MODULE 3

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

- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements
- Firmware is considered as the master brain of the embedded system.
- The embedded firmware imparts intelligence to an Embedded system.
- It is a onetime process and it can happen at any stage.
- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.
- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.
- In case of hardware breakdown , the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.

- The embedded firmware is usually stored in a permanent memory (ROM) and it is non alterable by end users.
- Designing Embedded firmware requires understanding of the particular 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 (either low level Assembly Language or High level language like C/C++ or a combination of the two)
- The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools like UML

- The UML diagrams or flow chart gives a diagrammatic representation of the decision items to be taken and the tasks to be performed.
- Once the program model is created, the next step is the implementation of the tasks and actions by capturing the model using a language which is understandable by the target processor/controller.
- The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of operation required, etc.

EMBEDDED FIRMWARE DESIGN APPROACHES

- Two basic approaches for the design and implementation of embedded firmware
  - 'Conventional Procedural Based Firmware Design'(also known as The Super loop based approach)
  - The Embedded Operating System based approach
- The decision on which approach needs to be adopted for firmware development is purely dependent on the complexity and system requirements

Embedded firmware Design Approaches – The Super loop:

- The Super loop based firmware development approach is Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable).
- It is very similar to a conventional procedural programming where the code is executed task by task
- The tasks are executed in a never ending loop.
- The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task

- A typical super loop implementation will look like:

1. Configure the common parameters 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.

- The ‘C’ program code for the super loop is given below
- void main ()

```
{
Configurations ();
Initializations ();
while (1) {
 Task 1 ();
Task 2 ();
:
:
Task n ();
}
 }
```

- Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation.
- From the above ‘C’ code you can see that the tasks 1 to n are performed one after another and when the last task (nth task) is executed, the firmware execution is again re-directed to Task 1 and it is repeated forever in the loop.
- This repetition is achieved by using an infinite loop.
- Here the while (1) { } loop.
- This approach is also referred as ‘Super loop based Approach’.

- Since the tasks are running inside an infinite loop, the only way to come out of the loop is either a hardware reset or an interrupt assertion.
- A hardware reset brings the program execution back to the main loop.
- Whereas an interrupt request suspends the task execution temporarily and performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted.

Pros:

- **Doesn’t require an Operating System for task scheduling and monitoring and free from OS related overheads**
  - In a super loop based design, the priorities are fixed and the order in which the tasks to be executed are also fixed.
  - Hence the code for performing these tasks will be residing in the code memory without an operating system image.

Contd..

- **This type of design is deployed in low-cost embedded products and products where response time is not time critical**

– Some embedded products demands 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 card, authenticating the operation, reading/writing, etc.

– it should strictly follow a specified sequence and the combination of these series of tasks constitutes a single task-namely data read/ write.

– No use in putting these sub-tasks into independent tasks and running them parallel - it won't work at all.

Pros: Contd..

- A typical **example of a 'Super loop based' product** is an **electronic video game** toy containing keypad and display unit.
- Program reads the keys to detect whether the 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 reasonably high rate.
- Even if the application misses a key press, it won't create any critical issues; rather it will be treated as a bug in the firmware ☺.
- It is **not economical to embed an OS** into low cost products and it is an utter waste to do so **if the response requirements are not crucial**.

- **Simple and straight forward without any OS related overheads**

Cons:

- **Any issues in any task execution may affect the functioning of the product**
  - any failure in any part of a single task may affect the total system.
  - If the program hangs up at some point while executing a task, it may remain there forever and ultimately the product stops functioning.
  - Remedial measures for overcoming this - 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 hangs up.
  - This, in turn, may cause additional hardware cost and firmware overheads.

Cons: Contd..

- **Non Real time in execution behavior**
  - As the number of tasks increases the frequency at which a task gets CPU time for execution also increases
  - Probability of missing out some events.
  - For example in a system with Keypads, according to the 'Super loop design', there will be a 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).
  - In order to identify the key press, you may have to press the keys for a sufficiently long time till the keypad status monitoring task is executed internally by the firmware.
  - This will really lead to the lack of real timeliness.

Cons: Contd..

- Corrective measures for this –
  - use interrupts for external events requiring real time attention.
  - use of low cost high speed processors/controllers in super loop design greatly reduces the time required to service different tasks and thereby are capable of providing a nearly real time attention to external events.

The Embedded Operating System (OS) Based Approach

- The Operating System (OS) based approach contains operating systems, which can be either
  - a General Purpose Operating System (GPOS) or
  - a Real Time Operating System (RTOS)
- to host the user written application firmware.

General Purpose Operating System (GPOS)

- The General Purpose OS (GPOS) based design is **very similar to a conventional PC** based application development where the device contains an operating system (Windows/Unix/ Linux, etc. for Desktop PCs) and you will be creating and running user applications on top of it.
- **Microsoft® Windows XP Embedded 8.1** is an example of GPOS for embedded devices
- Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs , Patient Monitoring Systems etc are **examples of embedded devices running on embedded GPOSs**

Contd

- For Developing applications on top of the OS, the OS supported **APIs** are used.
- Similar to the different hardware specific drivers, OS based applications also require **'Driver software'** for different hardware present on the board to communicate with them.

A Real Time Operating System (RTOS)

- Employed in embedded products demanding Real-time response.
- RTOS respond in a timely and predictable manner to events.
- Real Time operating system contains a Real Time kernel responsible for performing
  - pre-emptive multitasking
  - scheduler for scheduling tasks
  - multiple threads etc.
- A Real Time Operating System (RTOS) allows flexible scheduling of system resources like the CPU and memory and offers some way to communicate between tasks.

- Examples of RTOS employed in embedded product development.
  - Windows Embedded Compact
  - Psos
  - VxWorks
  - ThreadX
  - MicroC/OS-III
  - Embedded Linux
  - Symbian
- Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on RTOSs

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

- Assembly Language based Development
- High Level Language Based Development
- Mixing Assembly and High Level Language
  - Mixing High Level Language (Like C) with Assembly Code
  - Mixing Assembly code with High Level Language (Like C)
  - Inline Assembly

Assembly Language Based Development

- Assembly Language is the human readable notation of ‘machine language’
- Machine language is a processor understandable language
- Processors deal only with binaries (1s and 0s).
- Machine language is a binary representation and it consists of 1s and 0s
- Machine language is made readable by using specific symbols called ‘ mnemonics’.
- Assembly language and machine languages are processor/controller dependent - an Assembly language program written for one processor/controller family will not work with others

- Assembly language programming is the process of writing processor specific machine code in mnemonic 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 an Opcode followed by Operands
- The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode
- It is not necessary that all opcode should have Operands following them.

- Some of the Opcode implicitly contains the operand and in such situation no operand is required.
- The mnemonic INC A is an example for **instruction holding operand implicitly in the Opcode**.
- The machine language representation of the same is 00000100.
- This instruction increments the 8051 Accumulator register content by 1.
- The operand may be a single operand, dual operand or more
- The 8051 Assembly Instruction **(single operand)**

  MOV A, #30

- Moves decimal value 30 to the 8051 Accumulator register. Here MOV A is the Opcode and 30 is the operand (single operand). The same instruction when written in machine language will look like

  01110100 00011110

- The first 8 bit binary value 01110100 represents the opcode MOV A and the second 8 bit binary value 00011110 represents the operand 30.
- LJMP 16bit address (**dual operand** instruction)
- The machine language for the same is

  00000010 addr_bit15 to addr_bit 8 addr_bit7 to addr_bit 0

- The first binary data is the representation of the LJMP machine code.
- The first operand and second operand represents the bits 8 to 15 of the 16bit address and the bits 0 to 7 of the address to which the jump is targeted respectively

- Assembly language instructions are written one per line
- A machine code program consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand)
- **Each line of an assembly language** program is split into four fields as:

  LABEL OPCODE OPERAND COMMENTS
- LABEL is an optional field. A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers for 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 maximum length of a label differs between assemblers.
  - Assemblers insist strict formats for labeling.
  - Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from 0 to 9 and special character _ (underscore).

- Labels are used for representing subroutine names and jump locations in Assembly language programming.
- It is to be noted that ‘LABEL’ is not a mandatory field; it is optional only.
- The sample code given below using 8051 Assembly language illustrates the structured assembly language programming.

```
;#########################################################################
;      SUBROUTINE FOR GENERATING DELAY
;      DELAY PARAMETR PASSED THROUGH REGISTER R1
;      RETURN VALUE NONE
;      REGISTERS USED: R0, R1
;#########################################################################
      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 symbol ; represents the start of a comment. Assembler ignores the text in a line after the ; symbol while assembling the program
- DELAY is a label for representing the start address of the memory location where the piece of code is located in code memory
- The above piece of code can be executed by giving the label DELAY as part of the instruction. E.g. LCALL DELAY; LMP DELAY
- In the above example the LABEL DELAY represents the reference to the start of the subroutine DELAY.
- You can directly replace this LABEL by putting the desired address first and then writing the Assembly code for the routine as given below

```
ORG 0100H
        MOV R0, #255      ; Load Register R0 with 50H
        DJNZ R1, 0100H    ; Decrement R1 and loop till R1= 0
        RET               ; Return to calling program
```

- The advantage of using a label is that the required address is calculated by the assembler at the time of assembling the program and it replaces the Label
- The statement ORG 0100H in the above example is not an assembly language instruction; it is an assembler directive instruction.
- It tells the assembler that the Instructions from here onward should be placed at location starting from 0100H.
- The Assembler directive instructions are known as ‘pseudo-ops’.
- They are used for
  1. Determining the start address of the program (e.g. ORG 0000H)
  2. Determining the entry address of the program (e.g. ORG 0100H)
  3. Reserving memory for data variables, arrays and structures (e.g. var EQU 70H
  4. Initialising variable values (e.g. val DATA 12H)
- The EQU directive is used for allocating memory to a variable and DATA directive is used for initialising a variable with data.
- No machine codes are generated for the ‘pseudo-ops’.

Assembly Language – Source File to Hex File Translation:

- The Assembly language program written in assembly code is saved as .asm (Assembly file) file or a .src (source) file or a format supported by the assembler
- Each module is represented by a '.asm' or '.src' file
- The software utility called 'Assembler' performs the translation of assembly code to machine code
- The assemblers for different family of target machines are different.
  - A51 Macro Assembler from Keil software is a popular assembler for the 8051 family micro controller

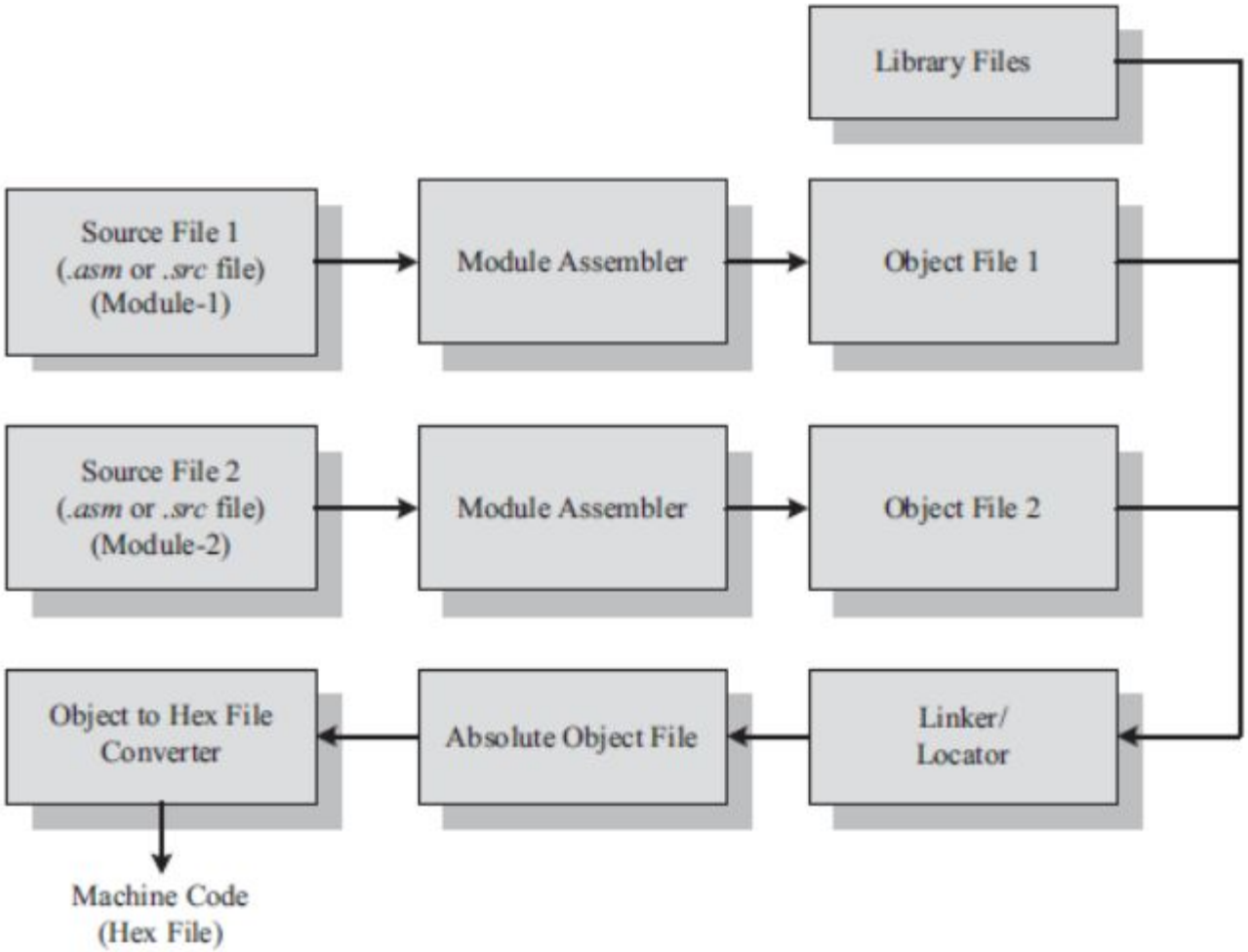

Fig. 9.1 Assembly language to machine language conversion process

- Each source file can be assembled separately to examine the syntax errors and incorrect assembly instructions
- On successful 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 needs to be placed on the program memory and hence it is called a re-locatable segment.
- It can be placed at any code memory location
- The software program called linker/locater is responsible for assigning absolute address to object files during the linking process
- Absolute address allocation is done at the absolute object file creation stage.

- Each module can share variables and subroutines among them.
- Exporting a variable/function from a module is done by declaring that variable/function as PUBLIC in the source module.
- Importing a variable or function from a module is done by declaring that variable or function as EXTRN (EXTERN) in the module where it is going to be accessed.
- The ‘PUBLIC’ Keyword informs the assembler that the variables or functions declared as ‘PUBLIC’ needs to be exported.
- ‘EXTRN’ Keyword tells the assembler that the variables or functions declared as ‘EXTRN’ needs to be imported from some other modules.

- While assembling a module, on seeing variables/functions with keyword ‘EXTRN’, the assembler understands that these variables or functions come from an external module and it proceeds assembling the entire module without throwing any errors, though the assembler cannot find the definition of the variables and implementation of the functions.
- For all those modules using variables or functions with ‘EXTRN’ keyword, there should be one and only one module which exports those variables or functions with ‘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.

- Illustrative example for A51 Assembler–Usage of ‘PUBLIC’ for importing variables with same name on different modules.
- The target application (Simulator) contains three modules namely ASAMPLE1.
- A51, ASAMPLE2.A51 and ASAMPLE3.A51 (The file extension .A51 is the .asm extension specific to A51 assembler).
- The modules ASAMPLE2.A51 and ASAMPLE3.A51 contain a function named PUTCHAR.
- Both of these modules try to export this function by declaring the function as ‘PUBLIC’ in the respective modules.
- While linking the modules, the linker identifies that two modules are exporting the function with name PUTCHAR.
- This confuses the linker and it throws the error ‘MULTIPLE PUBLIC DEFINITIONS’.

```
Build target 'Simulator'
assembling ASAMPLE1.A51...
assembling ASAMPLE2.A51...
assembling ASAMPLE3.A51...
linking...
*** ERROR L104: MULTIPLE PUBLIC DEFINITIONS
 SYMBOL:  PUTCHAR
 MODULE:  ASAMPLE3.obj (CHAR_IO)
```

- If a variable or function declared as ‘EXTRN’ in one or two modules, there should be one module defining these variables or functions and exporting them using ‘PUBLIC’ keyword.
- If no modules in a project export the variables or functions which are declared as ‘EXTRN’ in other modules, it will generate ‘linker’ warnings or errors depending on the error level/warning level settings of the linker.

- Illustrative example for A51 Assembler–Usage of EXTRN without variables exported.
- The modules ASAMPLE1.A51 imports a function named PUT_CRLF which is declared as ‘EXTRN’ in the current module
- But none of the other modules export this function by declaring the function as ‘PUBLIC’ in the respective modules.
- While linking the modules, the linker identifies that there is no function exporting for this function.
- The linker generates a warning or error message ‘UNRESOLVED EXTERNAL SYMBOL’ depending on the linker ‘level’ settings.

```
*** WARNING L1: UNRESOLVED EXTERNAL SYMBOL
 SYMBOL:   PUT_CRLF
 MODULE:   ASAMPLE1.obj (SAMPLE)
```

Library File Creation and Usage

- Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time.
- When the linker processes a library, only those object modules in the library that are necessary to create the program are used.
- Library files are generated with extension '.lib'.
- For using a library file in a project, add the library to the project.
- If you are using a commercial version of the assembler/compiler suite for your development, the vendor of the utility may provide you pre-written library files for performing multiplication, floating point arithmetic, etc. as an add-on utility or as a bonus.
  - 'LIB51' from Keil Software is an example for a library creator and it is used for creating library files for A51 Assembler/C51 Compiler for 8051 specifi c controller

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 generates an absolute object module by extracting the object modules 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 module 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.
  - ‘BL51’ from Keil Software is an example for a Linker & Locator for A51 Assembler/C51 Compiler for 8051 specific controller.

Object to Hex File Converter

- This is the final stage in the conversion of Assembly language (mnemonics) to machine understandable language (machine code).
- Hex File is the representation of the machine code and the hex file is dumped into the code memory of the processor/controller.
- The hex file representation varies depending on the target processor/controller make.
  - For Intel processors/controllers the target hex file format will be ‘Intel HEX’ and for Motorola, the hex file should be in ‘Motorola HEX’ format.

- HEX files are ASCII files that contain a hexadecimal representation of target application.
- ‘OH51’ from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/C51 Compiler for 8051 specific controller.

Advantages:

1. Efficient Code Memory & Data Memory Usage (Memory Optimization):
   - The developer is well aware of the target processor architecture and memory organization, so optimized code can be written for performing operations.
   - This leads to less utilization of code memory and efficient utilization of data memory.
   - memory is a primary concern in any embedded product
2. High Performance:
   - Optimized code not only improves the code memory usage but also improves the total system performance.
   - Through effective assembly coding, optimum performance can be achieved for target processor.
3. Low level Hardware Access:
   - Most of the code for low level programming like accessing external device specific registers from OS kernel ,device drivers, and low level interrupt routines, etc are making use of direct assembly coding

4.Code Reverse Engineering:

- It is the process of understanding the technology behind a product by extracting the information from the finished product.
- Reverse engineering is performed by ‘hackers’ to reveal the technology behind ‘Proprietary Products’.
- Though most of the products 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 a dis-assembler program for the target machine

Drawbacks:

1. High Development time:
   - Assembly language is much harder to program than high level languages.
   - The developer takes lot of time to study about architecture ,memory organization, addressing modes and instruction set of target processor/controller.
   - More lines of assembly code is required for performing a simple action.
2. Developer dependency:
   - There is no common written rule for developing assembly language based applications.
   - In assembly language programming, the developers will have the freedom to choose the different memory location and registers.
   - Also the programming approach varies from developer to developer depending on his/ her taste

- For example moving data from a memory location to accumulator can be achieved through different approaches.
- If the approach done by a developer is not documented properly at the development stage, he/ she may not be able to recollect why this approach is followed at a later stage or when a new developer is instructed to analyse this code, he/she also may not be able to understand what is done and why it is done.
- Hence upgrading an assembly program or modifying it on a later stage is very difficult.
- Well documenting the assembly code is a solution for reducing the developer dependency in assembly language programming.
- If the code is too large and complex, documenting all lines of code may not be productive.

3. Non portable:

- Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/controllers.
- If the target processor/controller changes, a complete re-writing of the application using assembly language for new target processor/controller is required.