EMBEDDED FIRMWARE DESIGN & DEVELOPMENT

What is Firmware?

- Firmware refers to a small piece of code that resides in non-volatile memory.
- In hardware peripherals that are commonly found in offices these days (e.g. printers, VoIP(voice over Internet protocol) phones, etc.)
- Firmware's are usually responsible for loading (e.g. OS code signature verification) and managing (e.g. flashing OS in recovery mode) the operating systems that is installed on the machine.
- It's the operating system's job to carry out the actual task (e.g. printing)

Firmware design approaches:

- Two basic approaches are used in embedded systems.

1. Conventional procedural based design approach or super loop model.

2. Embeded operating system based design

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
- i.e..task by task approach.

➢ Task executed in serial in this approach.

Firmware execution flow:

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

Example: c program code

```
Void main ()
{
Configurations();
Initializations();
While(1)
{
Task 1();
Task 2();
…..
Task n();
}
}
```

Analysis:

- The tasks are running inside an infinite loop the only way to come out of the loop is hardware interrupt or an interrupt assertion.
- A hardware reset brings back the program execution back to the main loop.
- Interrupt routine 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.

Merits and demerits:

Merit:

- Low cost

Demerit:

- Failure in one part affect whole system
- Lack of real timeliness

Application:

- Electronic video game toy, card-reader

Embedded operating system based approach

- Operating system based approach contains operating system which can be general purpose operating system(GPOS) or real time operating system(RTOS) to host the user written application firmware.

- GPOS design is similar to that of PC based application where the device contains an OS(windows/unix/linux,etc for desktop pc) and will be creating and running user application on top of it.
- Ex: PDA(Personal digital Assistant), HAND HELD DEVICES

- This OS based applications requires “driver software” for different hardware present on the board to communicate with them.

- **RTOS** design approach is employed in embedded products demanding real time response .
- Responses in a timely and predictable manner to events .

- RTOS allows flexible scheduling of system resources like CPU and memory and offers someway to communicate between the tasks.

Ex: Windows CE, pSOS, VxWorks

Firmware development languages:

- Assembly language based development
- High level language based development
- Mixed assembly and high level language based development

Assembly level language development:

- Assembly language is the human readable notation of ‘machine language’.
- ‘machine language’ is processor understandable language , processor only deals with binaries(1 and 0).
- Machine language is an binary representation and it consist of 1s and 0s.machine language is made readable by using specific symbols called ‘mnemonics’.
- Machine language is an interface between processor and programmer.

- Low level , system related , programming is carried out using assembly.

“Assembly language programming is the task of writing processor specific machine code in mnemonic form, converting the mnemonic to machine laguage using assembler”

Format of assembly: generally in assembly instruction is an opcode followed by operand**.**

Ex:

Label opcode operand comment

mov a , #30 ;data transfer operation

- Opcode tells the processor what to do, in the previous example it is *mov a* .

- operand provides the data and information required to perform the action by the opcode, in previous example it is #30.

- The operand may be single or dual, more.

- Label is an optional field , a ‘label’ is an identifier used to reduce the reliance of the programmers for remembering where the data or the code is located.

- Used to represent usually memory location , address of program ,sub routine ,code portion..etc . its an optional field.
- Assembly program contains an main routine and it may or may not contain the sub routine.
- Labels are used for representing subroutine names and jump locations in assembly language programming.
- **Subroutine** is small portion of code which is used frequently by the main program.
- instead of writing again and again we just call the subroutine when it is required by using required instructions.

- Example of subroutine as follows such as delay in a program.

EX1:

```
delay1: mov R0,#255
        DJNZ R1,delay1
        RET
```

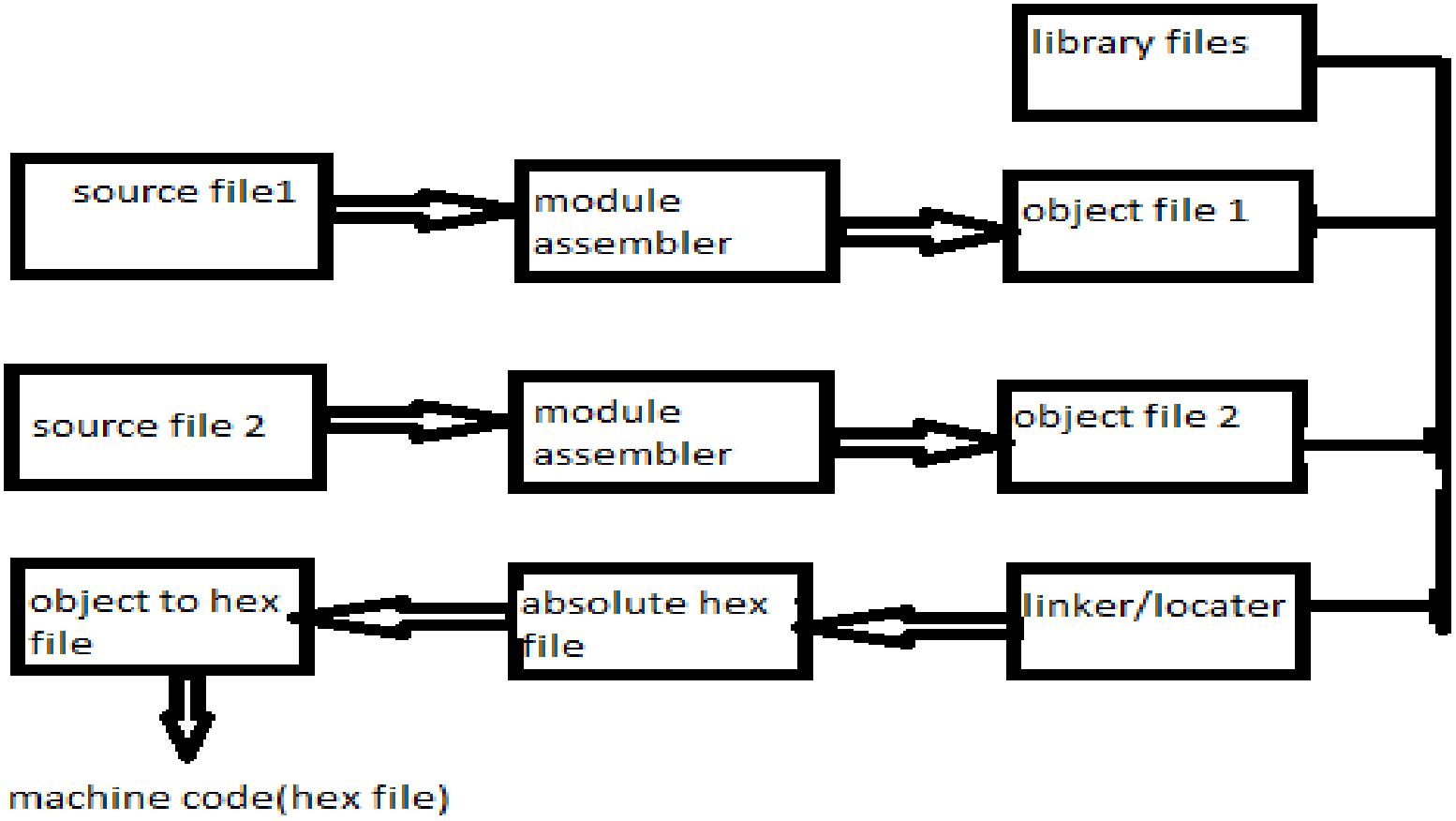
EX2:

```
delay2:
        ORG 0100H
        mov R0,#255
        DJNZ R1,0100H
        RET
```

- Here ORG 0100h is an assembler directive , assembler directive are used to determine starting address of the program and entry address of the program(ex..ORG 0100H).
- Also used for reserving memory for data variables ,arrays and structures(ex..var EQU 70h).
- Initializing variable values(ex..val data 12h).

Source file to object file translation

Assembly to machine language conversion:



- Assembler perform the translation of assembly to machine conversion which involves Source file to object file translation. Each source module is written in Assembly and is stored as .src file or .asm file.
- On successful assembling of each .src/.asm file a corresponding object file is created with extension '.obj'.
- Object file is an syntax corrected source file it does not contain absolute address of where the generated code to be placed on the program memory & it is called re-locatable segment

- Linker /locater is an another software utility responsible for "linking the various object modules in a multi module project and assigning absolute address to each module".

- Object to hex file is the final stage of the conversion converting assembly mnemonics to machine understandable code ,hex file is the representation of the machine code and the hex file is dumped into the code memory of the processor/controller.

Advantages :

- Efficient code memory and data memory usage
- High performance
- Low level hardware access

Disadvantages:

- High development time
- Developer dependency
- Non portable

Library file creation & usage

- It is formatted ordered program collection of object modules in the library
- It is some kind of source code hiding technique
- If we don't want to reveal the source code but want to distribute them to application developer for making use of them in their application
- It can be supplied as library

High level language :

- Since assembly language is most time consuming, tedious and requires skilled programmers , so the other alternative developed is high level language.
- Here compiler will take the action to convert the code written in high level(c, c++,java) to assembly level which can understood by machine.
- C is the most popularly used in embedded software, nowadays c++ is also using in emb. software.

- Various steps involved in HLL is similar to that of the assembly language except the conversion of the source file written in HLL to object file is done by cross compiler.

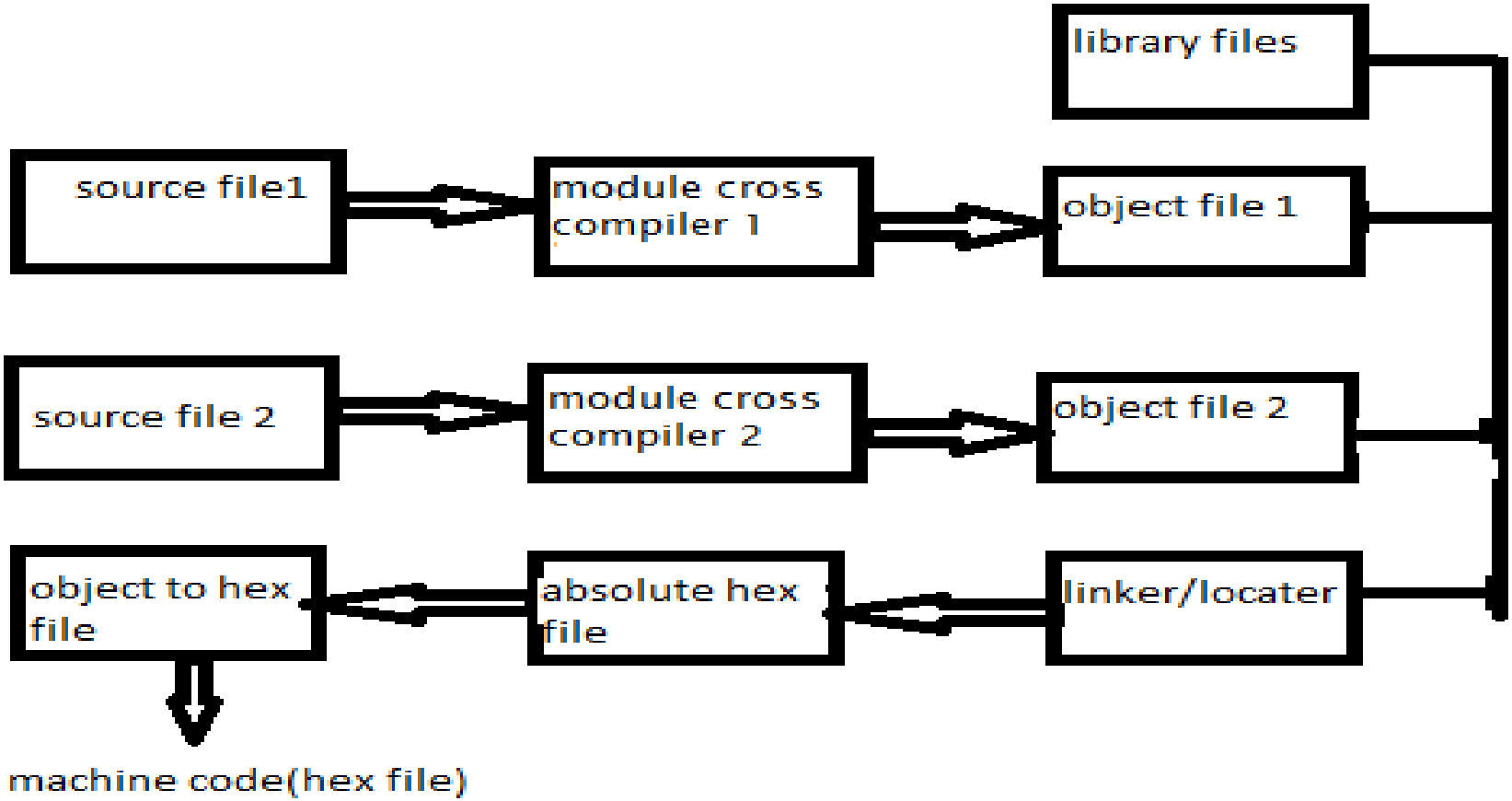
Advantages:

- Reduced development time
- Developer independency
- Portability

Limitation:

- Some cross compilers available for high level language are not so efficient in generating optimized target processor specific instructions.
- Investment is high for development tools used.

High level to machine code conversion:



Mixing assembly and high level language:

- Certain embedded firmware development require mixing both HLL and assembly , which can be done through three ways.

Mixing assembly with high level language:

- Assembly routine are mixed with the ‘c’ in that situation where the entire program is in c and the cross compiler is do not have a built in support for implementing certain features like interrupt service function(ISR) or programmer wants to take the advantage of speed and optimized code offered by the machine.

Mixing HLL with assembly:

- Useful in following scenarios
- Source code is already available in assembly language and the routine written in high level language like ‘c’ needs to be included in the existing code
- Entire source code is planned in assembly for various reasons like optimized code ,optimal performance, so that some portion of code is tedious to code in assembly.
- To include library files written in ‘c’ provided by the cross compiler . Ex.. graphics library function and string operation supported by c.

- In both of the mixing language is done by passing the parameter written in one language to other language ie..pass by value in a function.

Inline assembly mixing:

- inline assembly mixing is the another technique of inserting the targeted processor specific assembly instruction at a location of a code written in HLL ‘c’. Avoids the delay on calling an assembly routine from a c code.
- Special key words are used to indicate the start and end of assembly code. keywords are compiler specific.
- Ex..#pragma asm

  mov a,#13h

  #pragma endasm

```
int sum4(int a, int b, int c, int d){
      int t;
      __asm {
            ADD t, a, b;   // t, a, and b are virtual registers
            ADD t, c;      // Cannot directly access r0-r15
            ADD t, d;      // Use C comment style
      }
      return t;
}

int main(void){
      int s = sum4(1, 2, 3, 4);
      while(1);
}
```

C vs Embedded C

- C is well structured, well defined and standardised general purpose programming.
- ANSI(American national Standard Institute) and have various library files
- Compiler used for conversion

- Embedded C is subset of conventional C.
- Supports all C instructions and have few target processor specific instructions
- Tailored to target processor
- Cross compiler

C programming	Embedded C programming
Possesses native development in nature.	Possesses cross development in nature.
Independent of hardware architecture.	Dependent on hardware architecture (microcontroller or other devices).
Used for Desktop applications, OS and PC memories.	Used for limited resources like RAM, ROM and I/O peripherals on embedded controller.

Compiler Vs Cross-Compiler

- Compiler converts a source code written in high level language on top of particular OS running on a specific target processor.

- Cross compilers used in cross-platform applications.

The compiler running on a particular target processor converts the source code to machine code for a target processor whose architecture and instruction set is different from current development environment.

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 specifi c machine code).
- List File (.lst), Hex File (.hex), Pre-processor Output file, Map File (File extension linker dependent), Object File (.obj)

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.
- Page Header, Command Line,Source Code, Assembly Listing, Symbol Listing, Module Information, Warnings and Errors

Preprocessor Output File

- The preprocessor output file generated during cross-compilation contains the preprocessor output for the preprocessor instructions used in the source file.
- The preprocessor output file is a valid C source file.

Object 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.
- OMF51 or OMF2 are the two objects file formats supported by C51 cross compiler.
- The object file is a specially formatted file with data records for symbolic information, object code, debugging information, library references, etc.

It is the responsibility of the linker/locater to assign an absolute memory location to the object code.

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.

Map File (.MAP)

- Map file contains information about the link/locate process and is composed of a number of sections.
- It is not necessary that the map files generated by all linkers/ locaters should contain all these information. Some may contain less information compared to this or others may contain more information than given in this. It all depends on the linker/locater.

- Page Header
- Command Line
- CPU Details
- Input Modules
- Memory Map
- Symbol Table
- Inter Module Cross Reference
- Program Size
- Warnings and Errors

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.
- Intel HEX and Motorola HEX are the two commonly used hex file formats. Intel HEX file is an ASCII text file in which the HEX data is represented in ASCII format in lines.
- The lines in an Intel HEX file are corresponding to a HEX Record.
- Each record is made up of hexadecimal numbers that represent machine-language code and/or constant data.

Intel HEX file is used for transferring the program and data to a ROM or EPROM which is used as code memory storage.

- Each record is made up of five fields arranged in the following format:
- :llaaaattdd...cc

Field	Description
:	The colon indicating the start of every Intel HEX record
ll: **aaaa**: **tt**:	Record length field representing the number of data bytes (**dd**) in the record Address field representing the starting address for subsequent data in the record Field indicating the HEX record type. According to its value it can be of the following types 00: Data Record 01: End of File Record 02: 8086 Segment Address Record 04: Extended Linear Address record
dd:	Data field that represents one byte of data. A record can have number of data bytes. The number of data bytes in the record must match to the number specified by the '**ll**' field

cc:	Checksum field representing the checksum of the record. Checksum is calculated by adding the values of all hexadecimal digit pairs in the record and taking modulo 256. Resultant o/p is 2's complemented to get the checksum.

Intel hex fi le generated for “Hello World” application example is given below

```
:03000000020C1FD0
:0C0C1F00787FE4F6D8FD758121020C2BD3
:0E0C110048656C6C6F20576F726C64210A008E
:090C2B007BFF7A0C7911020862CA
:10080000E517240BF8E6051722780830070278 0B65
:10081000E475F001120BB4020B5C2000EB7F2ED2CA
:1008200008018EF540F2490D43440D4FF30040BD0
:10083000EF24BFB41A0050032461FFE518600215CD
:1008400018051BE51B7002051A30070D7808E475C2
:100BFA00B8130CC2983098FDA899C298B811F6306B
:070C0A0099FDC299F5992242
:00000001FF
```

Let’s analyse the first record

:	l	l	a	a	a	a	t	t	d	d	d	d	d	d	c	c
:	0	3	0	0	0	0	0	0	0	2	0	C	1	F	D	0

Motorola HEX File Format

- Motorola HEX file is also an ASCII text file where the HEX data is represented in ASCII format in lines.
- The lines in Motorola HEX file represent a HEX Record. Each record is made up of hexadecimal numbers that represent machine-language code and/or constant data.
- The general form of Motorola Hex record is given below.

SOR	RT	Length	Start Address	Data/Code	Checksum

In other words it can be represented as **Stllaaaaaddddd…cc**
The fields of the record are explained below.

Field	Description
SOR	Stands for Start of record. The ASCII Character 'S' is used as the Start of Record. Every record begins with the character 'S'
RT	Stands for Record type. The character't' represents the type of record in the general format. There are different meanings for the record depending on the value of 't' 0: Header. Indicates the beginning of Hex File 1: Data Record with 16bit start address 2: Data record with 24bit start address 9: End of File Record

Length (*ll*):	Stands for the count of the character pairs in the record, excluding the type and record length (Count includes the number of data/code bytes, data bytes representing start address and character pair representing the checksum). Two ASCII characters '*ll*' represent the length field .Each '*l*' in the representation can take values 0 to 9 and A to F.
Start Address (*aaaa*):	Address field representing the starting address for subsequent data in the record.
Code/Data (**dd**):	Data field that represents one byte of data. A record can have number of data bytes. The number of data bytes in the record must match to the number specified by (*ll - no. of character pairs for start address–1*)
Checksum (**cc**):	Checksum field representing the checksum of the record. Checksum is calculated by adding the values of all hexadecimal digit pairs in the record and taking modulo 256. Resultant o/p is 1's complemented to get the checksum.

DISASSEMBLER/DECOMPILER

- 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 are powerful tools for analyzing the presence of malicious codes (virus information) in an executable image.
- It is not possible for a disassembler/decompiler to generate an exact replica of the original assembly code/high level source code in terms of the symbolic constants and comments used.
- 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

- Simulator is a software tool used for simulating the various conditions for checking the functionality of the application firmware.
- The features of simulator based debugging are listed below.

1) 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. User only needs to know about the memory map of various devices within the target board and the firmware should be written on the basis of it. Firmware development can start well in advance immediately after the device interface and memory maps are finalized. This saves development time.
- **Simulate I/O Peripherals:** 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.
- It really helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behavior of the firmware under abnormal input conditions.

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 behavior.
- The debugging is developer driven and it is no way capable of creating a real time behaviour.

Emulators and Debuggers

- Debugging 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.
- 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.

various types of debugging techniques

- **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.
- The code will incorporate some indication support like lighting up an “LED (every embedded product contains at least one LED).
- 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.
- Ensure that before entering into one level up, the previous level has delivered a correct result. If the code corresponding to any functionality is found not giving the expected result, fix it by modifying the code and then only go for adding the next functionality for burning into the EEPROM.
- Combine the entire source for all functionalities together, re-compile and burn the code for the total system functioning.

Inline Breakpoint Based 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.
- Burn the corresponding hex file into the EEPROM.

Monitor Program Based Firmware Debugging

- 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;
- The monitor program always listens to the serial port of the target device and according to the command received from the serial interface it performs command specific actions like firmware downloading, memory inspection/modification, sends the debug information (various register and memory contents) back to the main debug program running on the development PC, etc.

- The first step in any monitor program development is determining a set of commands for performing various operations like fi rmware downloading, memory/ register inspection/modification, single stepping, etc.
- Once the commands for each operation is fixed, write the code for performing the actions corresponding to these commands.
- On receiving a command, examine it and perform the action corresponding to it.

- The most common type of interface used between target board and debug application is RS-232/USB Serial interface.
- After the successful completion of the ‘monitor program’ development, it is compiled and burned into the FLASH memory or ROM of the target board.
- The code memory containing the monitor program is known as the ‘ Monitor ROM’.

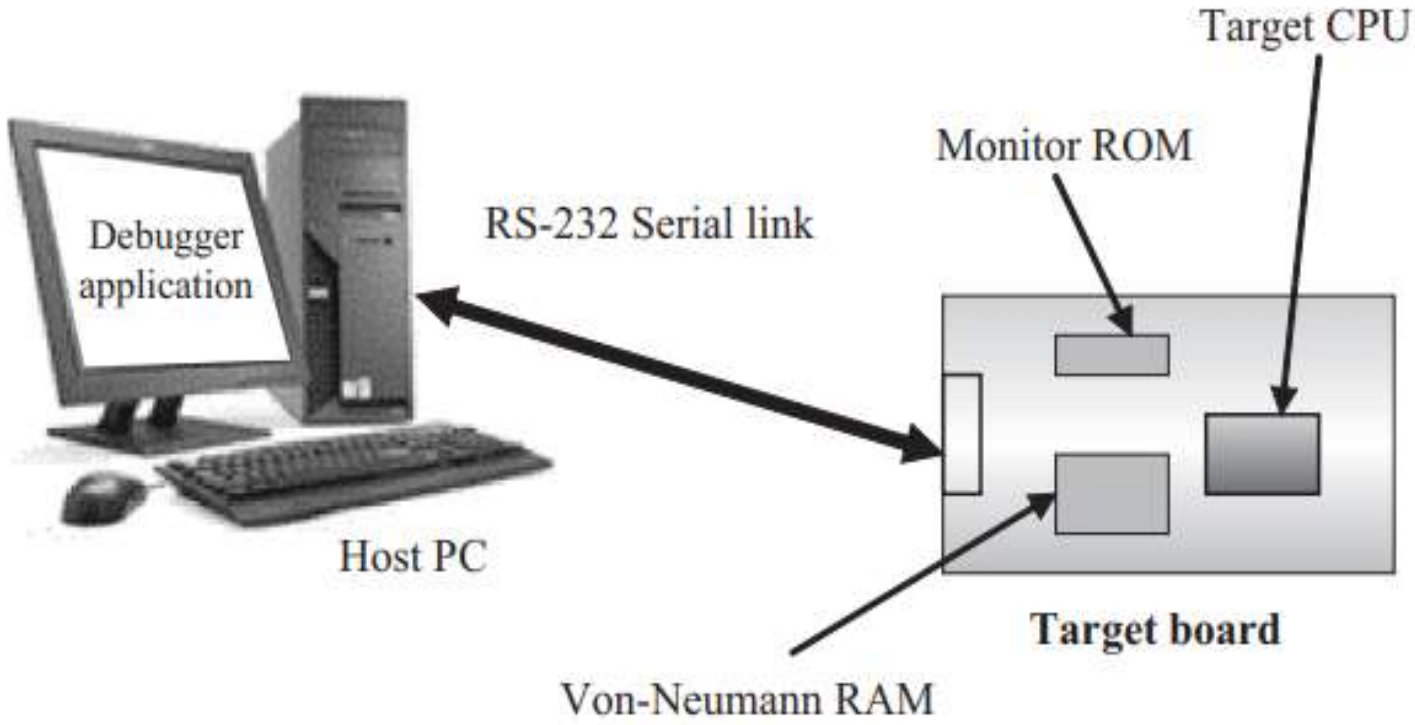


Fig. 13.39 Monitor Program Based Target Firmware Debug Setup

The monitor program features.

1. Command set interface to establish communication with the debugging application
2. Firmware download 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

The major drawbacks of monitor based debugging system are

- The entire memory map is converted into a Von-Neumann model and it is shared between the monitor ROM, monitor program data memory, monitor program trace buffer, user written fi rmware and external user memory.
- Wastage of a serial port

In Circuit Emulator (ICE) Based Firmware Debugging

- 'Emulator' is a self-contained hardware device which emulates the target CPU.
- In summary, the simulator 'simulates' the target board CPU and the emulator 'emulates' the target board CPU.
- pure software applications which perform the functioning of a hardware emulator is also called as 'Emulators' (though they are 'Simulators' in operation).
- The emulators for different families of processors/controllers are different.
- The Emulator POD forms the heart of any emulator system and it contains the following functional units

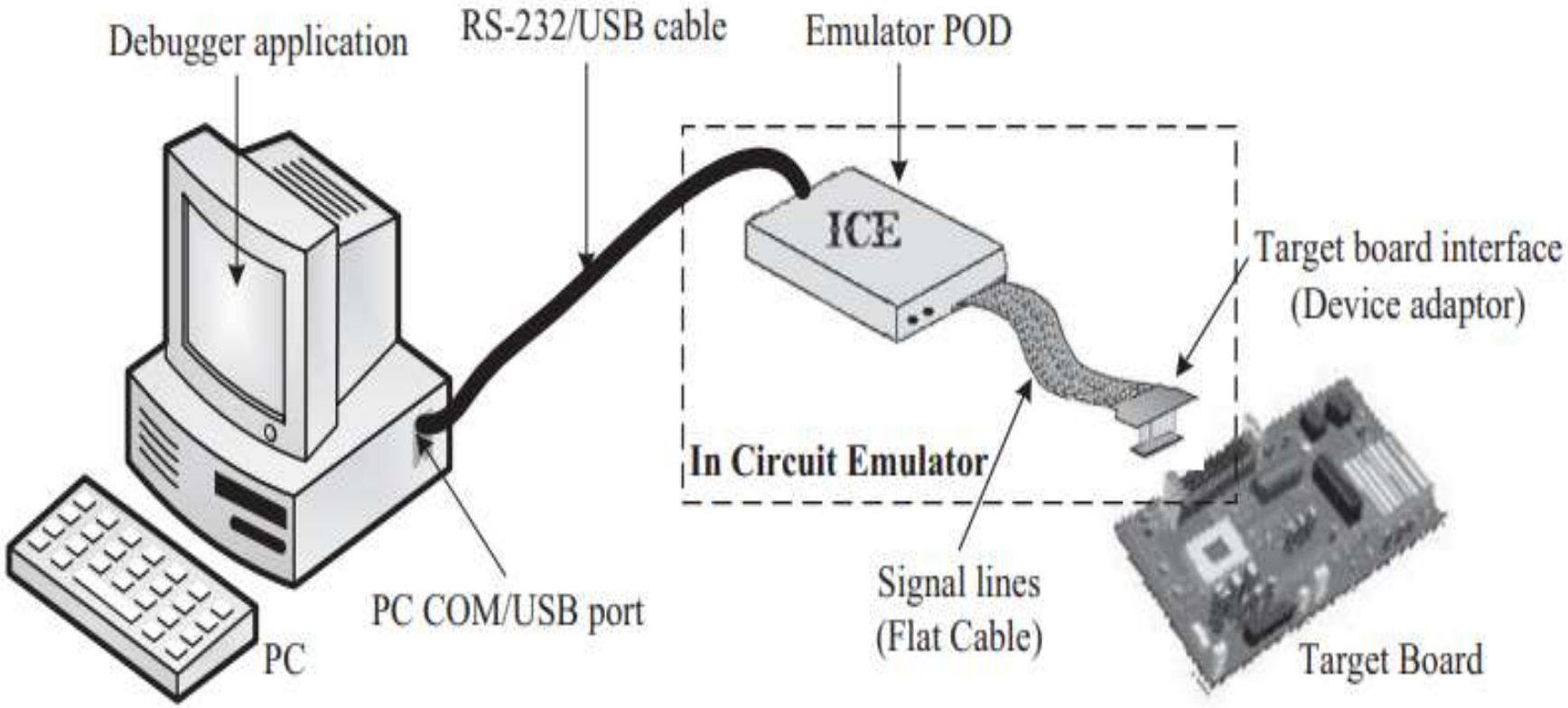


Fig. 13.40 In Circuit Emulator (ICE) Based Target Debugging

Emulation Device

- Emulation Device is a replica of the target CPU which receives various signals from the target board through a device adaptor connected to the target board and performs the execution of fi rmware 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.
- If a standard chip is used as the emulation device, the emulation will provide real-time execution behaviour ,emulator becomes dedicated to that particular device and cannot be re-used for the derivatives of the same chip.

PLD-based emulators can easily be re-configured to use with derivatives of the target CPU under consideration.

- PLD-based emulator logic is easy to implement for simple target CPUs but for complex target CPUs it is quite difficult.

Emulation Memory

- It is the Random Access Memory (RAM) incorporated in the Emulator device. It acts as a replacement to the target board's EEPROM
- Emulation memory also acts as a trace buffer in debugging.
- Trace buffer is a memory pool holding the instructions executed/registers modified/related data by the processor while debugging.
- The common features of trace buffer memory are

1)Trace buffer records each bus cycle in frames

2) Trace data can be viewed in the debugger application as Assembly/Source code

3)Trace buffering can be done on the basis of a Trace trigger (Event)

Emulator Control Logic

- Emulator control logic is the logic circuits used for implementing complex hardware breakpoints, trace buffer trigger detection, trace buffer control, etc.
- **Device Adaptors**
- Device adaptors act as an interface between the target board and emulator POD.
- Device adaptors are 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 device adaptor is usually connected to the emulator POD using ribbon cables.

- This type of emulators usually combines the entire emulation control logic and emulation device (if present) in a single board. They are known as ‘Debug Board Modules.
- Emulator hardware is partitioned into two, namely, ‘Base Terminal’ and ‘Probe Card’.
- The Base terminal contains all the emulator hardware and emulation control logic.
- The ‘Probe Card’ board contains the device adaptor sockets to plug the board into the target development board.
- The board containing the emulation chip is known as the ‘Probe Card’

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.
- 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.

The signal lines of JTAG protocol are

- Test Data In (TDI): It is used for sending debug commands serially from remote debugger to the target processor.
- Test Data Out (TDO): Transmit debug response to the remote debugger from target CPU.
- Test Clock (TCK): Synchronizes the serial data transfer.
- Test Mode Select (TMS): Sets the mode of testing.
- Test Reset (TRST): It is an optional signal line used for resetting the target CPU