Module4 : Embedded Firmware Design and Development

Introduction to Embedded Firmware Design

- 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.
- Imparting intelligence to an Embedded system is a one time process and it can happen at any stage.
    - It can be immediately after the fabrication of the embedded hardware or at a later stage.
- For most of the embedded products, the embedded firmware is stored at a permanent memory (ROM) and they are non-alterable by end users.
    - Some of the embedded products used in the Control and Instrumentation domain are adaptive.
- 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.
- Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modelling tools.
- 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.

Embedded Firmware Design Approaches

- 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.
- Two basic approaches are used for embedded firmware design:
    - Super Loop Based Approach (Conventional Procedural Based Design)
    - Embedded Operating System (OS) Based Approach

Super Loop Based Approach

- The Super Loop based firmware development approach is adopted for applications that are not time critical and where the response time is not so important.
- It is very similar to a conventional procedural programming where the code is executed task by task.
- The task listed at the top of the program code is executed first and the tasks just below the top are executed after completing the first task.
- In a multiple task based system, each task is executed in serial in this approach.
- The firmware execution flow for this will be

- Configure the common parameters and perform initialization for various hardware components memory, registers, etc.
- Start the first task and execute it
- Execute the second task
- Execute the next task
  :
  :
- Execute the last defined task
- Jump back to the first task and follow the same flow

➢ The order in which the tasks to be executed are fixed and they are hard coded in the code itself.
- Also the operation is an infinite loop based approach.

➢ We can visualise the operational sequence listed above in terms of a 'C' program code as

```
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.
- This repetition is achieved by using an infinite loop.
- Hence the name 'Super loop based approach'.

➢ The only way to come out of the loop is either a hardware reset or an interrupt assertion.

➢ Advantage of Super Loop Based Approach:
- It doesn't require an operating system
- There is no need for scheduling which task is to be executed and assigning priority to each task.
- 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.

➢ Applications of Super Loop Based Approach:
- 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.

➢ A typical example of a 'Super loop based' product is an electronic video game toy containing keypad and display unit.
- The program running inside the product may be designed in such a way that it 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.

➢ Drawbacks of Super Loop Based Approach:
- Any failure in any part of a single task will affect the total system.
  - If the program hangs up at some point while executing a task, it will remain there forever and ultimately the product stops functioning.
  - Watch Dog Timers (WDTs) can be used to overcome this, but this, in turn, may cause additional hardware cost and firmware overheads.
- 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.

Embedded Operating System (OS) Based Approach

➢ The Embedded 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.

➢ 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.
- Example of a GPOS used in embedded product development is Microsoft Windows XP Embedded.
- Examples of Embedded products using Microsoft Windows XP OS are Personal Digital Assistants (PDAs), Hand held devices/Portable devices and Point of Sale (POS) terminals.

➢ Use of GPOS in embedded products merges the demarcation of Embedded Systems and general computing systems in terms of OS.

➢ 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.

- Real Time Operating System (RTOS) based design approach is employed in embedded products demanding Real-time response.
  - RTOS responds 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. 'Windows CE', 'pSOS', 'VxWorks', 'ThreadX', 'MicroC/OS-II', 'Embedded Linux', 'Symbian', etc. are examples of RTOS employed in embedded product development.
- Mobile phones, PDAs (Based on Windows CE/Windows Mobile Platforms), handheld devices, etc. are examples of 'Embedded Products' based on RTOS.

Embedded Firmware Development Languages

- For embedded firmware development, we can use either
  - a target processor/controller specific language (Generally known as Assembly language or low level language) or
  - a target processor/controller independent language (Like C, C++, JAVA, etc. commonly known as High Level Language) or
  - a combination of Assembly and High level Language.

Assembly Language Based Development

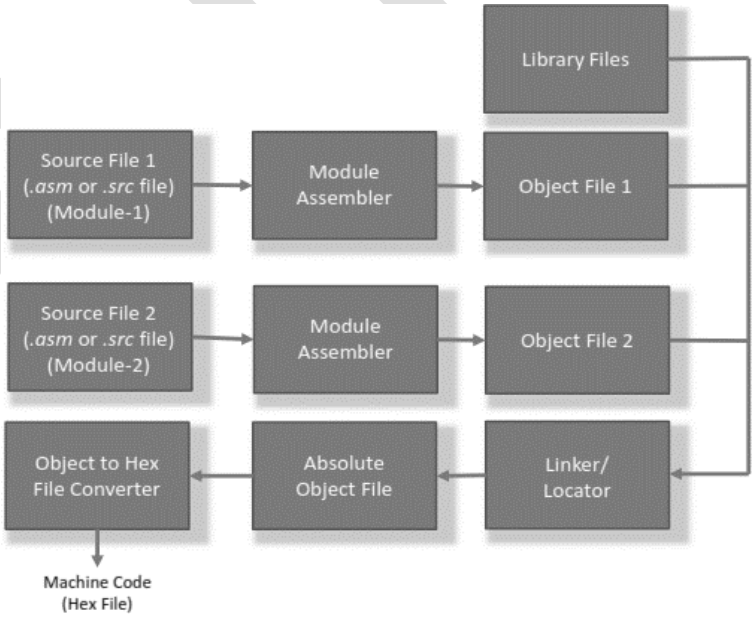
- Assembly language is the human readable notation of 'machine language'
  - 'Machine Ianguage' is a processor understandable language.
- Machine language is a binary representation and it consists of 1s and 0s.
- Machine language is made readable by using specific symbols called 'mnemonics'.
- Hence machine language can be considered as an interface between processor and programmer.
- Assembly language and machine languages are processor/controller dependent and an assembly program written for one processor/controller family will not work with others.
- Assembly language programming is the task of writing processor specific machine code in mnemonic 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 today also almost all low level, system related, programming is carried out using assembly language.
- In particular, assembly language is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers.
- 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.

- For example: MOV A, #30
- Here MOV is the Opcode and A, #30 is the operands

- The Assembly language program written in assembly code is saved as .asm (Assembly file) file or an .src (source) file (also. s file).
- Any text editor like 'Notepad' or 'WordPad' from Microsoft or the text editor provided by an Integrated Development (IDE) tool can be used for writing the assembly instructions.
- Similar to 'C' and other high level language programming, we can have multiple source files called modules in assembly language programming.
  - Each module is represented by an '.asm' or '.src' file.
  - This approach is known as 'Modular Programming'.
- Modular programming is employed when the program is too complex or too big.
  - In 'Modular Programming', the entire code is divided into submodules and each module is made re-usable.
  - Modular Programs are usually easy to code, debug and alter.

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.
  - A51 Macro Assembler from Keil software is a popular assembler for the 8051 family microcontroller.
- The various steps involved in the conversion of a program written in assembly language to corresponding binary file/machine language are illustrated in the figure.



- Each source module is written in Assembly and is stored as .src file or .asm file.
- Each 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 and it is the responsibility. of the linker/locater to assign absolute address for this module.

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.
  - Library files are generated with extension '. lib'.
- When the linker processes a library, only those object modules in the library that are necessary to create the program are used.
- Library file is some kind of source code hiding technique.
  - For example, '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 specific controllers.

Linker and Locator

- Linker and Locater 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 & Locater 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.
- 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.
    - 'OH51' from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/C51 Compiler for 8051 specific controller.

Advantages of Assembly Language Base Development

- Efficient Code Memory and Data Memory Usage (Memory Optimisation)
    - Since the developer is well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations.
    - This leads to less utilisation of code memory and efficient utilisation of data memory.
- High Performance
    - Optimised 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 a target application.
- 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 routines, etc. are making use of direct assembly coding since low level device specific operation support is not commonly available with most of the high-level language cross compilers.
- Code Reverse Engineering
    - Reverse engineering is the process of understanding the technology behind a product by extracting the information from a finished product.
    - Reverse engineering is performed by 'hawkers' 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 of Assembly Language Based Development

- High Development Time
    - Assembly language is much harder to program than high level languages.
    - The developer must pay attention to more details and must have thorough knowledge of the architecture, memory organisation and register details of the target processor in use.
    - Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.

- 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'.

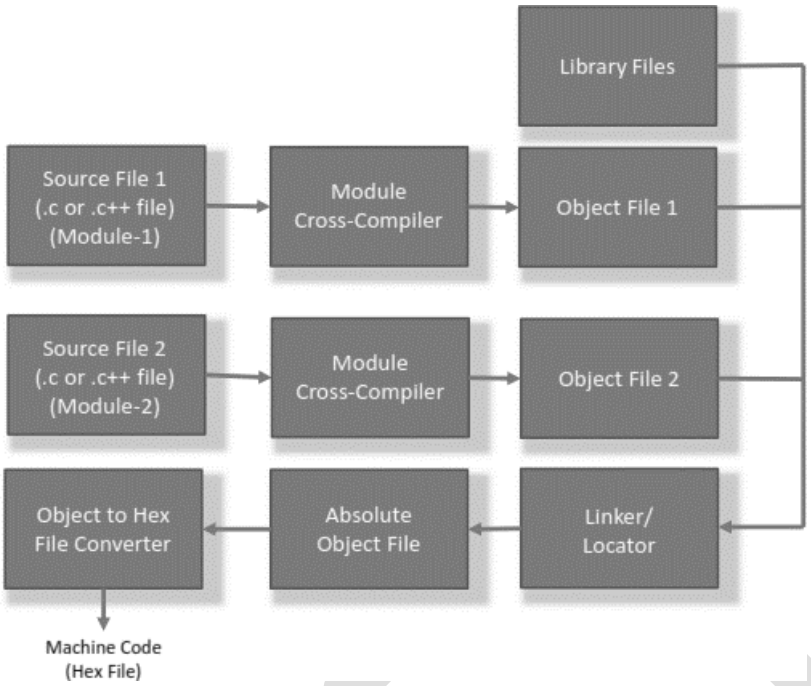
➢ Developer Dependency
- Unlike high level languages, there is no common written rule for developing assembly language based applications.
- In assembly language programming, the developers will have the freedom to choose the different memory location and registers.
- Also the programming approach varies 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.

➢ Non-Portable
- Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Intel x86 family of processors) and cannot be re-used for another target processors/controllers (Say ARM11 family of processors).
- If the target processor/controller changes, a complete re-writing of the application using the assembly instructions for the new target processor/controller is required.

High Level Language Based Development

➢ Any high level language (like C, C++ or Java) with a supported cross compiler for the target processor can be used for embedded firmware development.
➢ The most commonly used high level language for embedded firmware application development is 'C'.
- 'C' is well defined, easy to use high level language with extensive cross platform development tool support.

➢ Nowadays cross-compilers for C++ is also emerging out and embedded developers are making use of C++ for embedded application development.
➢ The various steps involved in high level language based embedded firmware development is same as that of assembly language based development except that the conversion of source file written in high level language to object file is done by a cross-compiler.
➢ In Assembly language based development it is carried out by an assembler.
➢ The various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language is illustrated in the figure.



- The program written in any of the high level languages is saved with the corresponding language extension (.c for C, .cpp for C++ etc).
- Any text editor like 'Notepad' or 'WordPad' from Microsoft or the text editor provided by an Integrated Development (IDE) tool can be used for writing the program.
- Most of the high level languages support modular programming approach and hence we can have multiple source files called modules written in corresponding high level language.
- The source files 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 across-compiler. Each high level language should have a cross-compiler for converting the high level source code into the target processor machine code.
    - C51 Cross-compiler from Keil software is an example for Cross-compiler used for 'C' language for the 8051 family of microcontroller.
- Conversion of each module's source code to corresponding object file is performed by the cross-compiler.
- Rest of the steps involved in the conversion of high level language to target processor's machine code are same as that of the steps involved in assembly language based development.

Advantages of High Level Language Based Development

- Reduced Development Time
    - Developer requires less or little knowledge on the internal hardware details and architecture of the target processor/controller.

- Bare minimal knowledge of the memory organisation and register details of the target processor in use and syntax of the high level language 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/controller specific assembly language based development.

➢ Developer Independency

- The syntax used by most of the high level languages are universal and a program written in the high level language can easily be understood by a second person knowing the syntax of the language.
- High level languages always instruct certain set of rules for writing the code and commenting the piece of code.
- If the developer strictly adheres to the rules, the firmware will be 100% developer independent.

➢ Portability

- Target applications written in high level languages are converted to target processor/controller understandable format (machine codes) by a cross-compiler.
- An application written in high level language for a particular target processor can easily be converted to another target processor/controller specific application, with little or less effort by simply re-compiling/little code modification followed by recompiling the application for the required target processor/controller, provided, the cross-compiler has support for the processor/controller selected.
- This makes applications written in high level language highly portable.
- Little effort may be required in the existing code to replace the target processor specific files with new header files, register definitions with new ones, etc.
- This is the major flexibility offered by high level language based design.

Limitations of High Level Language Based Development

➢ Poor Optimization by Cross-Compilers

- Some cross-compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions.
- Target images created by such compilers may be messy and non-optimised in terms of performance as well as code size.
- The time required to execute a task also increases with the number of instructions.

➢ Not Suitable for Low Level Hardware

- High level language based code snippets may not be efficient in accessing low level hardware where hardware access timing is critical (of the order of nano or micro seconds).

➢ High Investment Cost

- The investment required for high level language based development tools (Integrated Development Environment incorporating cross-compiler) is high compared to Assembly Language based firmware development tools.

Mixing Assembly and High Level Language

- Certain embedded firmware development situations may demand the mixing of high level language with Assembly and vice versa.
- 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 Language
  - Inline Assembly programming

Mixing Assembly Language with High Level Language

- Assembly routines are mixed with 'C' in situations where
  - the entire program is written in 'C' and the cross compiler in use do not have a built in support for implementing certain features like Interrupt Service Routine functions (ISR) or
  - if the programmer wants to take advantage of the speed and optimised 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 specifications may be very critical and a cross compiler generated binary may not be able to offer the required time specifications accurately.
  - Writing the hardware/peripheral access routine in processor/controller specific Assembly language and invoking it from 'C' is the most advised method to handle such situations.
- Mixing 'C' and Assembly is little complicated.
  - 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 'Assembly routine' is invoked from the 'C' code.
- Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross-compiler dependent.
- Consider an example Keil C51 cross compiler for 8051 controller.
- The steps for mixing assembly code with ‘C’ are:
  - Write a simple function in C that passes parameters and returns values the way you want your assembly routine to.
  - Use the SRC directive (#PRAGMA SRC at the top of the file) so that the C compiler generates an .SRC file instead of an .OBJ file.
  - Compile the C file. Since the SRC directive is specified, the .SRC file is generated. The .SRC file contains the assembly code generated for the C code you wrote.
  - Rename the .SRC file to .A51 file.
  - Edit the .A51 file and insert the assembly code you want to execute in the body of the assembly function shell included in the . A51 file.

Mixing High Level Language with Assembly Language

- Mixing the code written in a 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' needs to be included to the existing code.
  - 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, 16-bit multiplication and division in 8051 Assembly Language.
  - 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.
- 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.

Inline Assembly Programming

- Inline assembly is a 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.
- 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.

Embedded System Development Environments

This chapter is designed to give you an insight into the embedded-system development environment. The various tools used and the various steps followed in embedded system development are explained here. A typical embedded system development environment is illustrated in Fig.

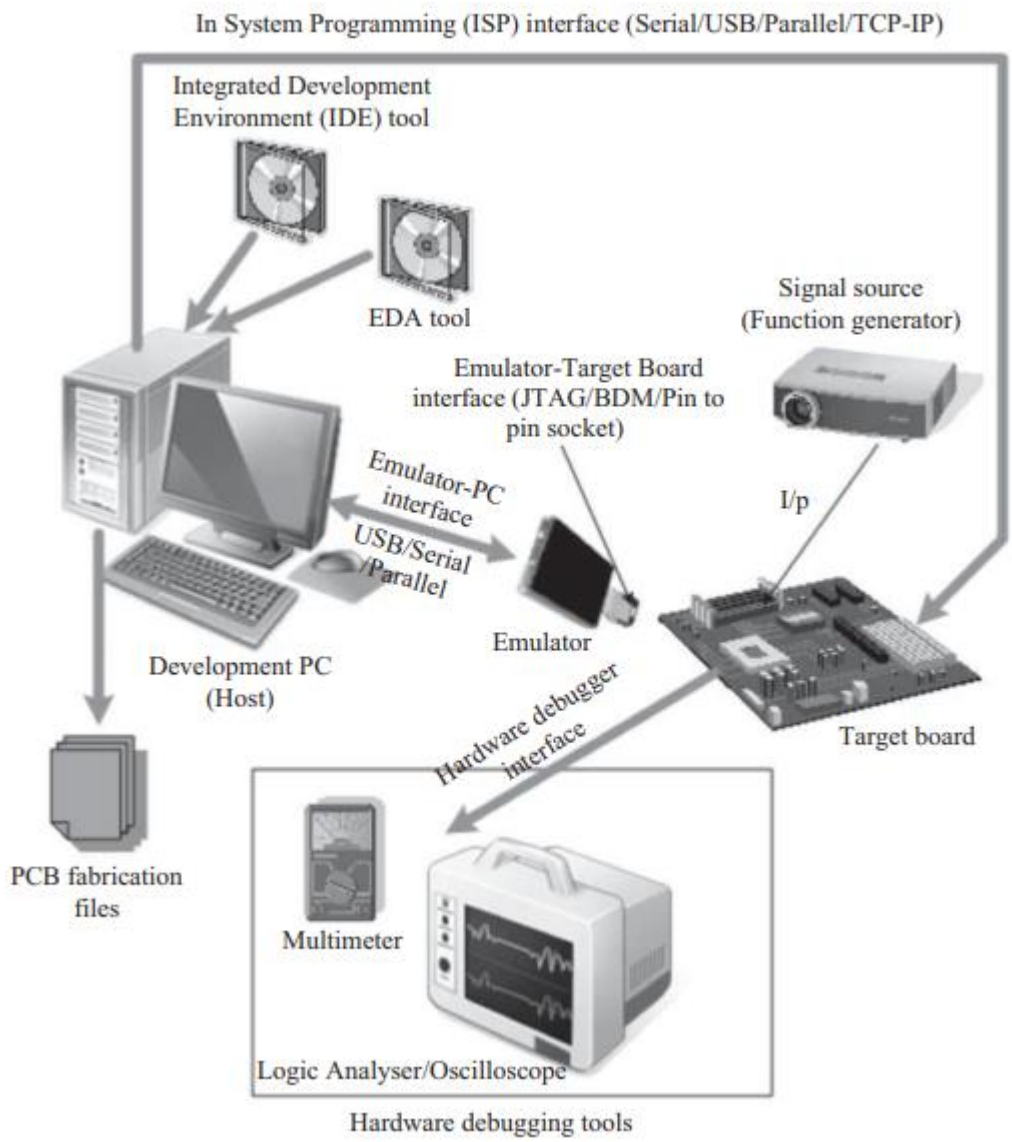


Fig: The Embedded System Development Environment

As illustrated in the fi gure, the development environment consists of a Development Computer (PC) or Host, which acts as the heart of the development environment, Integrated Development Environment (IDE) Tool for embedded fi rmware development and debugging, Electronic Design Automation (EDA) Tool for Embedded Hardware design, An emulator hardware for debugging the target board, Signal sources (like Function generator) for simulating the inputs to the target board, Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.) and the target hardware. The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement and they are supplied as Installable fi les in CDs/Online downloads by vendors. These tools need to be installed on the host PC

used for development activities. These tools can be either freeware or licensed copy or evaluation versions. Licensed versions of the tools are fully featured and fully functional whereas trial versions fall into two categories, tools with limited features, and full featured copies with limited period of usage.

TYPES OF FILES GENERATED ON CROSS-COMPILATION:

- Cross-compilation is the process of converting a source code written in high level language (like 'Embedded C') to a target processor/ controller understandable machine code (e.g. ARM processor or 8051 microcontroller specific machine code).
- The conversion of the code is done by software running on a processor/controller (e.g. x86 processor based PC) which is different from the target processor. The software performing this operation is referred as the 'Cross-compiler'. In a single word cross-compilation is the process of cross platform software/firmware development.
- The application converting Assembly instruction to target processor/controller specific machine code is known as cross-assembler.
- The various fi les generated during the crosscompilation/cross-assembling process are: **List File (.lst), Hex File (.hex), Pre-processor Output file, Map File (File extension linker dependent), Object File (.obj)**

I. List File (.LST File)

Listing file is generated during the cross-compilation process and it contains an abundance of 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' generated contains the following sections.

1. **Page Header**: A header on each page of the listing file which indicates the compiler version number, source file name, date, time, and page number.
2. **Command Line :** Represents the entire command line that was used for invoking the compiler.
3. **Source Code :** The source code listing outputs the line number as well as the source code on that line. Special cross compiler directives can be used to include or exclude the conditional codes (code in #if blocks) in the source code listings.
4. **Assembly Listing:** Assembly listing contains the assembly code generated by the cross compiler for the 'C' source code. Assembly code generated can be excluded from

the list file by using special compiler directives.

5. **Symbol Listing:** The symbol listing contains symbolic information about the various symbols present in the cross compiled source fi le. Symbol listing contains the sections symbol name (NAME) symbol classifi cation (CLASS (Special Function Register (SFR), structure, typedef, static, public, auto, extern, etc.)), memory space (MSPACE (code memory or data memory)), data type (TYPE (int, char, Procedure call, etc.)), offset ((OFFSET from code memory start address)) and size in bytes (SIZE). Symbol listing in list fi le output can be turned on or off by cross-compiler directives.
6. **Module Information:** The module information provides the size of initialized and un-initialized memory areas defined by the source file.
7. **Warnings and Errors:** Warnings and Errors section of list fi le records the errors encountered or any statement that may create issues in application (warnings), during cross compilation. The warning levels can be configured before cross compilation.

II. Preprocessor Output File:

The preprocessor output file generated during cross-compilation contains the preprocessor output for the preprocessor instructions used in the source file. Preprocessor output file is used for verifying the operation of macros and conditional preprocessor directives. The preprocessor output file is a valid C source file. File extension of preprocessor output file is cross compiler dependent.

III. Object File (.OBJ File)

Cross-compiling/assembling each source module (written in C/Assembly) converts the various Embedded C/ Assembly instructions and other directives present in the module to an object (.OBJ) file. The format (internal representation) of the .OBJ file is cross compiler dependent.

The list of some of the details stored in an object file is given below. 1. Reserved memory for global variables. 2. Public symbol (variable and function) names. 3. External symbol (variable and function) references. 4. Library files with which to link. 5. Debugging information to help synchronize source lines with object code.

IV. Map File (.MAP)

As mentioned above, the cross-compiler converts each source code module into a re-locatable object (OBJ) fi le. Cross-compiling each source code module generates its own list file. In a project with multiple source files, the cross-compilation of each module generates a corresponding object file. The object files so created are re-locatable codes, meaning their location in the code memory is not fixed. It is the responsibility of a linker

to link all these object files. The locater is responsible for locating absolute address to each module in the code memory. Linking and locating of re-locatable object files will also generate a list file called 'linker list file' or 'map file'.

V. HEX File (.HEX)

Hex file is the binary executable file created from the source code. The absolute object file created by the linker/locater is converted into processor understandable binary code. The utility used for converting an object file to a hex file is known as Object to Hex file converter. Hex files embed the machine code in a particular format.

DISASSEMBLER/DECOMPILER

- Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/instructions.
- The process of converting machine codes into Assembly code is known as 'Disassembling'.
- In operation, disassembling is complementary to assembling/crossassembling.
- Decompiler is the utility program for translating machine codes into corresponding high level language instructions.
- Decompiler performs the reverse operation of compiler/cross-compiler. The disassemblers/ decompilers for different family of processors/controllers are different.
- Disassemblers/decompilers help the reverse-engineering process by translating the embedded firmware into Assembly/high level language instructions.
- Disassemblers/ Decompilers are powerful tools for analysing the presence of malicious codes (virus information) in an executable image.
- However disassemblers/decompilers generate a source code which is somewhat matching to the original source code from which the binary code is generated.

SIMULATORS, EMULATORS AND DEBUGGING:

- Simulators and emulators are two important tools used in embedded system development. Simulator is a software tool used for simulating the various conditions for checking the functionality of the application firmware.
- The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality.
- Emulator is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.

i. Simulators

- Simulators simulate the target hardware and the firmware execution can be inspected using simulators. The features of simulator based debugging are listed below. Purely software based 2. Doesn't require a real target system 3. Very primitive (Lack of featured I/O support. Everything is a simulated one) 4. Lack of Real-time behavior.
- **Advantages of Simulator Based Debugging**
- Simulator based debugging techniques are simple and straightforward.
- No Need for Original Target Board Simulator based debugging technique is purely software oriented.
- Simulate I/O Peripherals, s. Using simulator's I/O support you can edit the values for I/O registers and can be used as the input/output value in the firmware execution.
- Simulates Abnormal Conditions: With simulator's simulation support you can input any desired value for any parameter during debugging the firmware and can observe the control flow of firmware.
- **Limitations of Simulator based Debugging**
- **Deviation from Real Behaviour**: Simulation-based firmware debugging is always carried out in a development environment where the developer may not be able to debug the firmware under all possible combinations of input.
- Under certain operating conditions we may get some particular result and it need not be the same when the firmware runs in a production environment.
- **Lack of Real Timeliness :** The major limitation of simulator based debugging is that it is not real-time in behaviour.
- The debugging is developer driven and it is no way capable of creating a real time behaviour.

ii. Emulators and Debuggers

What is debugging and why debugging is required?

- 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 and checking the signals from various buses of the embedded hardware.
- Debugging process in embedded application is broadly 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.

As technology has achieved a new dimension from the early days of embedded system development, various types of debugging techniques are available today. The following section describes the improvements over firmware debugging starting from the most primitive type of debugging to the most sophisticated On Chip Debugging (OCD).

Incremental EEPROM Burning Technique

- This is the most primitive type of firmware debugging technique where the code is separated into different functional code units. Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremental order, where the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.
- If the first functionality is found working perfectly on the target board with the corresponding code burned into the EEPROM, go for burning the code corresponding to the next functionality and check whether it is working. Repeat this process till all functionalities are covered.
- After you found all functionalities working properly, combine the entire source for all functionalities together, re-compile and burn the code for the total system functioning.
- Obviously it is a time-consuming process. But remember it is a onetime process and once you test the firmware in an incremental model you can go for mass production.

Inline Breakpoint Based Firmware Debugging

- Inline breakpoint based debugging is another primitive method of firmware debugging. Within the firmware where you want to ensure that firmware execution is reaching up to a specified point, insert an inline debug code immediately after the point.
- The debug code is a printf() function which prints a string given as per the firmware. You can insert debug codes (printf()) commands at each point where you want to ensure the firmware execution is covering that point.

Monitor Program Based Firmware Debugging

- Monitor program based firmware debugging is the first adopted invasive method for

firmware debugging (Fig.).

- In this approach a monitor program which acts as a supervisor is developed. The monitor program controls the downloading of user code into the code memory, inspects and modifies register/memory locations; allows single stepping of source code, etc.
- The monitor program implements the debug functions as per a pre-defined command set from the debug application interface.
-

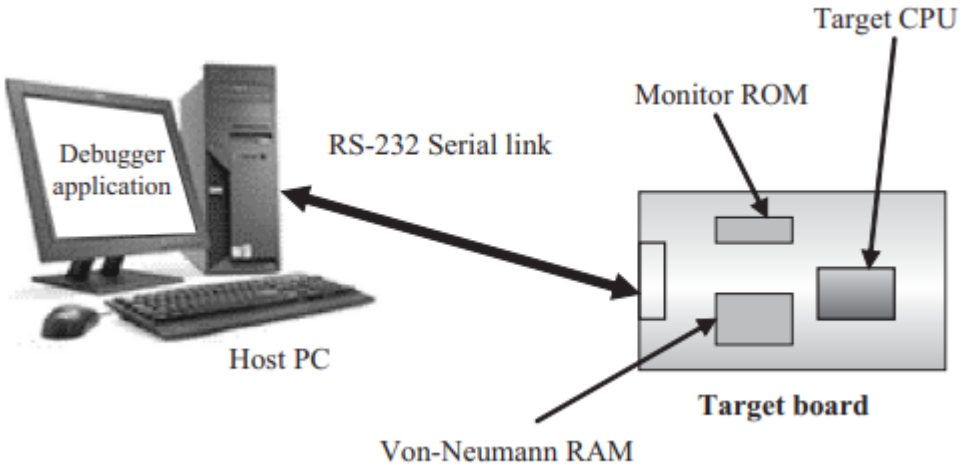


Fig: Monitor Program Based Target Firmware Debug Setup

The monitor program contains the following set of minimal features.

1. Command set interface to establish communication with the debugging application
2. Firmware downloads option to code memory
3. Examine and modify processor registers and working memory (RAM)
4. Single step program execution
5. Set breakpoints in firmware execution 6. Send debug information to debug application running on host machine

In Circuit Emulator (ICE) Based Firmware Debugging

- Emulator' is a self-contained hardware device which emulates the target CPU. The emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end.
- The emulator application for emulating the operation of a PDA phone for application development is an example of a 'Software Emulator'. A hardware emulator is controlled by a debugger application running on the development PC. The debugger application may be part of the Integrated Development Environment (IDE) or a third party supplied tool.

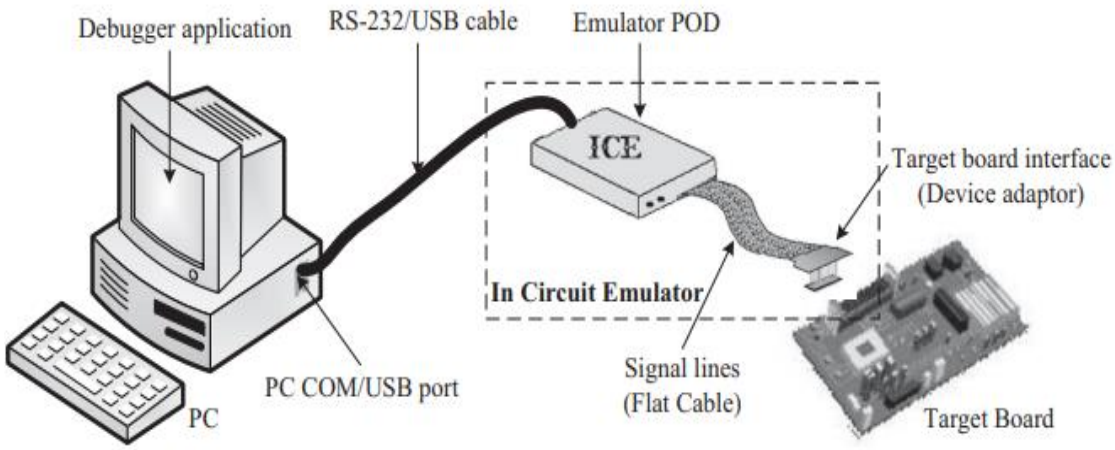


Fig: In Circuit Emulator (ICE) Based Target Debugging

- Emulation device is a replica of the target CPU which receives various signals from the Emulator on Device target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug commands from the debug application. The emulation device can be either a standard chip same as the target processor (e.g. AT89C51) or a Programmable Logic Device (PLD) configured to function as the target CPU.
- It is the Random Access Memory (RAM) incorporated in the Emulator device. It acts Emulator on Memory as a replacement to the target board's EEPROM where the code is supposed to be downloaded after each firmware modification. Hence the original EEPROM memory is emulated by the RAM of emulator. This is known as 'ROM Emulation'.
- Emulator control logic is the logic circuits used for implementing complex Emulator Control Logic hardware breakpoints, trace buffer trigger detection, trace buffer control, etc. Emulator control logic circuits are also used for implementing logic analyser functions in advanced emulator devices.
- Device adaptors act as an interface between the target board and emulator POD. Device Device Adaptors adaptors are normally pin-to-pin compatible sockets which can be inserted/plugged into the target board for routing the various signals from the pins assigned for the target processor. The device adaptor is usually connected to the emulator POD using ribbon cables. The adaptor type varies depending on the target processor's chip package. DIP, PLCC, etc. are some commonly used adaptors.

On Chip Firmware Debugging (OCD)

- Today almost all processors/controllers incorporate built in debug modules called On

Chip Debug (OCD) support. Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging.

- Processors/controllers with OCD support incorporate a dedicated debug module to the existing architecture. Usually the on-chip debugger provides the means to set simple breakpoints, query the internal state of the chip and single step through code.
- BDM and JTAG are the two commonly used interfaces to communicate between the Debug application running on Development PC and OCD module of target CPU.
- Background Debug Mode (BDM) interface is a proprietary On Chip Debug solution from Motorola. BDM defines the communication interface between the chip resident debug core and host PC where the BDM compatible remote debugger is running.
- Chips with JTAG debug interface contain a built-in JTAG port for communicating with the remote debugger application. JTAG is the acronym for Joint Test Action Group.

Model Questions

1. Explain Super loop-based Approach?
2. Advantages and disadvantages of Assembly Language Base Development.
3. What are the Advantages and limitations of High Level Language Based Development.
4. Explain types of files generated on cross compilation.
5. Explain simulators?
6. Explain Incremental EEPROM Burning Technique and Inline Breakpoint Based Firmware Debugging.
7. Explain Monitor based firmware debugging.
8. Explain in circuit emulator-based firmware debugging.