MODULE-3

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

1. Firmware Design and Development

1.1- 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.
- 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.
- 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.
- 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.

1.2-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.
  - Conventional Procedure based Firmware Design/Super Loop Design.
  - Embedded Operating System Based Design.

a) 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.
- True procedural one.
    - In multiple task based systems, each task executed in serial.

- Firmware execution flow of this will be:
    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.
    5. ….
    6. Execute the last defined task
    7. 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-cost embedded products where the response time is not time critical.
- Some embedded products demand this type of approach if some tasks itself are sequential.
- 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, it won't create any critical issue.
  -Rather it will treated as a bug in the firmware.

<u>Drawback of Super Loop based Design:</u>

- 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.

– 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.

b) EMBEDDED OPERATING SYSTEM BASED APPROACH

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

2-Embedded Firmware DevelopmentLanguages

- 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

a) Assembly language based development

- Assembly language is human readable notation of machine language whereas machine language is a processor understandable language.
- 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.

- The general format of an assembly language instruction is Opcode followed by the Operand
- Opcode tells what to do and Operand gives the information to do the task
- The operand may be single operand, dual operand or more
  - MOV A, #30
  - Here MOV A is the opcode and 30 is Operand
  - Same instruction in machine language like this 01110100 00011110
- 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
  - 00000010 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

- 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
  - 00000010 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
- 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 differs 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  R0, DELAY         ; Decrement R0and loop till R0=0

           RET                     ; return to callingprogram
```

- The assembly program contains 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 assemblyinstruction should be written in a separate line
- More than one ASM code lines are not allowed in a 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  R0, DELAY         ; Decrement R0and loop till R0=0
         RET                     ; return to callingprogram
```

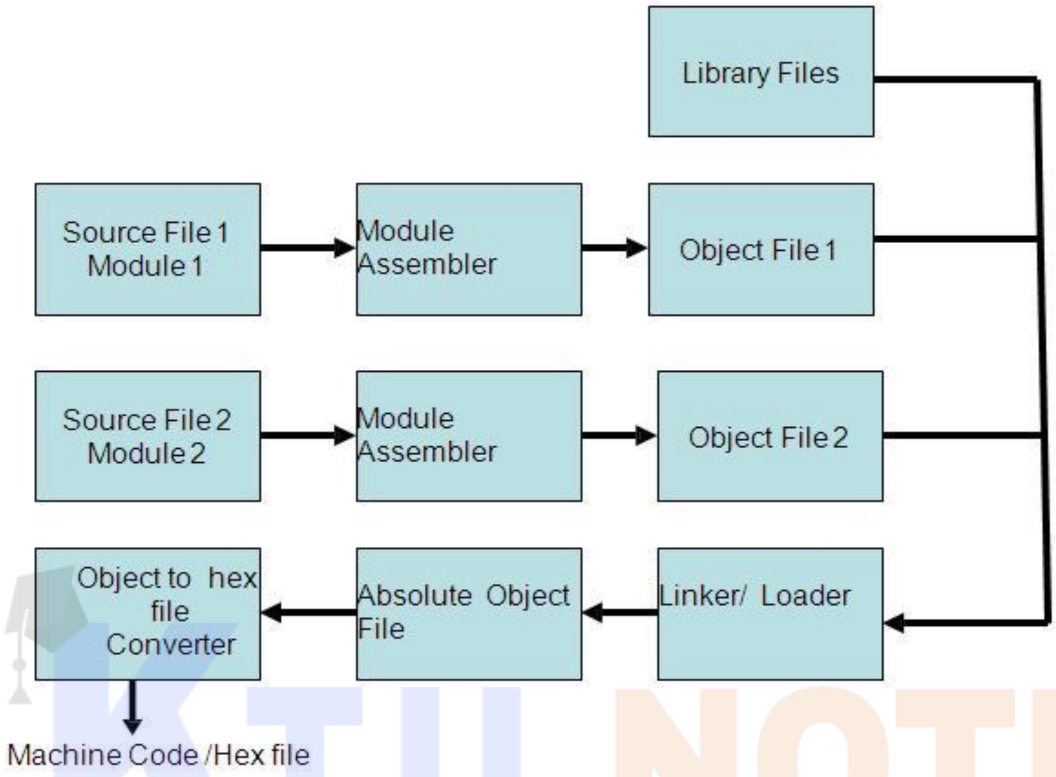
- We can directly replace the LABEL by putting desired address first and then writing assembly code for the routine

```
ORG 0100H
       MOV   R0, #255          ; load Register R0 with 255
       DJNZ  R0, DELAY         ; Decrement R0and loop till R0=0
       RET                     ; return to callingprogram
```

- 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

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

SOURCE FILE TO OBJECTFILE TRANSLATION



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
- For Intel processor the target hex file format will be '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

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
- 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

- 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

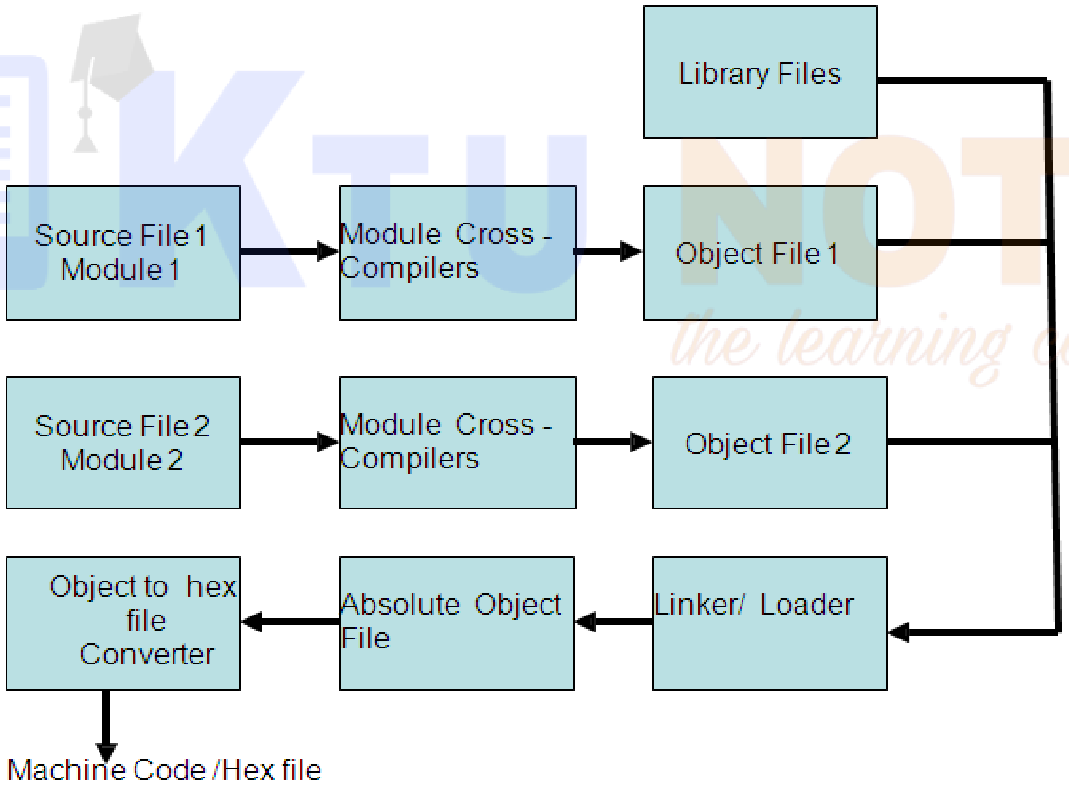
Drawbacks Of Assembly Language Based Development

- **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
  - Solution
    - Use a readily available developer who is well versed in target processor architecture assembly instructions
  - Also more lines of assembly code are required for performing an action which can be done with a single instruction in a high level language like C
- **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
  - If the approach 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 instructed 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
  - Solution
    - Well documentation
- **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

B) High Level Language Based Development

- Any High level language with a 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



- 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

Advantages Of High Level Language Based Development

- 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 assembly language based development
- 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
- 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