# MODULE 3

# MODULE 3

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

## Outcome:

✓ Model the operation of a given embedded system

## Reference text:


1. J Staunstrup and Wayne Wolf, Hardware / Software Co-Design: Principles and Practice, Prentice Hall.
2. Jean J Labrose Micro C/OS II: The Real Time Kernel
3. Raj Kamal, Embedded Systems: Architecture, Programming and Design, Third Edition, McGraw Hill Education (India), 2014.
4. **Shibu K.V., Introduction to Embedded Systems, McGraw Hill Education (India),2009. Chapter 9.1,9.2**
5. Steave Heath, Embedded System Design, Second Edition, Elsevier.
6. Wayne Wolf , Computers as Components-Principles of Embedded Computer System Design, Morgan Kaufmann publishers, Third edition, 2012.

# 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 develoment

# 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

# 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. ….
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 1to n are performed one after another and when the last task is executed, the firmware execution is again redirected to task 1and 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
- 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 , it won’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 <u>any failure in any part of a single task will affect the total system</u>
  - 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 connectedI/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 topress the key for a sufficientlylong time tillthe keypad status monitoringtask is executed internally.
  - Lead to lack of real timeliness

# 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
  - 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 different hardware present on the board to communicate with them

- 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 devolopment

- 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
- Even in 1990s majority of console video games were written in assembly languages
- Even today almost all low level, system related, programming is carried out using assembly language
- 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 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
  - 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
  - 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 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
  - Label is only an optional field

```
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 involked 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 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  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
```

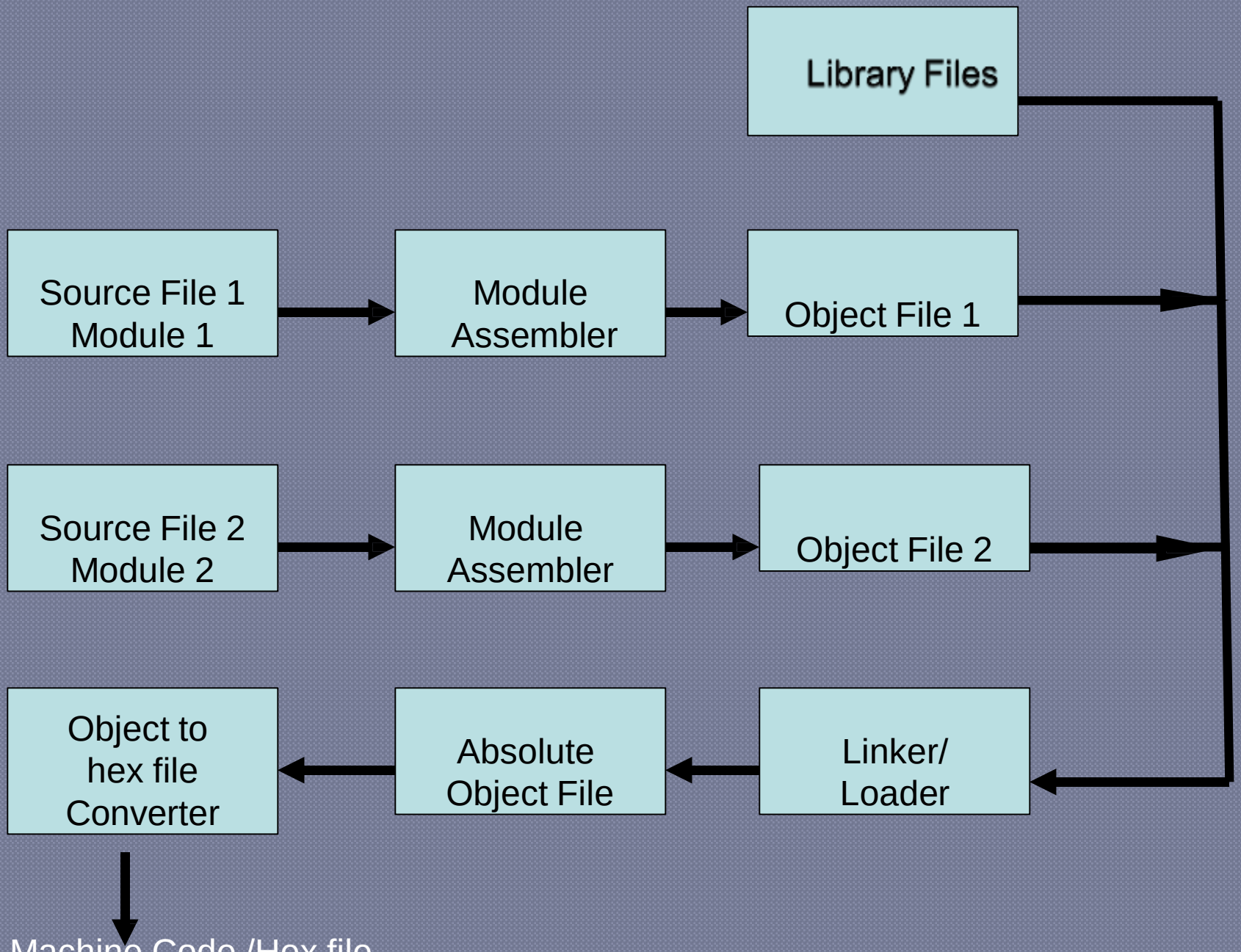
- ORG 0100H is not an assembly language instruction; it is an assembler directive instruction. It tells the assembler that the instruction from here onwards should be placed at location starting from 0100H
- Assembler directive instructions are known as ‘pseudo ops’
- They are used for
  - Determining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

# SOURCE FILE TO OBJECT FILE TRANSLATION

- Translation of assembly code to machine code is performed by assembler
- The assemblers for different target machines are differentand it iscommon that assemblers from multiple vendorsare available in themarket forthesame 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 8051family microcontroller


Library Files
Source File 1
Module 1
Module
Assembler
Object File 1
Source File 2
Module 2
Module
Assembler
Object File 2
Object to
hex file
Converter
Absolute
Object File
Linker/
Loader
Machine Code /Hex file

- 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
- While assembling a module , on seeing variable /function with keyword EXTRN , assembler understand that these variables or function come from an external module and it proceeds assembling the entire module without throwing any errors, though the assembler cannot find the definition of variables and implementation of that function

- 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

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)
```

# 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

# 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

# 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 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
- Hex file is created from the final ‘Absolute Object File’ using the Object to Hex file Converter utility
- Example ‘OH51’ from keil software

# 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

- 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

- Code Reverse Engineering
  - Reverse Engineering is the process nof 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

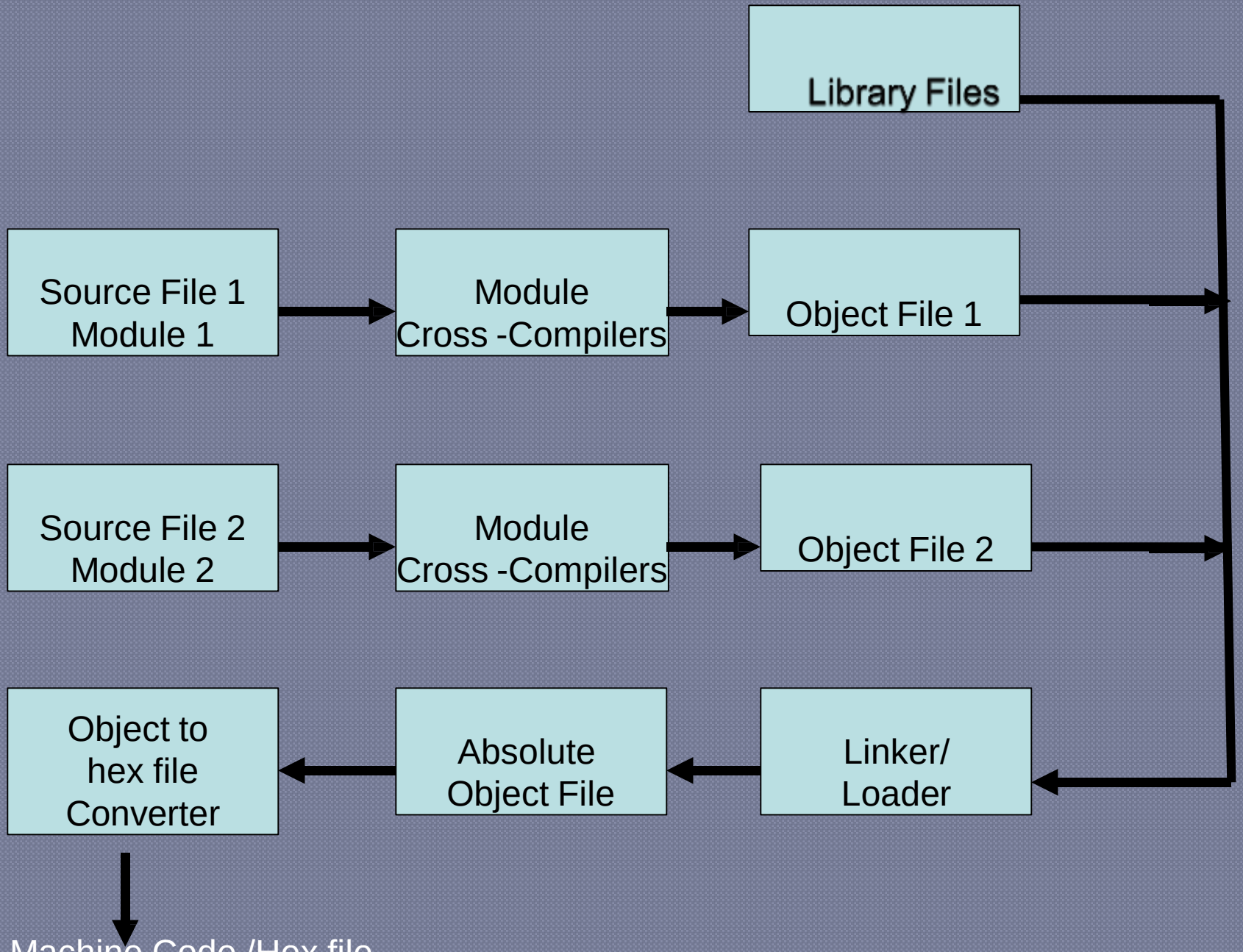
- **High Development time**
  - **Assembly language programs are much harder to program than high level languages**
  - **Developer must have thorough knowledge of architecture, memory organization and register details of target processor in use**
  - **Learning the inner details of the processor and 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
  - 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
  - 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

# 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


Library Files
Source File 1
Module 1
Module
Cross -Compilers
Object File 1
Source File 2
Module 2
Module
Cross -Compilers
Object File 2
Object to
hex file
Converter
Absolute
Object File
Linker/
Loader
Machine Code /Hex file

- 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 develoment
  - Example C51 Compiler fro Keil

# 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

# 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

# Mixing Assembly Language with High level Language (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

- Example
  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

```
#pragma src
Unsigned char my_assembly_func(unsigned int
    argument)
{
Return (argument+1);
}
```

**This C function on cross compilation generate 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);
;SOURCE LINE #5
                                MOV A,R7
                                INC   A
                                MOV R7,A
;}
;source line #6
;?C0001;
                RET
;END OF _my_assembly_func
                END
```

- The special compiler directive SRC generates the Assembly code corresponding to the ‘C’ function and each line of the source code is converted to the corresponding Assembly instruction.
- You can easily identify the Assembly code generated for each line of the source code since it is implicitly mentioned in the generated .SRC file
- By inspecting this code segments you can find out which register are used for holding the variables of the ‘C’ function and you can modify the source code by adding the assembly routine you want

# Mixing High level Language with Assembly Language ('C' with Assembly Language)

- Mixing the code written in high level language like C and assembly language is useful in the following scenarios;
  - The source code is already available in Assembly language and a routine written in a high level language like C need to be included to the existing code
  - The entire code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the Assembly, etc. But some portions of the code may be very difficult and tedious to code in Assembly
  - To include built in library functions written in C language provided by the cross compiler

- Most often the functions written in C use parameter passing to the function and returns values to the calling function
- By mixing C with Assembly
  - How parameters are passed to the function
  - How values are returned from the function
  - 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 dependant and it varies across cross compilers
- Example Keil C51 cross compiler

- C51 allows passing of maximum of three arguments through general purpose registers R2 to R7
- If three arguments are char variables, they are passed to the functions using registers R7,R6, and R5
- If the parameters are int variables they are passing using register pairs(R7,R6),(R5,R4) and (R4,R3)
- If the number of arguments are greater than three, the first three arguments are passed through registers and the rest is passed through fixed memory locations
- Return values are usually passed through fixed memory locations
- R7 is used for returning char value and register pair (R7,R6) used for returning int values

- The C subroutine can be involked from the assembly program using the subroutine call Assembly instruction

  LCALL _Cfunction

- Where Cfunction is a function written in C
- The  prefix inform 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

- This is another technique for inserting target processor/controller specific assembly instructions at any location of source code written in high level language C
- This avoid the delay in calling an assembly routine from a C code
- Special keywords are used to indicate that the start and end of the Assembly instructions
- The keywords are cross compiler specific
- C51 uses #pragma asm and #pragma endasm to indicate a block of code written in assembly

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

# Previous Year Questions

1) Explain the firmware execution flow of super loop based approach
2) Describe mixing high level language with Assembly code with an example
3) Differentiate General purpose Operating System (GPOS) with Real time Operating system(RTOS)
4) Describe the firmware design approaches used in an embedded product
5) What is 'Inline Assembly' ? Explain with an example

6) Explain the library file in assembly language context. What is the benefit of ‘library file’.
7) With a neat diagram explain the steps in converting assembly language to machine language
8) Explain the merits and demerits of assembly language based embedded firmware development .