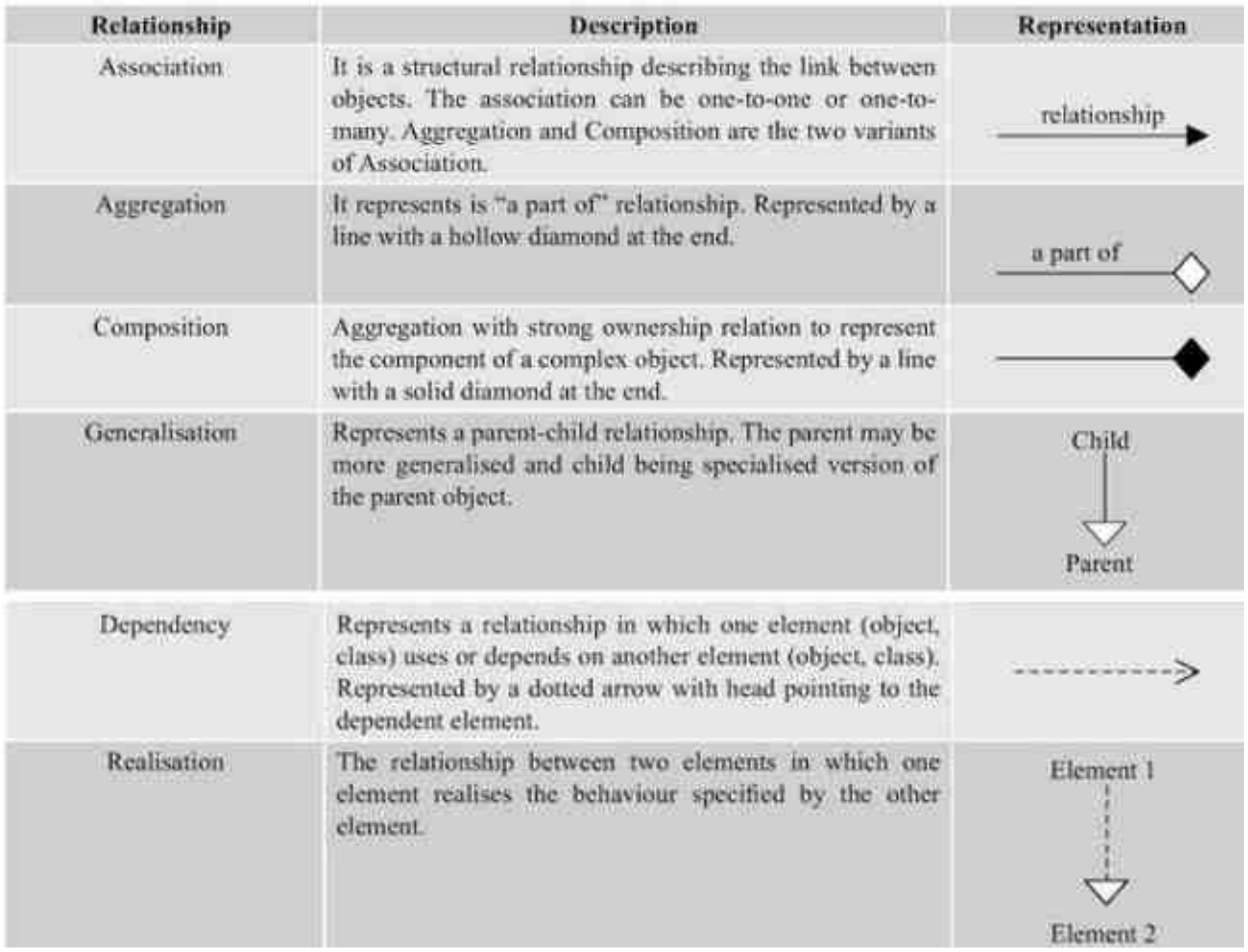
Thing	Element	Description	Representation
Structural	Class	A template describing a set of objects which share the same attributes, relationships, operations and semantics. It can be considered as a blueprint of object.	
	Active Class	Class presenting a thread of control in the system. It can initiate control activity. Active class is represented in the same way as that of a class but with thick border lines.	
	Interface	A collection of externally visible operations which specify a service of a class. It is represented as a circle attached to the class	
	Use case	Defines a set of sequence of actions. It is normally represented with an ellipse indicating the name.	
	Collaboration (Use case Realization)	Interaction diagram specifying the collaboration of different use cases. It is normally represented with a dotted ellipse indicating the name.	
	Component	Physical packaging of classes and interfaces.	
	Node	A computational resource existing at run time. Represented using a cube with name.	



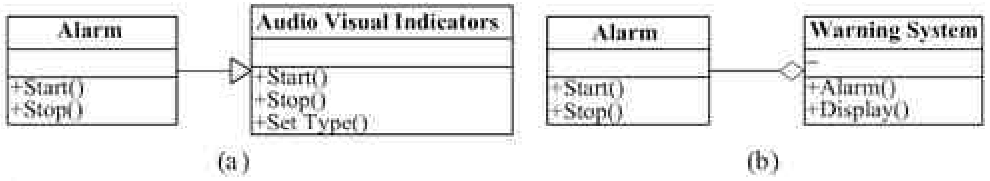
Behavioral	Interaction	Behavior comprising a set of objects exchanging messages to accomplish a specific purpose. Represented by arrow with name of operation	Name $\longrightarrow$
	State Machine	Behavior specifying the sequence of states in response to events, through which an object traverses during its lifetime.	Name
Grouping	Package	Organizes elements into packages. It is only a conceptual thing. Represented as a tabbed folder with name.	Name
Annotational	Note	Explanatory element in UML models. Contains formal informal explanatory text. May also contain embedded image.	

2.Relationships in UML

Expresses the types of relationship between UML objects (objects, classes etc)

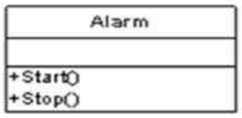


Relationship	Description	Representation
Association	It is a structural relationship describing the link between objects. The association can be one-to-one or one-to-many. Aggregation and Composition are the two variants of Association.	relationship
Aggregation	It represents is "a part of" relationship. Represented by a line with a hollow diamond at the end.	a part of
Composition	Aggregation with strong ownership relation to represent the component of a complex object. Represented by a line with a solid diamond at the end.	
Generalisation	Represents a parent-child relationship. The parent may be more generalised and child being specialised version of the parent object.	Child Parent
Dependency	Represents a relationship in which one element (object, class) uses or depends on another element (object, class). Represented by a dotted arrow with head pointing to the dependent element.	
Realisation	The relationship between two elements in which one element realises the behaviour specified by the other element.	Element 1 Element 2



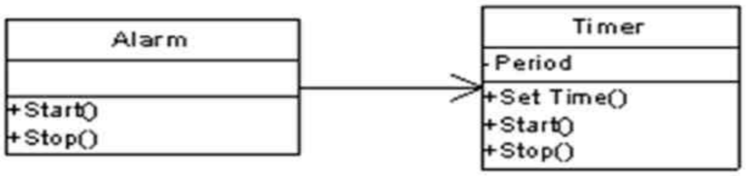
(a) Alarm is a special type of Audio Visual Indicator (Generalisation), (b) Alarm is a part of Warning System (Aggregation)

The Seat Belt Warning System – UML modeling



The Alarm Class

State Representation for 'Alarm ON' state



Alarm – Timer Class interaction for the Seat belt Warning System

3.UML diagrams

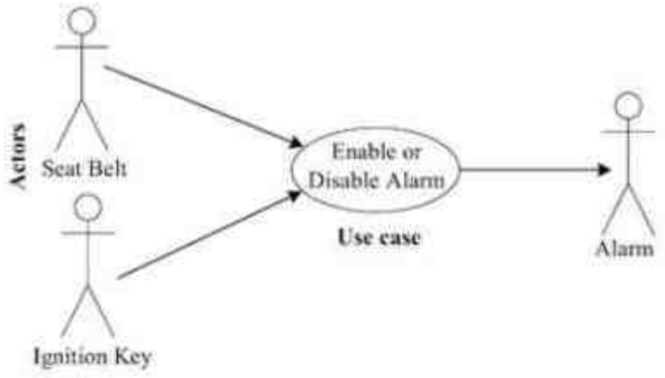
UML diagrams give pictorial representation of static aspects, behavioral aspects and organization and management of different modules(into classes, packages) of the systems

Static Diagrams: Diagram representing the static (structural) aspects of the system. Class Diagram, Object Diagram, Component Diagram, Package Diagram, Composite Structure Diagram and Deployment Diagram falls under this category

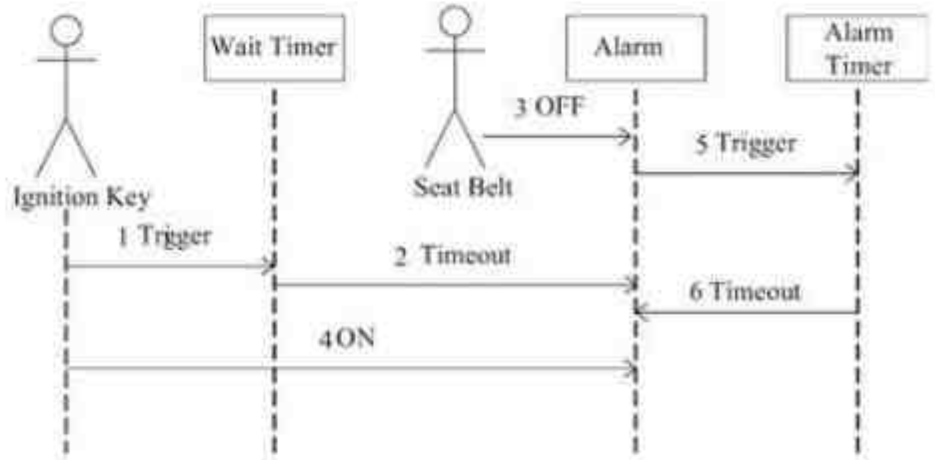
Diagram	Description
Object Diagram	Gives a pictorial representation of a set of objects and their relationships. It represents the structural organization between objects.
Class Diagram	Gives a pictorial representation of the different classes in a UML model, their interfaces, the collaborations, interactions and relationship between the classes etc. It captures the static design of the system.
Component Diagram	It is a pictorial representation of the implementation view of a system. It comprises of components (Physical packaging of classes and interfaces), relationships and associations among the components.
Package Diagram	It is a representation of the organization of packages and their elements. Package diagrams are mostly used for organizing use case diagrams and class diagrams.
Deployment Diagram	It is a pictorial representation of the configuration of run time processing nodes and the components associated with them.

Behavioral Diagram

It represents the dynamic(behavioral) aspects of the system.Use case diagram,sequence diagram,state diagram,communication diagram,activity diagram,timing diagram and interaction diagram are diagrams in UML



Use case diagram for seat belt warning system



Sequence diagram for seat belt warning system

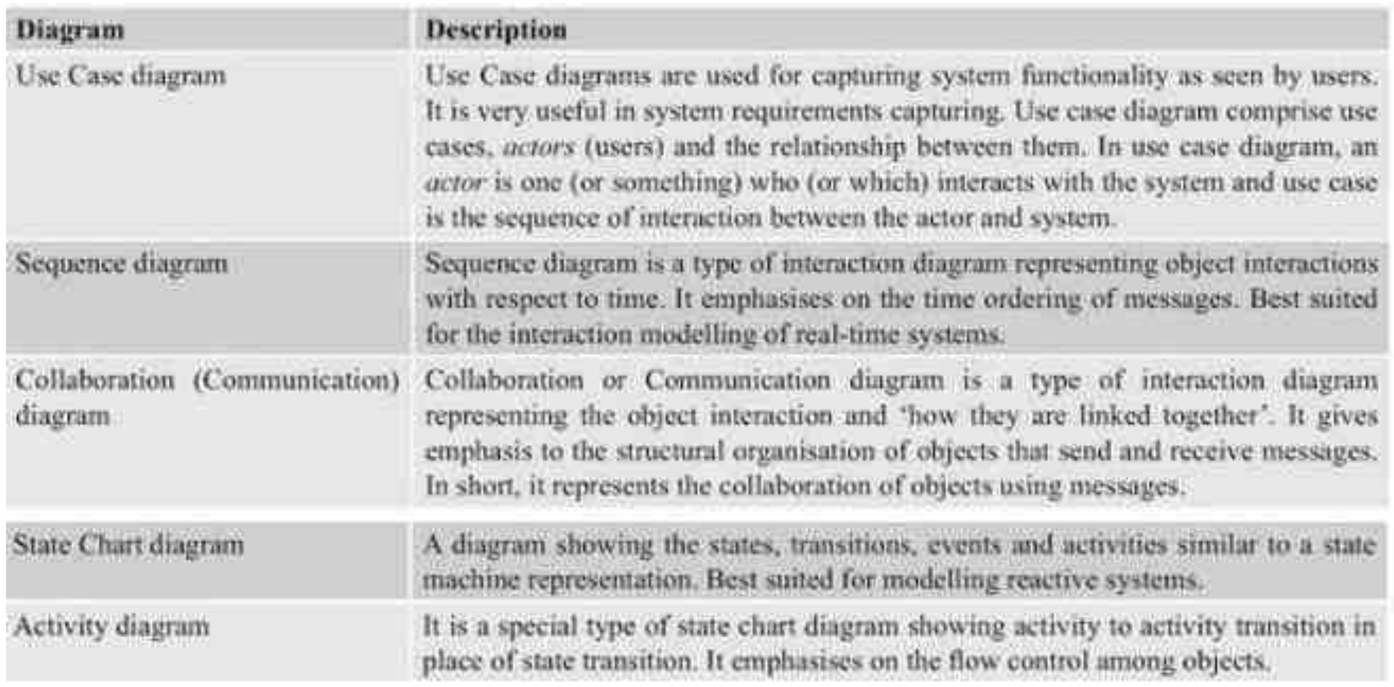


Diagram	Description
Use Case diagram	Use Case diagrams are used for capturing system functionality as seen by users. It is very useful in system requirements capturing. Use case diagram comprise use cases, *actors* (users) and the relationship between them. In use case diagram, an *actor* is one (or something) who (or which) interacts with the system and use case is the sequence of interaction between the actor and system.
Sequence diagram	Sequence diagram is a type of interaction diagram representing object interactions with respect to time. It emphasises on the time ordering of messages. Best suited for the interaction modelling of real-time systems.
Collaboration (Communication) diagram	Collaboration or Communication diagram is a type of interaction diagram representing the object interaction and 'how they are linked together'. It gives emphasis to the structural organisation of objects that send and receive messages. In short, it represents the collaboration of objects using messages.
State Chart diagram	A diagram showing the states, transitions, events and activities similar to a state machine representation. Best suited for modelling reactive systems.
Activity diagram	It is a special type of state chart diagram showing activity to activity transition in place of state transition. It emphasises on the flow control among objects.

Hardware Software Trade-offs

- Certain system level processing requirements may be possible to develop in either hardware or software
- The decision on which one to opt is based on the trade-offs and actual system requirement
- Processing speed, performance, memory requirements, effort, re-usability etc. are examples for some of the hardware software trade-offs in the partitioning of a system requirement

MODULE 5

The **real-time operating system** used for a real-time application means for those applications where data processing should be done in the fixed and small quantum of time. It is different from general purpose computer where time concept is not considered as much crucial as in Real-Time Operating System. RTOS is a time-sharing system based on clock interrupts. Interrupt Service Routine (ISR) serve the interrupt, raised by the system. RTOS used Priority to execute the process. When a high priority process enters in system low priority process preempted to serve higher priority process. Real-time operating system synchronized the process. So that they can communicate with each other. Resources can be used efficiently without wastage of time. RTOS are controlling traffic signal; Nuclear reactors Control scientific experiments, medical imaging systems, industrial system, fuel injection system, home appliance are some application of Real Time operating system

Real time Operating Systems are very fast and quick respondent systems. These systems are used in an environment where a large number of events (generally external) must be accepted and processed in a short time. Real time processing requires quick transaction and characterized by supplying immediate response. For example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion.

In real time operating system there is a little swapping of programs between primary and secondary memory. Most of the time, processes remain in primary memory in order to provide quick response, therefore, memory management in real time system is less demanding compared to other systems.

Time Sharing Operating System is based on Event-driven and time-sharing the design.

The event Driven : In event-driven switching, higher priority task requires CPU service first than a lower priority task, known as priority scheduling.

Time Sharing : Switching takes place after fixed time quantum known as Round Robin Scheduling.

In these design, we mainly deal with three states of the process cycle

1) Running : when CPU is executing a process, then it is in running state.

2) Ready : When a process has all the resources require performing a process, but still it is not in running state because of the absence of CPU is known as the Ready state.

3) Blocked : when a process has not all required resources for execution, then it is blocked state.

Interrupt Latency : Interrupt latency is time between an interrupt is generated by a device and till it serviced. In RTOS, Interrupt maintained in a fixed amount of time, i.e., latency time bounded.

Memory Allocation: RTOS support static as well as dynamic memory allocation.Both allocations used for different purpose.Like Static Memory, the allocation is used for compile and design time using stack data structure. Dynamic memory allocation used for runtime used heap data structure.

The primary functions of the real time operating system are to:

1. Manage the processor and other system resources to meet the requirements of an application.
2. Synchronize with and respond to the system events.

3. Move the data efficiently among processes and to perform coordination among these processes.

The Real Time systems are used in the environments where a large number of events (generally external to the computer system) is required to be accepted and is to be processed in the form of quick response. Such systems have to be the multitasking. So the primary function of the real time operating system is to manage certain system resources, such as the CPU, memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system Apart from these primary functions of the real time operating system there are certain secondary functions that are not mandatory but are included to enhance the performance:

1. To provide an efficient management of RAM.
2. To provide an exclusive access to the computer resources.

The term real time refers to the technique of updating files with the transaction data immediately just after the event that it relates with.

Few more examples of real time processing are:

1. Airlines reservation system.
2. Air traffic control system.
3. Systems that provide immediate updating.
4. Systems that provide up to the minute information on stock prices.
5. Defense application systems like as RADAR.

Real time operating systems mostly use the preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled up to a maximum time, set by the time-slice parameter, in a round robin manner. Hundred of the priority levels are commonly available for scheduling. Some specific tasks can also be indicated to be non-preemptive.

Components of real time operating system

A real time operating system contains the following components:

• **The Scheduler**: this element in RTOS tells that in which order the tasks can be executed which is generally based on the priority. It is the main element of RTOS.

• **Symmetric Multiprocessing (SMP)**: a number of different multiple tasks can be handled by the RTOS as this is its ability so that parallel processing can be done. This is also known as multitasking.

• **Function Library**: It is the element of RTOS which acts as a interface so that it can connect kernel and application code. This application code sends the requests to the kernel via function library so that the application can give the desired result.

• **Memory Management**: this element is required so that the system can allocate the memory to every program. It is an important element of the RTOS.

• **Fast dispatch latency or context switch time**: The term dispatch latency means the time interval between the termination of the task which is identified by the operating system and the time taken by the thread, which is in the ready queue, that has started processing.

As its name implies, context switch time is the time which the system takes to switch from one running thread to another. This time saves the context of the current task and also replaces it with

the context of new thread and this time needs to be minimal and optimal for an RTOS.
• **User-defined data objects and classes**: RTOS makes use of the programming languages like C/C++ which are organized according to the type of their operation. RTOS will use them so that it can control the specified application.

Types of Real-Time Operating System

1) Soft Real-Time Operating System : A process might not be executed in given deadline. It can be crossed it then executed next, without harming the system. Example are a digital camera, mobile phones, etc.

2) Hard Real-Time Operating System : A process should be executed in given deadline. The deadline should not be crossed. Preemption time for Hard Real-Time Operating System is almost less than few microseconds.
Examples are Airbag control in cars, anti-lock brake, engine control system, etc.

Applications of Real Time Operating System

• **Control systems**: RTOS are designed in such a way so that they can control actuators and sensors. They can execute control system commands. Controlled systems are those which can be monitored by using sensors and their tasks can be altered with the help of actuators. Now the task of the RTOS is to read the data from the sensors and move the actuators by performing some calculations so that flight's movement can be handled.
• **Image processing (IP)**: real time image processing is done so that we can make some adjustments for the moving objects. In this, we need our computers, cameras or other gadgets should work in real time as utmost precision is required in the industrial automation tasks. For ex, something happens with the conveyor belt when the item is moving downwards or some other defect occurs, we can control these problems in the real time if our system works in real time.
• **Voice Over IP (VoIP)**: we can transmit voice over IPs or internet protocol in real time. This is known as VoIP. In this our voice is digitalized, compressed into small form and converted into IP packets in real time before transmitting it to the other network on router through IP.

Difference between in GPOS and RTOS

General-Purpose Operating System (GPOS)	**Real-Time Operating System (RTOS)**
1) It used for desktop pc, laptop.	1) It applied for the embedded application.
2) Process-based Scheduling used.	2) Time-based scheduling used like round robin.
3) Interrupt latency is not considered as much crucial as in RTOS.	3) Interrupt lag is minimal, measured in few microseconds.
4) No priority inversion mechanism is present in the system.	4) Priority inversion mechanism is current.Once priority set by the programmer, it can't be changed by the system itself.
5) Kernel operations may or may not be preempted.	5) Kernel operation can be preempted.

Disadvantages of Real Time Operating System

• **Limited tasks**: RTOS can run very limited tasks simultaneously and it concentrates on only those applications which contain error so that it can avoid them. Due to this, some tasks have to wait for unlimited time.
• **Low multi-tasking**: As RTOS is the system which concentrates on few tasks. So sometimes it is difficult for these systems to do multi-tasking.
• **Device driver and interrupt signals**: Some specific drivers are needed for the RTOS so that it can give fast response to the interrupt signals to maintain its speed.
• **Expensive**: As stated earlier, a lot of resources are used by RTOS which makes it expensive.
• **Low priority of tasks**: The tasks which has the low priority has to wait for a long time as the RTOS maintain the accuracy of the current programs which are under execution.
• **Thread priority**: as we know that there is very less switching of tasks is done in RTOS. So that's why its thread priority is not good.
• **Precision of code**: as we know, in RTOS, event handling of tasks is a strict process as it is error-free. So more precision in code is required which is a difficult task for the programmer as getting exact precision is not so easy.
• **Complex algorithms**: it makes use of some complex algorithms so that the system can give the desired output. These complex algorithms are difficult to understand.
• **Use heavy system resources**: real time operating system makes use of a lot of resources. Sometimes it is not good for the system. It seems to be expensive also.
• **Not easy to program**: Sometime the programmer has to write the complex programs for the efficient working of the RTOS which is not an easy task.
• **Other factors**: some other factors like error handling, memory management and CPU are also needed to be considered.

OS Services

- OS Goal
- Processor modes(User & Supervisory)
- System structure
- Kernel

OS Goal

- Easy sharing of resources as per schedule and allocations
- Easy implementation
- Scheduling
- Management of the processes
- Subsystems and protocols(Files ,I/O and network)
- Portability
- Inter-opearbility
- Common set of interfaces
- GUIs and APIs
- Maximizing the system performance

Mode structure

- User mode : User function call, which is not a system call, is not permitted to read and write into the protected memory allotted to the OS functions ,data and heap.

- Supervisory mode:The OS runs the privileged functions and instructions in protected mode and the OS only accesses the hardware resources and protected area memory.Only a system call is permitted to access protected memory

System structure

- Application software
- Application Programming Interface(API)
- System software other than one provided at the OS
- OS interface
- OS
- Hardware-OS interface
- Hardware

Kernel

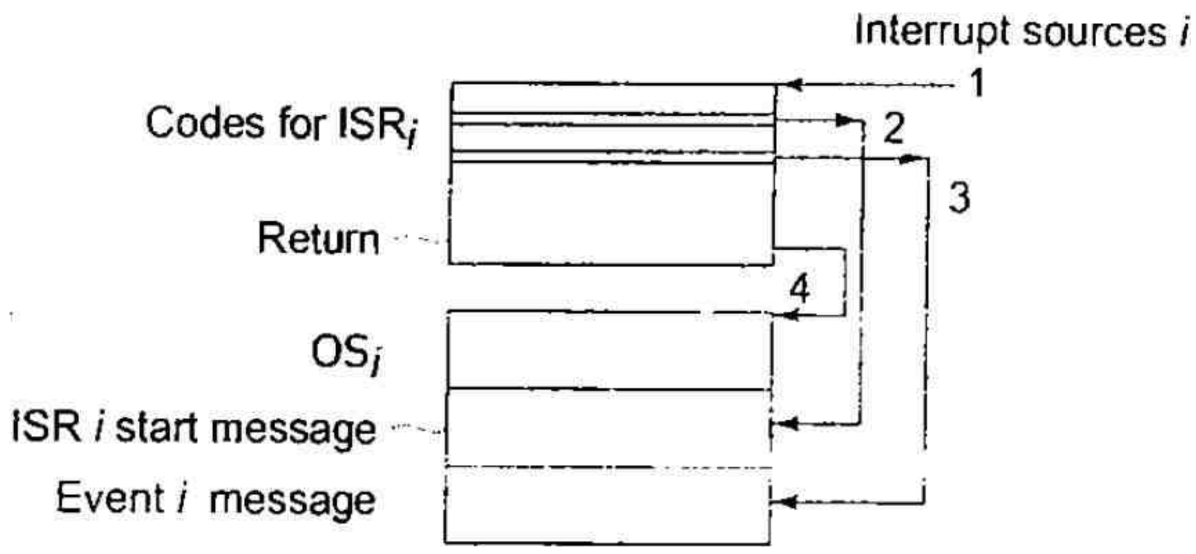
- An os includes
  1. A kernel with device management and file management as a part of kernel in the given OS
  2. Kernel without device management and file management
- Functions of the kernel
  1. Process management
  2. Memory management
  3. File management
  4. Device management and device drivers
  5. I/O sub system management

Interrupt handling in RTOS environment

ISRs in RTOSes

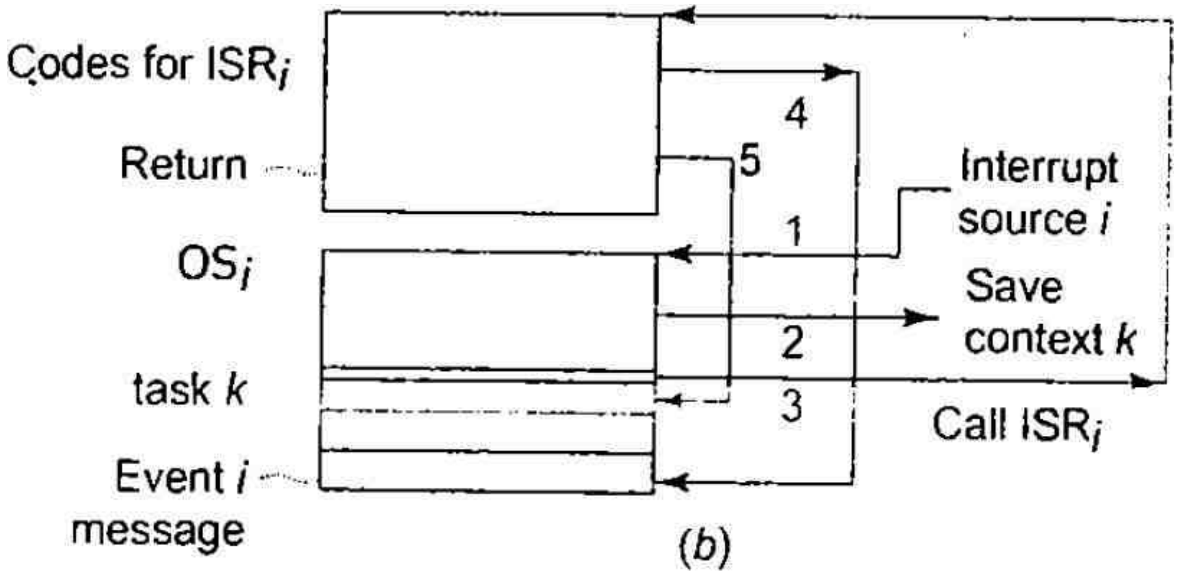
ISRs have the higher priorities over the RTOS functions and the tasks. An ISR should not wait for a semaphore, mailbox message or queue message An ISR should not also wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run. Only the IPC accept function for these events (semaphore, mailbox, queue) can be used, not the post function

1. Direct call to ISR by interrupting source
2. RTOS first interrupted on an interrupt then OS calling ISR
3. RTOS first interrupted on an interrupt then OS calling a fast ISR and the fast ISR calls a slow ISR(IST)



(1)

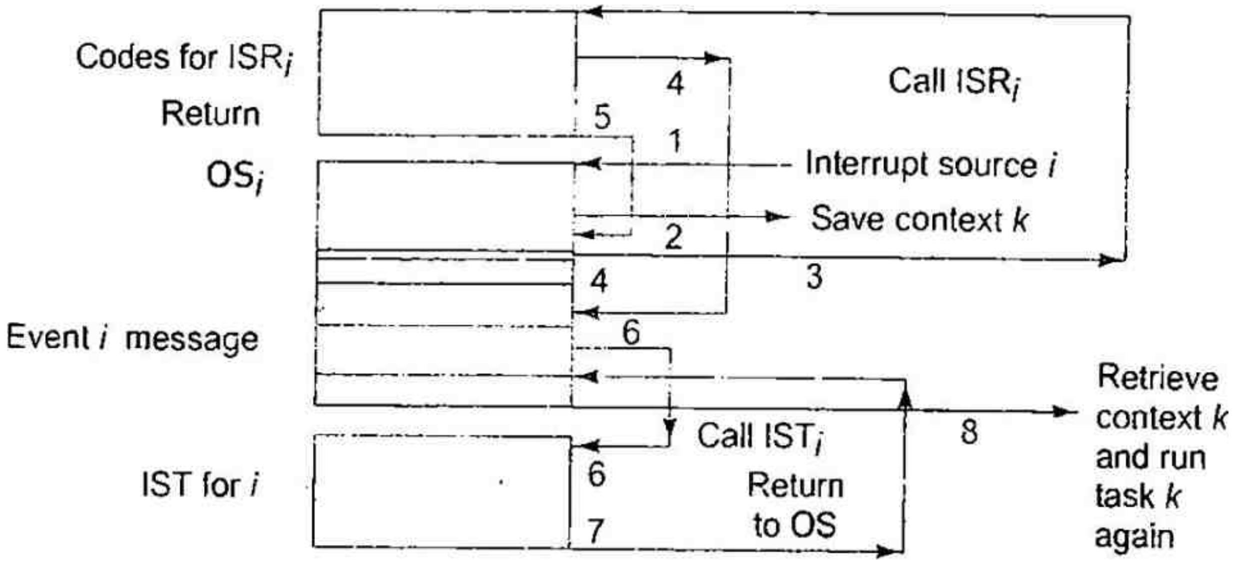
1. **Direct call to ISR by interrupting source**
   - On an interrupt, the process running at the CPU is interrupted
   - ISR corresponding to that source starts executing.
   - A hardware source calls an ISR directly.
   - The ISR just sends an ISR enter message to the RTOS. ISR enter message is to inform the RTOS that an ISR has taken control of the CPU.
   - ISR IPC messages and Exit message
   - ISR code can send into a mailbox or message queue but the task waiting for a mailbox or message queue does not start before the return from the ISR
   - When ISR finishes, it send s Exit message to OS
   - On return from ISR by retrieving saved context, The RTOS later on returns to the interrupted process (task) or reschedules the processes (tasks).
   - RTOS action depends on the event-messages, whether the task waiting for the event message from the ISR is a task of higher priority than the interrupted task on the interrupt



2. RTOS first interrupted on an interrupt then OS calling ISR

- On interrupt of a task, say, k-th task, the RTOS first gets itself the hardware source call and initiates the corresponding ISR after saving the present processor status (or context)
- Then the ISR during execution then can post one or more outputs for the events and messages into the mailboxes or queues.
- The ISR must be short and it must simply puts post the messages for another task.
- This task runs the remaining codes whenever it is scheduled.
- RTOS schedules only the tasks (processes) and switches the contexts between the tasks only.
- ISR executes only during a temporary suspension of a task.

(2)



(3)

3. **RTOS first interrupted on an interrupt then OS calling a fast ISR and the fast ISR calls a slow ISR(IST)**

- An RTOS can provide for two levels of interrupt service routines, a fast 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 Windows CE The SLISR is called interrupt service thread (IST) in Windows CE.
- 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 functions as deferred procedure call (DPC) of the ISR. An i-th interrupt service thread (IST) is a thread to service an i-th interrupt source call.

RTOS Design Principles

1. Design with ISR and task
2. RTOS provides nesting of ISRs
3. A task can wait and take the messages
4. ISR code should be optimally short execution time
5. Task-modular design
6. Data encapsulation
7. Limit the number of tasks and select the appropriate number of tasks
8. Design with taking care of memory and power needs
9. Design with taking care of the time spend in the system calls
10. Use appropriate precedence assignment strategy and use of preemption in place of time slicing
11. Avoid task deletion
12. Use idle CPU time for internal functions
13. Design with memory allocation and de-allocation by tasks
14. Design with control on latency periods
15. Design with limited RTOS functions

Task scheduling models

- Cooperative scheduling model
- Cooperative with precedence constraints
- Cyclic and round robin with time slicing scheduling models
- Preemptive scheduling model
- Scheduling using Earliest deadline first(EDF) precedence
- Rate monotonic schedulers (RMS) using higher rate of events occurrences first precedence
- Fixed (static) time scheduling models
- Scheduling of periodic, sporadic and aperiodic tasks
- Advanced scheduling algorithm using the probabilistic time Petri nets (stochastic) or multithread graphs. For multiprocessor and distributed systems

How to Choose an RTOS

Functional

1. Processor support
2. Memory requirement
3. Real time capabilities
4. Kernel and interrupt latency
5. IPC and task synchronization
6. Modularization support
7. Support for networking and communication
8. Development language support

Non Functional

1. Custom developed or off the shelf
2. Cost
3. Development and debugging tools availability
4. Ease of use
5. After sales

Case Study – MicroC/OS-II

- MicroC/OS-II is a portable, ROMable, scalable, preemptive, real-time, multi-tasking, priority-based OS.
- Open source ANSI C and free for academic use.
- Was ported to 40+ architectures (8 to 64 bit) since 1992.
- Written by Jean J. Labrosse of Micrium, http://ucos-ii.com

MicroC/OS-II

- Task creation and Management
- Kernal Functions and initialization
- Task scheduling
- Inter-task communication
- Mutual exclusion and task synchronization
- Timing and reference
- Memory management
- Interrupt handling

Task creation and Management

- Five possible states for a task to be in
  - Dormant – not yet visible to OS (use OSTaskCreate(), etc.)
  - Ready
  - Running
  - Waiting
  - ISR – preempted by an ISR
- A task is usually creates as a non ending function

```
//creates a task with name MicroCtask
Void MicroCTask(void *arg)
{
        While(1)
        {
                //code corresponds to task
        }
}
```

- Need to pass this task to MicroC kernel to make ready for execution (supports upto 64 tasks)
- OSTaskCreate() in os_task.c
    - Create a task
    - Arguments: pointer to task code (function), pointer to argument, pointer to top of stack , desired priority (unique)
    - OSTaskCreateEx() in os_task.c
        - *Create a task*
        - *Arguments: same as for OSTaskCreate(), plus*
        - *id: user-specified unique task identifier number*
        - *pbos: pointer to bottom of stack. Used for stack checking (if enabled).*
        - *stk_size: number of elements in stack. Used for stack checking (if enabled).*
        - *pext: pointer to user-supplied task-specific data area (e.g. string with task name)*
        - *opt: options to control how task is created.*
- OSTaskSuspend()
    - Task will not run again until after it is resumed ; kernel reschedules the next highest priority for execution
    - Sets OS_STAT_SUSPEND flag, removes task from ready list if there
    - Argument: Task priority (used to identify task)
- OSTaskResume()
    - Task will run again once any time delay expires and task is in ready queue
    - Clears OS_STAT_SUSPEND flag
    - Argument: Task priority (used to identify task)
- OSTaskDelRequest()
- OSTaskDel()
    - Sets task to DORMANT state, so no longer scheduled by OS
    - Removed from OS data structures: ready list, wait lists for semaphores/mailboxes/queues, etc.
- OSTaskChangePrio()
    - Identify task by (current) priority
    - Changes task's priority
- OSTaskQuery()
    - Identify task by priority

        - Copies that task's TCB into a user-supplied structure
        - Useful for debugging
- OSTaskStkCheck()
    - Checks stack overflow for creation of tasks
    - OSTasknameGet()
        - Returns the name associated with a task
    - OSTaskNameSet()

Application requested delays

- OSTimeDly()/OSTimeDlyHMSM()
    - Task A calls OSTimeDly or OSTimeDlyHMSM() in os_time.c
    - TCB->OSTCBDly set to indicate number of ticks to wait;usually-2^{16} ticks
    - Otherwise H(0-255),M(0-59),S(0-590),M(0-999)
    - Note that OSTickISR() in os_cpu_a.a30, OSTimeTick() in os_core2.c decrement this field and determine when it expires
- OSTimeDlyResume()
    - Task B can resume Task A by calling OSTimeDlyResume()
    - Resumes a task which is delayed by a call to either OSTimeDly()/OSTimeDlyHMSM()
- **Tasks example**
    - Task 1
        - Flashes red LED
        - Displays count of loop iterations on LCD top line
    - Task 2
        - Flashes green LED
    - Task 3
        - Flashes yellow LED

Kernel Functions and initialization

- Kernel needs to be initialized and started before executing the user tasks
- OS kernel initialization function OSInit() is executed first. It performs
    - Initialize the different OS kernel data structure
    - Creates the idle task OSIdle() –low priority
- Code snippet for OS initialization and start operations

```
#include<includes.h>
//main task
Void main(void)
{
//initializes the kernel
OsInit(); //create the first task .
OSStart(); //Start OS by executing the highest priority task
}
```

Task scheduling

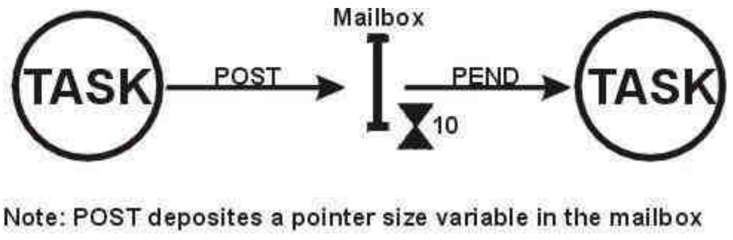
- MicroC/OS-II supports preemptive priority based scheduling (64 level 4-60 based)

- When the kernel starts ,it first executes the highest priority task
- MicroC/OS-II supports preemptive priority based scheduling (64 level 4-60 based)
- When the kernel starts ,it first executes the highest priority task
- A task rescheduling happens when
    - Whenever a higher priority task becomes ‘ready’ or when task enters waitng state due to OsTaskSuspend() ,OSTaskResume(),OSTaskDel(),OSTimeDly() etc
    - Whenever a interrupt occurs ,the task are rescheduled upon return from the ISR
- Task level scheduling is executed by the kernel service OS_Shed() and ISR level by OSIntExit()
- Scheduler runs highest-priority task using OSSched()
    - OSRdyTbl has a set bit for each ready task
    - Checks to see if context switch is needed
    - Macro OS_TASK_SW performs context switch
        - *Implemented as software interrupt which points to OSCtxSw*
        - *Save registers of task being switched out*
        - *Restore registers of task being switched in*
- Scheduler locking
    - Can lock scheduler to prevent other tasks from running (ISRs can still run)
        - *OSSchedLock()*
        - *OSSchedUnlock()*
    - Nesting of OSSchedLock possible
    - Don’t lock the scheduler and then perform a system call which could put your task into the WAITING state!
- Idle task
    - Runs when nothing else is ready
    - Automatically has prioirty OS_LOWEST_PRIO
    - Only increments a counter for use in estimating processor idle time
- Selecting a thread to run
    - OSSched() in os_core2.c
- Context switching
    - OS_TASK_SW in os_cpu.h
    - OSCtxSw in os_cpu_a.a30
- What runs if no tasks are ready?
    - OSTaskIdle() in os_core2.c

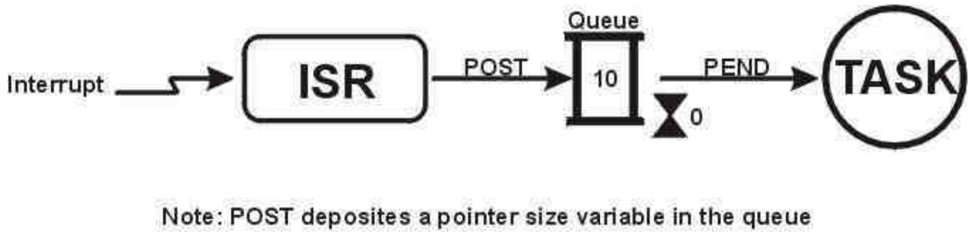
Inter-task communication

MicroC/OS-II supports following IPC techniques for data sharing and co-operation among tasks

- Message Mailboxes
    - For passing of pointers
    - *Create, pend, post, post_opt, accept, query*
- Message Queues
    - For queuing pointer size messages
    - *Create, del, pend, post, post_front, post_opt, accept, flush, query*



Message Mailbox



Message Queue

Message Mailboxes

- OSMboxCreate()
- OSMboxPost()/OSMboxPostOpt()
  - A message is poseted in the mailbox
  - If there is already a message is there in the mailbox,an error is returned(not overwritten)
  - If tasks waiting for a message from the mailbox,the task with highest priority is removed from the wait list and scheduled to run
- OSMboxPend()
  - Used for retrieving messages from mailbox.
  - If mailbox is empty ,the task is immediately blocked and moved to the wait list
  - A time-out value can be specified

Message Mailboxes

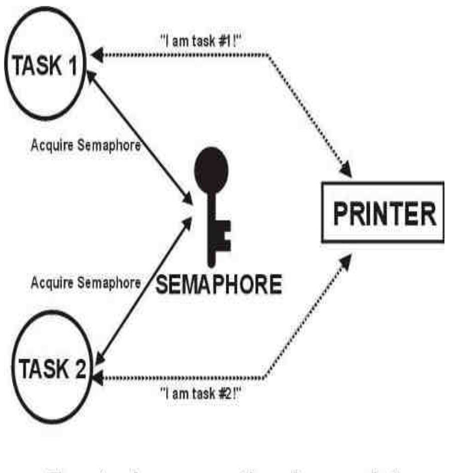
- OSMboxAccept()
  - Its also used for retrieving message from the mailbox
  - The calling task is not blocked('pended') if there is no message to read in the mailbox
- OSMboxQuery()
  - Retrieving information(whether message is there in inbox,is there any task is waiting ,how many tasks etc) from mailbox
- OSMboxDel()

Message Queue

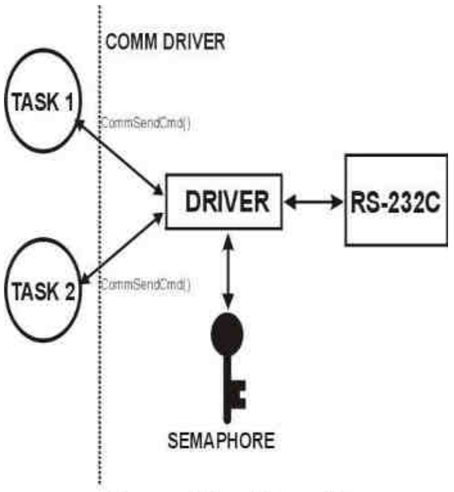
- OSQCreate()
- OSQPost()/OSQPostFront()
- OSQPostOpt()
- OSQPend()
- OSQAccept()
- OSQQuery()
- OSQFlush()
- OSQDel()

Mutual exclusion and task synchronization

- In multitasking system, multiple tasks executes concurrently and share the system resources and data
- The access to shared resources should be made mutually exclusive to prevent ata corruption and race conditions
- Task synchronization can be used by the tasks to synchronize their execution
- Microc/OS-|| provides access to shared resources through the different stept
    1. Disabling task scheduling
    2. Disabling interrupts
    3. Semaphores for mutual exclusion
       Binary and Counting Semaphores
       For Event signaling, control access to shared resource, and synchronization
       accept, create, del, pend, post, query
       Mutual Exclusion Semaphores
       For controlling access to a shared resource (with priority inheritance)
       accept, create, del, pend, post, query
    4. Events (flags) for synchronization



Two tasks competing for a printer



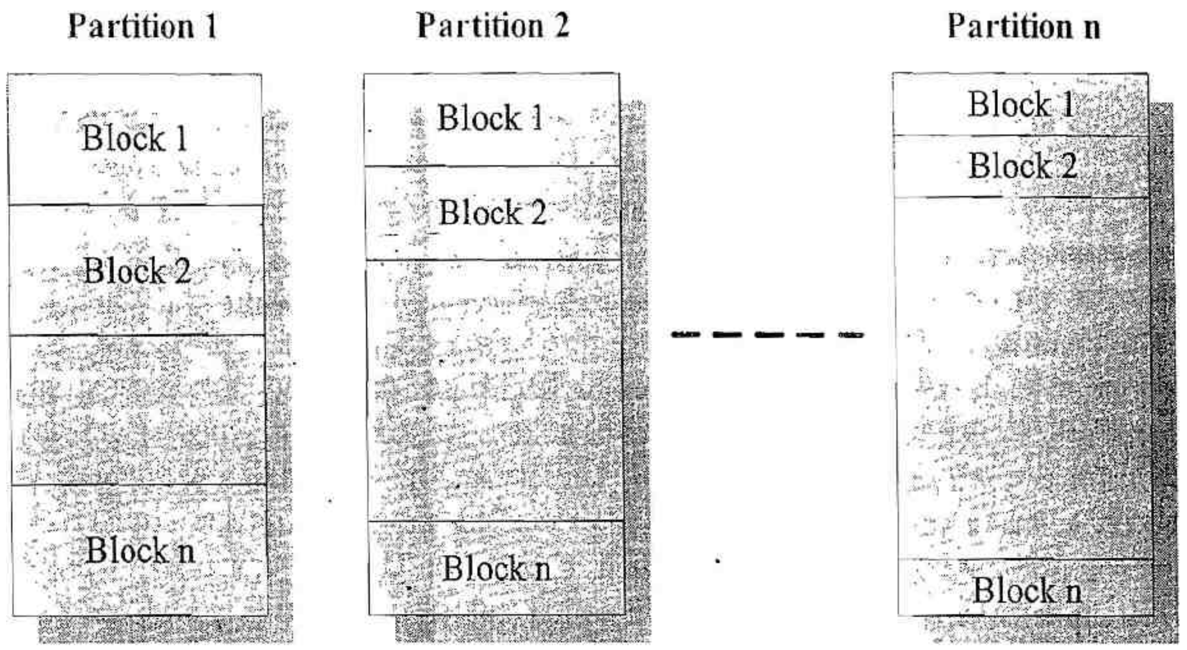
Comm Driver Example

Timing and reference

- The ‘clock tick’ acts as the time source for providing timing reference for time delays and timeouts
- Its generates periodic interrupts
    - OSClockTick, 10 – 100 Hz, increments a 32 bit counter
- OSTimeDly()
- OSTimeDlyHMSM()
- OSTimeDlyResume()
- OSTimeGet() -counter
- OSTimeDlySet()

Memory management

- Alternative to malloc() and free()
- Memory is divided into multiple sectors(partitions)
- Partitions memory into blocks of equal sizes
- OSMemCreate(), get, put, query, nameget, nameset



Memory partitioning under Micro C/OS-II

Interrupt handling

1. Structure needed

        Save CPU registers – NC30 compiler adds this automatically
        Call OSIntEnter() or increment OSIntNesting (faster, so preferred)
            OSIntEnter uses OS_ENTER_CRITICAL and OS_EXIT_CRITICAL, so make sure these use method 2 (save on stack)
        Execute code to service interrupt – body of ISR
        Call OSIntExit()

        - Has OS find the highest priority task to run after this ISR finishes (like OSSched())
    - Restore CPU registers – compiler adds this automatically
    - Execute return from interrupt instruction – compiler adds this automatically
2. Good practices
    - Make ISR as quick as possible. Only do time-critical work here, and defer remaining work to task code.
    - Have ISR notify task of event, possibly send data
        - OSSemPost – raise flag indicating event happened
        - OSMboxPost – put message with data in mailbox (1)
        - OSQPost – put message with data in queue (n)
            - Example: Unload data from UART receive buffer (overflows with 2 characters), put into a longer queue (e.g. overflows after 128 characters) which is serviced by task