Integration and Testing of Embedded Hardware and Firmware

Integration testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development. The final embedded hardware constitute of a PCB with all necessary components affixed to it as per original schematic diagram. Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirements on the product. Embedded firmware will be in a target processor/controller understandable format called machine language (sequence of 1's and O's-Binary). The target embedded hardware without embedding the firmware is a dumb device and cannot function properly. If you power up the hardware without embedding the firmware, the device may behave in an unpredicted manner.

Both embedded hardware and firmware should be independently tested to ensure their proper functioning. Functioning of individual hardware sections can be done by writing small utilities which checks the operation of the specified part. The targeted functionalities of the embedded firmware can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE. By simulating the firmware, the memory contents, register details, status of various flags and registers can easily be monitored and it gives an approximate picture of "What happens inside the processor/controller and what are the states of various peripherals" when the firmware is running on the target hardware.

Integration of Hardware and Firmware

Integration of hardware and firmware deals with the embedding of firmware into the target hardware board. It is the process of 'Embedding Intelligence' to the product. The embedded processors/controllers used in the target board may or may not have built in code memory.

If the processor/controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/controller, an external dedicated EPROM/ FLASH memory chip is used for holding the firmware. This chip is interfaced to the processor/controller.

A variety of techniques are used for embedding the firmware into the target board. The commonly used firmware embedding techniques for a non-OS based embedded system are explained below.

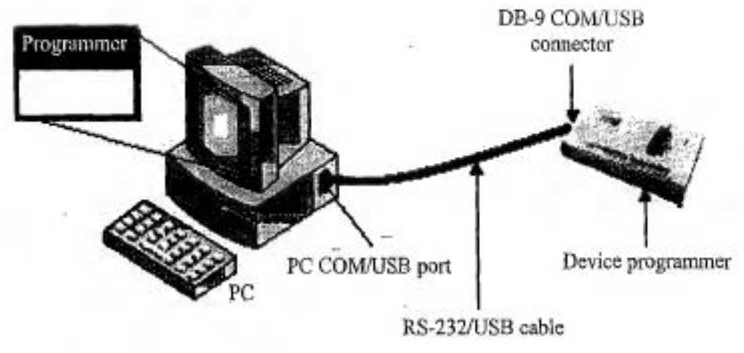
1. Out of Circuit Programming
2. In System Programming (ISP)
3. In Application Programming (IAP)
4. Use of Factory Programmed Chip
5. Firmware Loading for Operating System Devices

1. Out of Circuit Programming

Out-of-circuit programming is performed outside the target board. The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and it is programmed with the help of a programming device.

The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals. It contains a ZIF socket with locking pin to hold the device to be programmed. The device will be under the control of a utility program running on a PC.

The commands to control the programmer are sent from the utility program to the programmer through the interface in the figure.



The sequence of operations for embedding the firmware with a programmer is listed below.
1. Connect the programming device to the specified port of PC (USB/COM port/parallel port)
2. Power up the device.
3. Execute the programming utility on the PC and ensure proper connectivity is established between PC and programmer. In case of error, turn off device power and try connecting it again
4. Unlock the ZIF socket by turning the lock pin
5. Insert the device to be programmed into the open socket as per the insert diagram shown on the programmer

6. Lock the ZIF socket
7. Select the device name from the list of supported devices
8. Load the hex file which is to be embedded into the device
9. Program the device by 'Program' option of utility program
10. Wait till the completion of programming operation.
11. Ensure that programming is successful by checking the status LED on the programmer
12. Unlock the ZIF socket and take the device out of programmer.

The major drawback of out-of-circuit programming are

- High development time: Whenever the firmware is changed, the chip should be taken out of the development board for re-programming. This is tedious and prone to .chip damages due to frequent insertion and removal.
- The programmer facilitates programming of only one chip at a time and it is not suitable for batch production. Using a 'Gang Programmer' resolves this issue to certain extent. A gang programmer is similar to an ordinary programmer except that it contains multiple ZIF sockets (4 to 8) and capable of programming multiple devices at a time.
- Once the product is deployed in the market in a production environment, it is very difficult to upgrade the firmware.

The out-of-system programming technique is used for firmware integration for low end embedded products which runs without an operating system. Out-of-circuit programming is commonly used for development of low volume products and Proof of Concept (PoC) product Development.

2. In System Programming (ISP)

With ISP, programming is done 'within the system', meaning the firmware is embedded into the target device without removing it from the target board. It is the most flexible and easy way of firmware embedding. The only pre-requisite is that the target device must have an ISP support.

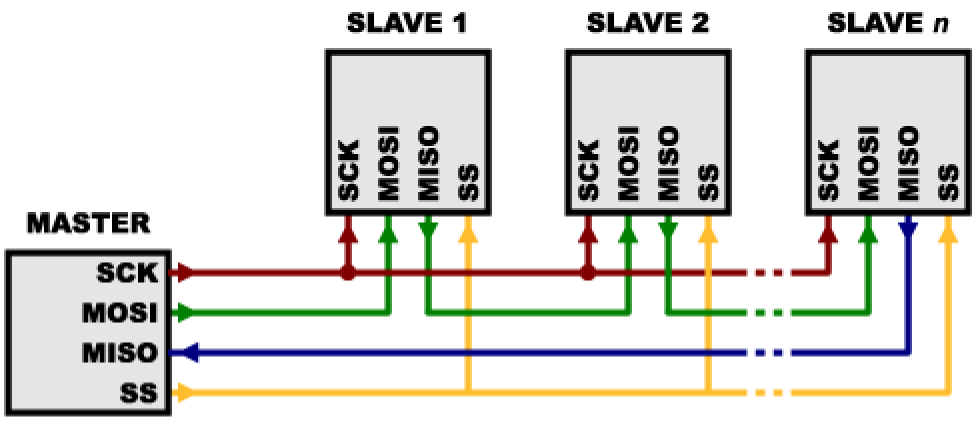
The communication between the target device and ISP utility will be in a serial format. The serial protocols used for ISP may be **Joint Test Action Group** (JTAG) or **Serial Peripheral Interface** (SPI) or any other proprietary protocol.

2.1 In System Programming with SPI Protocol

Devices with SPI In System Programming support contains a built-in SPI interface and the on-chip EEPROM or FLASH memory is programmed through this interface.

The primary I/O lines involved in SPI - In System Programming are listed below.

- MOSI - Master Out Slave In
- MISO - Master In Slave Out
- SCK-System Clock
- RST - Reset of Target Device
- GND - Ground of Target Device



Serial Peripheral Interface Protocol

PC acts as the master and target device acts as the slave in ISP. The program data is sent to the MOSI pin of target device and the device acknowledgement is originated from the MISO pin of the device. SCK pin acts as the clock for data transfer.

A utility program can be developed on the PC side to generate the above signal lines. Since the target device works under a supply voltage less than 5V, it is better to connect these lines of the target device with the parallel port of the PC.

The key player behind ISP is a factory programmed memory (ROM) called **Boot ROM**. The Boot ROM normally resides at the top end of code memory space. It contains a set of Low-level Instruction APIs and these APIs allow the processor/controller to perform the FLASH memory programming, erasing and reading operations. The contents of the Boot ROM are provided by the chip manufacturer and the same is masked into every device. Firmware upgrades for products supporting ISP is quite simple.

In Application Programming (IAP)

In Application Programming (IAP) is a technique used by the firmware running on the target device for modifying a selected portion of the code memory. It is not a technique for first time embedding of user written firmware. It modifies the program code memory under the control of the embedded application. Updating calibration data, look-up tables, etc., which are stored in code memory, are typical examples of IAP.

The Boot ROM resident API instructions which perform various functions such as programming, erasing, and reading the Flash memory during ISP-mode, are made available to the end-user written firmware for IAP.

The Boot ROM is shadowed with the user code memory in its address range. This shadowing is controlled by a status bit. When this status bit is set, accesses to the internal code memory in this address range will be from the Boot ROM. When cleared, accesses will be from the user’s code memory. Hence the user should set the status bit prior to calling the common entry point for IAP operations.

Use of Factory Programmed Chip

It is possible to embed the firmware into the target processor/controller memory at the time of chip fabrication itself. Such chips are known as **Factory programmed chips**. Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory.

Factory programmed chips are convenient for mass production applications and it greatly reduces the product development time. It is not recommended to use factory programmed chips for development purpose where the firmware undergoes frequent changes. Factory programmed ICs are bit expensive.

Firmware Loading for Operating System Devices

The OS based embedded systems are programmed using the In System Programming (ISP) technique. It contain a special piece of code called 'Boot loader' program. The features of boot loader program are as follows:

- Takes control of the OS and application firmware embedding and copy the OS image to the RAM of the system for execution.
- Contains necessary drivers for initializing the supported interfaces like UART, TCP/IP.
- Implements menu options for selecting the source for OS image to load
- In case of the network based loading, the bootloader broadcasts the target's presence over the network and the host machine on which the OS image resides can identify the target device by capturing this message. Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

Embedded System development environment

Components of Embedded development environment

- Host Computer: Acts as the heart of development environment.
- IDE Tools: Tools for firmware design and development
- Electronic Design Automation Tools: Embedded Hardware Design
- Emulator hardware: Debugging target board
- Signal Sources (function generator): Simulates inputs to target board
- Target Hardware Debugging tools: CRO, Multimeter ,Logic Analyser for debugging hardware
- Target Hardware

IDE

In Embedded System, IDE stands for an integrated environment for developing and debugging the target processor specific embedded firmware. An IDE is also known as integrated design environment or integrated debugging environment. IDE is a software package which bundles a Text Editor, Cross-compiler, Linker and a Debugger. IDE is a software application that provides facilities to computer programmers for software development. IDEs can either command line based or GUI based. IDE consists of

1. Text Editor or Source code editor
2. A compiler and an interpreter
3. Build automation tools
4. Debugger
5. Simulators
6. Emulators and logic analyzer

An example of IDE is Turbo C/C++ which provides platform on windows for development of application programs with command line interface. The other category of IDE is known as Visual IDE which provides the platform for visual development environment, ex- Microsoft Visual C++. IDEs used in embedded firmware are slightly different from the generic IDE used for high level language based development for desktop applications. In Embedded applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open source.

Cross Compiler/ Cross Assembler

Cross compilation is the process of converting a source code written in high level language to a target processor/controller understandable machine code. The conversion of the code is done by software running on a processor/controller which is different from the target processor. The software performing this operation is referred as the Cross-compiler. In other words cross-compilation the process of cross platform software/firmware development.

Cross Assembling is similar to cross-compiling; the only difference is that the code written in a target processor/controller specific Assembly code is converted into its corresponding machine code. The application converting assembly instruction to target processor/ controller specific machine code is known as cross-assembler.

Need for Cross Compiler

There are several advantages of using cross compiler some of them are as below:

- By using cross compliers we can not only develop complex embedded system but also reliability can be improved and maintenance is easy.
- Knowledge of the processor instruction set is not required.
- Register allocation and addressing mode details are managed by the compiler.
- The ability to combine variable selection with specific operations improves program readability.

Different types of file generated during cross compilation

1. List file (.LST file)

Listing file is generating during cross compilation process. It contain information about the cross compilation process like cross compiler details, formatted source text, assembly code generated from source file, symbol tables, error and warning during the cross compilation process. The list file generated contain the following section

Page Header: A Header on each page indicates the compiler version number, source file name, data, time and page number.

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

Source Code: The source code listing outputs the line number as well as the source code on that line. Special cross compiler directives are used to include or exclude the conditional code in the source code listings. Special cross compiler directives can be used to include the entire contents of the include file in the list file.

Assembly Listing: It contains the assembly code generated by the cross compiler for the C source code.

Symbol Listing: It contains symbolic information about the various symbols present in the cross compiled source file. Symbol listing contain symbol name (NAME), symbol classification (CLASS), memory space (MSPACE), data type (TYPE), offset (OFFSET) and size in bytes (SIZE).

Module Information: The module information provide the size of initialized and uninitaialized memory areas defined by the source file.

Warning and Errors: It records the errors encountered or any statement that may create issue in application during cross compilation. We can ignore certain warning (eg: local variable is declared in a function and it is not used anywhere in the program). Certain warning require prompt attention.

2. Preprocessor Output File

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

3. Object File (.OBJ File)

It is the lowest level file format for any platform. Cross compiling each source module converts the various Embedded instructions and other directives present in the module to an object(.OBJ) file. The object file is specially formatted file with data records for symbolic information, object code, debugging information etc. OMF 1 & OMF2 are the 2 object files supported by C51 Cross compiler. List of details included in object file are

1. Reserved memory for global variables
2. Public symbol (variable or function)names
3. External symbol (variable or function)references
4. Library files with which to link
5. Debugging information to help synchronize source lines with object files

During cross compilation process, the cross compiler sets the address of references to external variables and functions as 0. The external references are resolved by the linker during the linking process. Hence it is obvious that the code generated by the cross compiler is not executable without linking it for resolving external references.

4. Map File (.MAP)

Object file created contains relocatable codes that is their location in memory is not fixed. It is the responsibility of linker to link these object modules. The locator is responsible for locating the absolute address to each module in the code memory. Map files are generated by the linker and loader. These files are used to keep the information of linking and locating process. Map file contains information about the link/locate process and is composed of a number of sections.

The different sections in the map file are as follows:

Page Header: It indicates the linker version, date, time and page number.

Command Line: Represent the entire command line used for invoking linker.

CPU Details: It contain details of target CPU and memory model.

Input Modules: It includes the name of all object modules, library file and modules 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 value, type and name for all symbols from different input modules.

Inter Module Cross Reference: It includes the section name, memory type and the name of the module in which it is defined and all modules in which it is accessed.

Program Size: It contain the size of various memory areas as well as constant and code space for the entire application.

Warning and Errors: Errors and warnings generated while linking a program are written to this section. It is very useful in debugging link errors.

5. Hex File (.Hex file)

Hex file is the binary executable file created from source 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 two commonly used hex file formats in embedded applications.

Intel HEX file format

It is an ASCII text file in which the HEX data is represented in ASCII format in lines. The lines in an Intel HEX files are corresponding to HEX record. Each record is made up of hexadecimal number 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**
:	Start of every Intel HEX record
ll	Record length in data bytes
aaaa	Address field representing the start address for subsequent data in the record.
tt	Record type 00: Data Record; 01: End of File Record; 10: 8086 Segment Address Record; 11: Extended Linear Address Record.
dd	Data field that represent one byte of data.
cc	Checksum field representing the checksum of the 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

: indicates the start of a new record. ll(03) gives the number of data bytes in the record. The start address (aaaa) of data in the record is 0000H. The record type byte tt for this record is 03. The data for above record are 02, 0C and 1F. They are supposed to place at three consecutive memory locations in the EPROM with starting address 0000H.

Motorola HEX file Format

It is an ASCII text file where 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 following fields

Field	**Description**
SOR	Start of Record.
RT	Record Type. 0: Header; 1: data Record with 16 bit start address; 2: Data record with 24 bit start address; 9: End of File Record
Length (ll)	Stands for count of character pairs in the record.
StartAddress (aaaa)	Starting address of subsequent data in the record.
Code/Data (dd)	Data field that represent one byte of data.
Checksum (cc)	Checksum field representing the checksum of the record.

Disassembler / Decompiler

Disassembler is the utility program that convert machine code into assembly code. It is complementary to assembly or cross assembly. Decompiler is a utility program that convert machine language instruction to high level language instruction. It performs reverse operation of compiler or cross compiler.

Both are reverse engineering tools. Reverse engineering is a technology used to reveal the technology behind the working of a product. It is used to find out the secret behind popular proprietary product. It helps the reverse engineering process by translating embedded firmware to assembly /high level instruction. These are powerful tools for analyzing the presence of malicious contents in an executable image. They are available as either freeware tool or as a commercial tool. They generate a source code which is somewhat matching to the original source code from which binary code is generated.

Simulator

Simulator is a software tool for simulating various functionality of the application software. IDE provides simulator support. Simulator simulates target hardware and firmware execution can be simulate using simulators. The features of Simulators are as follows

- Purely software based
- Doesn't require a real target system
- Very primitive
- Lack of real time behavior

Advantage of Simulator Based Debugging

1. No need of target board: It is purely software oriented, IDE simulates the target board. Since real hardware is not needed we can start immediately after the device interface and memory maps are finalized this saved development time.

2. Simulated I/O peripherals: It eliminates the need for connecting IO devices for debugging the firmware.

3. Simulates abnormal conditions: It can input any parameter as input during debugging hence can check for abnormal conditions easily.

Limitation of Simulator based Debugging

1. Deviation from real behavior: It is always carried in a development environment where developers may not be able the debug the firmware under all possible combination of input.

2. Lack of Real Timeliness: The major limitation of simulator based debugging is that it is not real time in behavior.

Debugging

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.

Debugging is 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.

Debugger is a special program used to find errors or bugs in other programs. It allows a programmer to stop a program at any point and examine and change the values of the variables.

The various debugging techniques used are as follows:

1. Incremental EEPROM Burning Techniques

This is the most primitive type of firmware debugging technique where the code is separated into different functional units. Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremented order. The code will be incorporate some indication support like lighting up an LED.

If the first functionality is found working perfectly on the target board with the corresponding code is 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 are working properly, combine the entire source for all functionalities together, recompile and burn the code for total system functioning.

Obviously it is time-consuming. But remember it is one time process and once you test the firmware in an incremental model you can go for mass production. Incremental firmware burning technique is widely adopted in small, simple system developments and in product development where time is not a big constraint. It is also very useful in product development environments where no other debug tool are available.

2. 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 print a string given as per the firmware. You can insert debug codes 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. You can view the printf generated data on a Terminal Program. Typical usage of inline debug code and the debug information retrieved on terminal is illustrated below

```
prinft(“Starting configuration\n”);

configuration();

printf(“End of configuration\n”);

printf(“Start of initialization\n”);

initialization();

printf(“End of initialization\n”);
```

If the firmware is error free and the execution occurs properly, you will get all the debug messages on the terminal program.

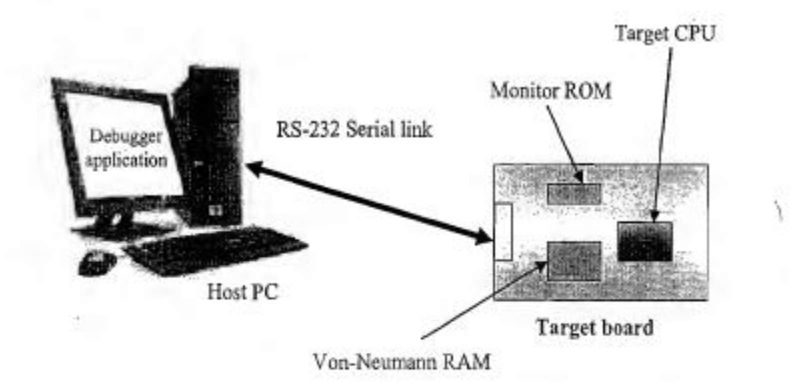
3. Monitor Program based Firmware Debugging

Monitor program based firmware debugging is the first adopted invasive method for firmware debugging. In this approach, a monitor program which acts as a supervisor is developed.

The monitor program contains the following set of 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 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 FLASH memory or ROM of the target board. The code memory containing the monitor program is known as the Monitor ROM.



The monitor program usually resides at the rest vector of the target processor. The actual code memory is downloaded into a RAM chip which is interfaced to the processor in the Von-Neumann architecture model. Monitor ROM size varies in the range of a few kilobytes.

Monitor ROM based debugging is suitable only for development work and it is not good choice for mass produced systems. The major drawbacks of the monitor based debugging systems are

- The entire memory map is converted into Von-Neumann model and it is shared between the monitor ROM, monitor program data memory, monitor program trace buffer, user written firmware and external user memory. For 8051, the original Harvard architecture supports 64K code memory and 64K external data memory. Going for a monitor based debugging shrink the total memory to 64k Von-Neumann memory and it needs to accommodate all kinds of memory requirements.
- The communication link between the debug application running on Development PC and monitor program residing in the target system is achieved through a serial link and usually the controller's on chip UART is used for establish this link. Hence one serial port of the target processor become dedicated to the monitor application and it cannot be used for any other device interfacing.

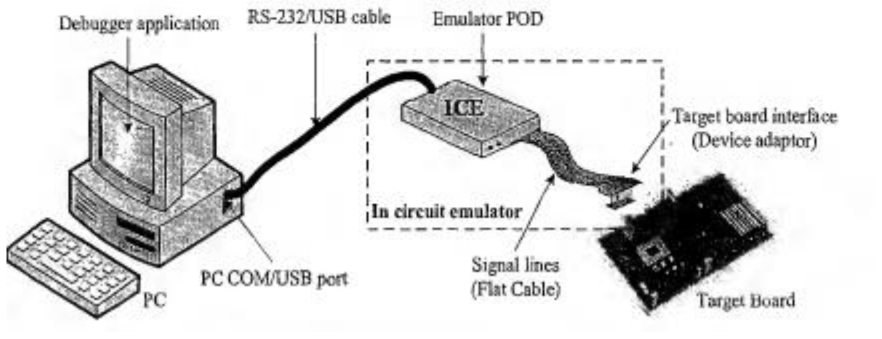
4. In Circuit Emulator (ICE) Based Firmware Debugging

The terms ‘Simulator’ and ‘Emulator’ are little bit confusing and sounds similar. Though their basic functionality is to debug the target firmware, the way in which they achieve this functionality is totally different.

‘Simulator’ is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU, whereas an ‘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. In summary, the simulator ‘simulates’ the target board CPU and the emulator ‘emulates’ the target board CPU.

Nowadays pure software applications which perform the functioning of a hardware emulator is also called as ‘Emulators’. The emulator application for emulating the operation of a PDA phone for application development is an example of a ‘Software Emulator’.

The Emulator POD forms the heart of any emulator system and it contains the following functional units.



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 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** 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 behavior. At the same time the 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. A major drawback of PLD-based emulator is the accuracy of replication of target CPU functionalities. 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 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’. ROM emulation eliminates the hassles of ROM burning and it offers the benefit of infinite number of reprogrammings.

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 trace buffer size is emulator dependent and the trace buffer holds the recent trace information when the buffer overflows.

Emulator Control Logic

Emulator control logic is the logic circuits used for implementing complex 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

Device adaptors act as an interface between the target board and emulator POD. Device 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.

The above-mentioned emulators are almost dedicated ones, meaning they are built for emulating a specific target processor and have little or less support for emulating the derivatives of the target processor for which the emulator is built. 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 (DBMs).

An alternative method of emulator design supports emulation of a variety of target processors. Here the emulator hardware is partitioned into two, namely, 'Base Terminal’ and ‘Probe Card’.

The Base terminal contains all the emulator hardware and emulation control logic except the emulation chip (Target board CPU’s replica). The base terminal is connected to the Development PC for establishing communication with the debug application.

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'.

5. On Chip Firmware Debugging

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.

OCD module implements dedicated registers for controlling debugging. An On Chip Debugger can be enabled by setting the OCD enable bit. Debug related registers are used for debugger control (Enable/disable single stepping, Freeze execution, etc.) and breakpoint address setting.

Background Debug Mode (BDM) and Joint Test Action Group (JTAG) are the two commonly used interfaces to communicate between the Debug application running on Development PC and OCD module of target CPU. The interface between the hardware and PC may be Serial/Parallel/USB.

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. BDM makes use of 10 or 26 pin connector to connect to the target board. **Serial data in** (DSI), **Serial data out** (DSO) and **Serial clock** (DSCLK) are the three major signal lines used in BDM. DSI sends debug commands serially to the target processor from the remote debugger application and DSO sends the debug response to the debugger from the processor. Synchronization of serial transmission is done by the serial clock DSCLK generated by the debugger application. Debugging is controlled by BDM specific debug commands.

Chips with JTAG debug interface contain a built-in JTAG port for communicating with the remote debugger application. JTAG is the alternate nappe for IEEE 1149.1 standard. Like BDM, JTAG is also a serial interface. The signal lines of JTAG protocol are explained below.

- 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.
- The serial data transfer rate for JTAG debugging is chip dependent. It is usually within the range of 10 to 1000 MHz.