MODULE 4: Embedded Firmware Design and Development: Embedded Firmware Design Approaches, Embedded Firmware Development Languages

Embedded System Development Environments: Types of files generated on cross compilation, disassemble/ decompliler, Simulators, Emulators and Debugging

EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT

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. For most of the embedded products, the firmware is stored at a permanent memory (ROM) and they are non-alterable by end users.

Embedded firmware development process starts with 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.

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 are:

- Super Loop Based Approach (Conventional Procedural Based Design)
- Embedded Operating System (OS) Based Approach

Super Loop Based Approach

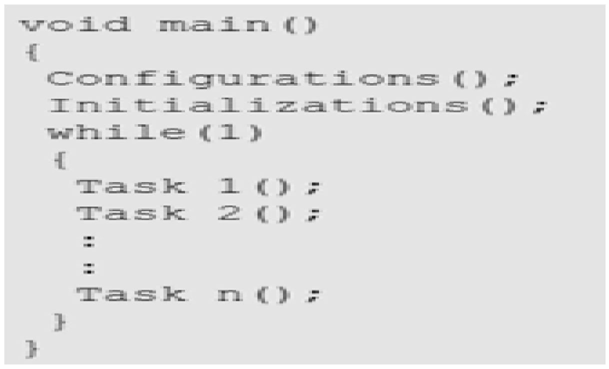
This approach is adopted for applications that are not time critical & where the response time is not so important. This approach is similar to a conventional procedural programming where the code is executed task by task. The task listed at the top of the program code is executed first and the tasks 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:

1. Configure the common parameters and perform initialization for various hardware components memory, registers, etc.
2. Start the first task and execute it
3. Execute the second task
4. Execute the next task

5. :
6. :
7. Execute the last defined task
8. Jump back to the first task and follow the same flow

The order in which the tasks to be executed are fixed & they are hard coded in the code itself. Also the operation is an infinite loop based approach. We can visualize the operational sequence listed above in terms of a '*C*' program code as shown:



```
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 this approach is called as **'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.
- The code for performing these tasks will be residing in the code memory without an operating system image.

Applications & Examples 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

A typical example of a 'Super loop based' product is an electronic video game toy containing keypad and display unit.

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.

- *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 operating system 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 GPOS based design is similar to a conventional PC based application development where the device contains an operating system and you will be creating and running user applications on top of it. Example of a GPOS is Microsoft Windows XP Embedded. Examples of Embedded products using Microsoft Windows XP OS are Personal Digital Assistants (PDAs), Hand held devices/Portable devices. 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. RTOS contains a Real Time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc. A RTOS allows flexible scheduling of system resources like the CPU and memory and offers some way to communicate between tasks. 'Windows CE', 'pSOS', 'ThreadX', 'MicroC/OS-II', 'Embedded Linux', 'Symbian', etc. are examples of RTOS employed in embedded product development. Mobile phones, 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 human readable notation of '**machine language'**, whereas 'Machine Language' 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'**. Machine Language can be considered as interface between processor & 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 data using an assembler.

The general format of an assembly language instruction is an Opcode followed by Operands. The Opcode tells the processor/ controller what to do and the Operands provide the data and information required to perform the action specified by the opcode.

Example: MOV A, #30 // Here MOV A is the Opcode and 30 is the operand.

Assembly language instructions are written one per line. Each line of an assembly language program is split into four fields as given below:

LABEL OPCODE OPERANDS COMMENT

LABEL is an optional field. A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers or remembering where data or code is located.

The sample code given below using 8051 Assembly language illustrates the structured assembly language programming.

```
DELAY: MOV R0, #255 ; Load Register R0 with 255.
            DJNE R0, DELAY ; Decrement R0 and loop till R0 = 0.
            RET ; Return to calling program.
```

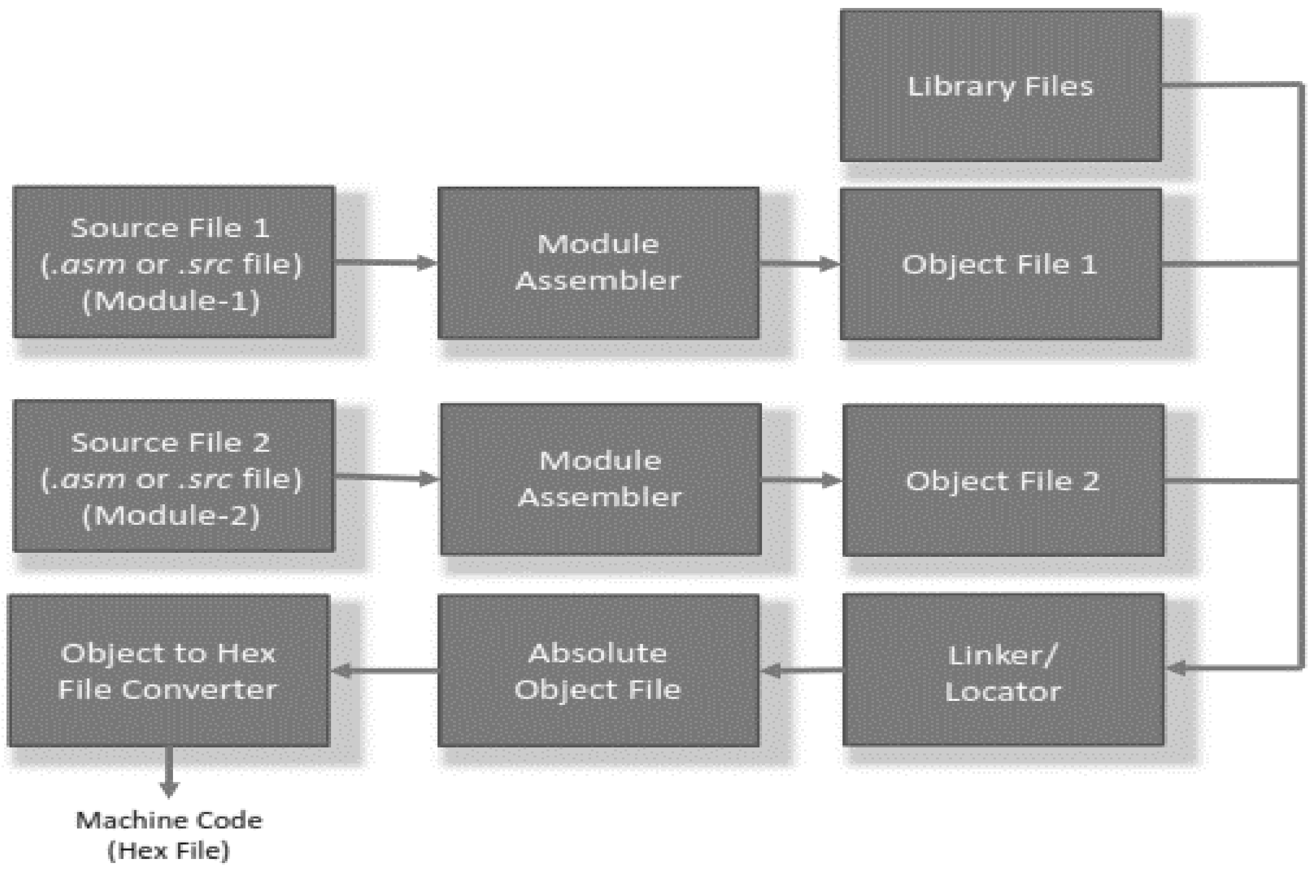
The Assembly program contains a main routine and it may or may not contain subroutines. The example given above is a subroutine, which can be invoked by a main program by the assembly instruction: LCALL DELAY. Executing this instruction transfers the program flow to the memory address referenced by the 'LCALL DELAY'.

The Assembly language program written in assembly code is saved as .asm (Assembly file) file or an .src (source) file. Any text editor like 'notepad' or 'WordPad' or the text editor of an Integrated Development (IDE) tool can be used for writing the assembly instructions.

When a program is too complex or too big; the entire code can be divided into sub-modules and each module can be re-usable. This concept is called as Modular Programming. 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. Various steps involved in the conversion of a program written in assembly language to corresponding binary file/machine language are shown below:



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. When the linker processes a library, only those object modules in the library that are necessary to create the program are used. Library files are generated with extension '.lib'. 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 controller.

Linker and Locater: Linker and Locater is a software utility responsible for "linking the various object modules in a multi-module project and assigning absolute address to each module". Linker is a program which combines the target program with the code of other programs (modules) and library routines. During the process of linking, the **absolute object module** is created.

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 Optimization):** Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations. This leads to less utilization of code memory and efficient utilization of data memory.
- **High Performance:** Optimized code not only improves the code memory usage but also improves the total system performance. Through effective assembly coding,

optimum performance can be achieved for 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'.

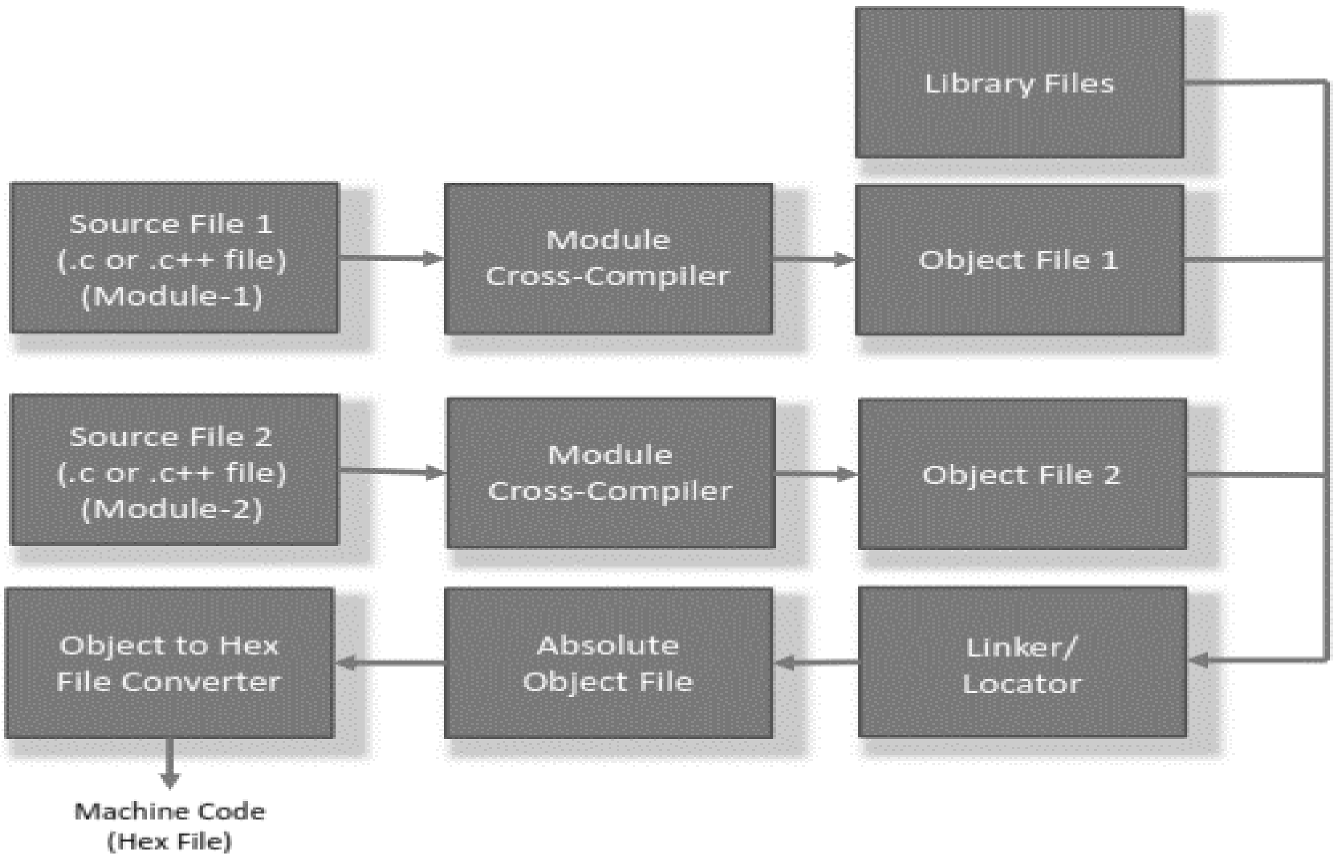
Drawbacks of Assembly Language Based Development:

- ***High Development Time:*** Assembly language is much harder to program than high level languages. The developer must have thorough knowledge of the architecture, memory organization 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.
- ***Developer Dependency:*** Unlike high level languages, there is no common written rule for developing assembly language based applications. In assembly language programming, the developers 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.
- ***Non-Portable:*** Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/ controllers. If the target processor/ controller changes, a complete re-writing of the application using the assembly instructions for the new target processor/ controller is required.

High Level Language (HLL) 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. Commonly used HLL for embedded firmware application development is 'C'. 'C' is well defined, easy to use HLL with extensive cross platform development tool support. Nowadays embedded developers are making use of cross-compilers 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. The various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language is illustrated in below figure:



The program written in any of the high level language 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 supporting the high level language can be used for writing the program. Most of the HLLs support modular programming approach and hence can have multiple source files called modules. Translation of high level source code to executable object code is done by a cross-compiler. Cross-compilers for different HLLs for same target processor are different. C51 is a popular Cross-compiler available for 'C' language for the 8051 family of micro controller. Conversion of each module's source code to corresponding object file is performed by the cross compiler. Rest of the steps 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 organization and register details of the target processor in use and syntax of the HLL are the only pre-requisites for HLL based firmware development.

  With HLL, 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 HLLs 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.
- ***Portability:*** Target applications written in HLLs are converted to target processor / controller understandable format (machine codes) by cross-compiler.

  An application written in HLL for a particular target processor can easily be converted to another target processor/ controller specific application, with no/little modification.

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 optimized target processor specific instructions.
- **Not Suitable for Low Level Hardware:** HLL based code 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 HLL based development tools (Integrated Development Environment incorporating cross-compiler) is high compared to Assembly Language based firmware development tools.

TYPES OF FILE GENERATED ON CROSS COMPILATION

Cross compilation is the process of converting the source code written in high level language to target processor/controller understandable machine code. Following are some of the files generated upon cross compilation:

List file (.lst), Preprocessor output file, Object file .obj, Map file (.map), & Hex file (.hex)

1. List File (.LST File)

Listing file is generated during the cross-compilation process and it contains an 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 contain the following sections:

Page Header: A header on each page of the listing file indicates the compiler version name, source file name, Date, Time and Page No.

Example: C51 COMPILER V8.02 SAMPLE 05/23/2006 11:12:58 PAGE 1

Command Line: Represents the entire command line that was used for invoking the compiler.

COMPILER INVOKED BY: C:\Keil\C51\BIN\C51.EXE sample.c

Source Code: The source code listing outputs the line numbers as well as the source code on that line. Apart from source code lines, the list file will also include the comments in the source file.

Assembly listing: It contains the assembly code generated by compiler for the given 'C' source code.

Symbol listing: It contains symbolic information about the various symbols present in the cross compiled source file. Symbol listing contains: name (NAME), symbol classification (CLASS: SFR, structure, typedef, static, public, auto etc.), memory space (MSPACE: Code memory or data memory)), data type (TYPE: int, char, call etc.), offset address (OFFSET: Code memory start address), size in bytes (SIZE).

Module Information: The module information provides the size of initialized and un-initialized memory areas defined by the source file.

Warnings and Errors: This section of list file records the errors encountered or any statement that may create issues in application (Warnings), during cross compilation.

C51 COMPILATION COMPLETE. 0 WARNING(S), 0 ERROR(S).

2. Preprocessor output file

The preprocessor output file generated during cross compilation contains preprocessor output for preprocessor instructions used in the source file. This file is used for verifying the operation of Macros and conditional preprocessor directive. The preprocessor output file is a valid C source file.

3. Object file (.OBJ File)

Cross-compiling/assembling each source module converts the various Embedded C/ Assembly instructions and other directives present in the module to an object (OBJ) file. The format of the .OBJ file is cross compiler dependent. OMF51 or OMF2 are the two objects file formats supported by C51 cross compiler. 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 synchronise source lines with object code.

The object code present in the object file are not absolute, meaning, the code is not allocated fixed memory location in code memory. It is the responsibility of the linker/locater to assign an absolute memory location to the object code.

4. Map File (.MAP)

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'.

Map file contains information about the link/locate process and is composed of a number of sections. The different sections listed in a map file are cross compiler dependent. The information generally held by map files is listed below

Page Header: A header on each page of the linker listing (MAP) file which indicates the linker version number, date, time, and page number.

e.g. BL51 BANKED LINKER/LOCATER V6.22 10/16/2014 15:47:10 PAGE 1

Command Line: Represents the entire command line that was used for invoking the linker.

BL51 BANKED LINKER/LOCATER V6.22, INVOKED BY: C:\KEIL V5\C511 BIN\BL51.EXE sample.obj, STARTUP.obj TO Sample

CPU Details: Details about the target CPU and memory model (internal data memory, external data memory. paged data memory, etc.) come under this category.

e.g. MEMORY MODEL: SMALL

Input Modules: It includes the names of all the object files, library files and other files that are included in the linking process.

Memory Map: It lists the starting address, length, relocation type and name of each segment in the program.

Symbol Table: It contains the name, value and type for all symbols from different input modules.

Inter Module Cross Reference: The cross reference listing includes the section name, memory type and module names in which it is defined and all modules where it is accessed.

Program Size: It contains the size of various memory areas, constants and code space for the entire application.

Warnings and Errors: It contains the warnings and errors that are generated while linking a program. It is used in debugging link errors.

LINK / LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S).

5. Hex file (.HEX File)

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. Hex files embed the machine code in a particular format. The format of Hex file varies across the family of processors/controllers.

Intel HEX and Motorola HEX are the two commonly used hex file formats in embedded applications.

1. Intel Hex File Format: Intel HEX file is composed of a number of hex records. Each record is made up of five fields in the following format:

:llaaaattdd....cc

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits (which make up a byte) as described below:

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.

2. Motorola HEX file format: Motorola file is 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: **Stllaaaadddd…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. Two ASCII characters 'Il represent the length field. Each 'l' 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'
Check-sum (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.

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/cross assembling.

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 deployed in reverse engineering.

Disassemblers/ Decompilers are powerful tools for analysing the presence of malicious codes (virus information) in an executable image. Disassemblers/Decompilers are available as either freeware tools readily available for free download from internet or as commercial tools.

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 help in debugging the firmware for checking its required functionality. In certain scenarios, simulator refers to a soft model (GUI model) of the embedded product. 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.

Simulators

Simulators simulate the target hardware and the firmware execution can be inspected using simulators. 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 behaviour

Advantages of Simulator Based Debugging: Simulator based debugging techniques are simple and straightforward. The major advantages of simulator based firmware debugging techniques are:

- **No Need for Original Target Board:** Simulator based debugging technique is purely software oriented. IDE's software support simulates the CPU of the target board. Since the real hardware is not required, firmware development can start well in advance and this saves development time.
- **Simulated I/O Peripherals**: Simulator provides the option to simulate various I/O peripherals. Hence it eliminates the need for connecting I/O devices for debugging the firmware.
- **Simulates Abnormal Conditions**: It helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behaviour of the firmware under abnormal input conditions.

Limitations of Simulator based Debugging: Though simulation based firmware possess certain limitations and we cannot fully rely upon the simulator-based firmware debugging. Some of the limitations of simulator-based debugging are:

- **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. 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 moreover in a real application the I/O condition may be varying or unpredictable. Simulation goes for simulating those conditions for known values.

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. Debugging process in embedded application is broadly classified into two:

- 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.

Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behavior. Types of firmware debugging are:

1. Incremental EEPROM Burning Technique
2. Inline Breakpoint Based Firmware Debugging
3. Monitor Program Based Firmware Debugging
4. In Circuit Emulator (ICE) Based Firmware Debugging
5. On Chip Firmware Debugging (OCD)

1. Incremental EEPROM Burning Technique

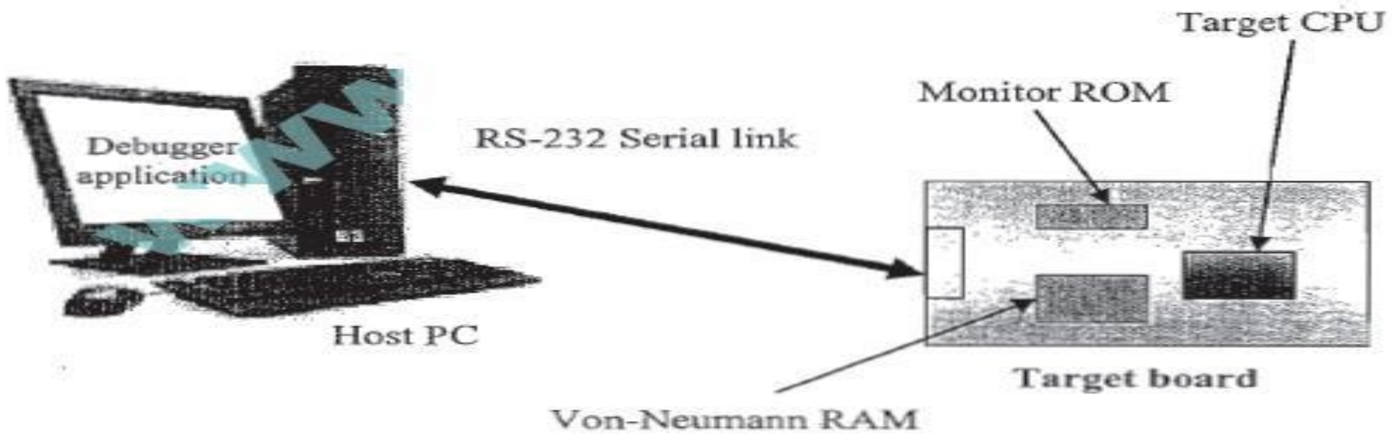
This is the most primitive type of firmware debugging technique. In this, the code is separated into different functional code units. The entire code is not burnt into the EEPROM chip at once. The code is burned in incremental order. The code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.

2. 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. Cross-compile the source code with the debug codes embedded within it. Burn the corresponding hex file into the EEPROM.

3. 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; 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.



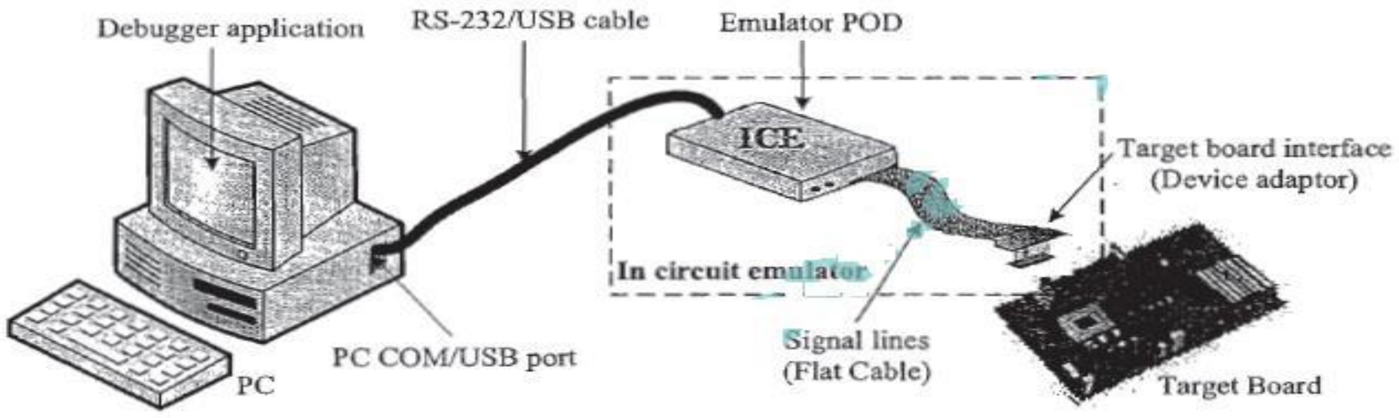
The first step in any monitor program development is determining a set of commands for performing various operations like firmware downloading, memory/ register inspection/ modification, single stepping, etc. The entire code stuff handling the command reception and corresponding action implementation is known as the "***monitor program***". The most common type of interface used between target board and debug application is RS-232C Serial interface.

The monitor program contains the following set of minimal 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.

4. 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 POD (shown in figure) forms the heart of any emulator system and it contains the following functional units:



- ***Emulation Device:*** is a replica of the target CPU which receives various signals from the target board through a device adaptor and performs the execution of firmware under the control of debug commands from the debug application.
- ***Emulation Memory:*** is the RAM incorporated in the Emulator device. It acts 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 hardware breakpoints, trace buffer trigger detection, trace buffer control, etc. They are also used for implementing logic analyzer functions in advanced emulator devices. The 'Emulator POD' is connected to the target board through a 'Device adaptor' and signal cable.
- ***Device Adaptors:*** act as an interface between the target board & emulator POD. Device adaptors are used for routing the various signals from pins assigned for the target processor. The device adaptor is connected to the emulator POD using ribbon cables.

5. On Chip Firmware Debugging (OCD)

Today almost all processors/controllers incorporate built in debug modules called On Chip Debug (OCD) support. It is a very good feature supporting fast and efficient firmware debugging. The On Chip Debug facilities integrated to the processor/ controller are chip vendor dependent and most of them are proprietary technologies. 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. OCD module implements dedicated registers for controlling 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.

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.

The signal lines of JTAG protocol are GIVEN below:

- Test Data In (TDI)
- Test Data Out (TDO)
- Test Clock (TCK)
- Test Mode Select (TMS)
- Test Reset (TRST)