Embedded firmware design approaches:

Firmware design depends on the complexity of functions to be performed and speed of operation required. Two basic approaches of Embedded firmware design are:

1. Conventional procedural based firmware design (Super loop model)
2. Embedded operating system (OS) based design.

1. Super loop based model:

* Super loop firmware design approach is adopted for applications that are not time critical and where the response time is not so important.
* These are the embedded systems where the code is executed task by task where mining deadlines are acceptable.
* Super loop based design doesnot require an operating system since there is no need for scheduling and assigning of priority to each task.
* In this design, priority of tasks are fixed and the order of execution is also fixed.

Ex:- Reading/Writing data to and from a card using card reader requires a sequence of operations like checking the presence of card, authentication of the operation, reading/writing etc.

It should follow a sequence and the combination of all these series of tasks constitute a single task.

Advantages:

* Superloop design is simple and straight forward without any OS related overheads.
* No need of special operating system (OS)

Disadvantages:

* Any failure in the part of task will affect the total system.
* Lack of real timeliners brings the probability of missing events.

The firmware execution flow of superloop model will be

① Configure the command parameters and perform initialization of various hardware components, memory, registers etc.
② Start the first task and execute it.
③ Execute second task
④ Execute next task
⑤ ⋮
⑥ ⋮
⑦ Execute the last defined task.
⑧ Jump back to the first task and follow the same flow.

Ex:-

```
Void main()
{
    Configurations();
    Initializations();
    While(1)
    {
        Task1();
        Task2();
        ⋮
    }   Taskn();
}
```

② Embedded Operating System (OS) based approach:

There are two types of operating systems. They are:

(a) General Purpose Operating System (GPOS)

[Ex:- Windows XP, Unix, Linux]

(b) Real time Operating System (RTOS)

[Ex:- Symbian, Elinux, Thread X, Vx Works etc.]

(a) General Purpose Operating System:

This is very similar to conventional personal computer based application development where the device contains an operating system and user applications will run on top of it.

Ex:- Windows, Unix, Linux etc.

Examples of Embedded products which use GPOS are Personal Digital Assistants (PDA's), Hand held devices/ portable devices and point of sales (POS) terminals.

(b) Real time Operating System:

RTOS Contains real time Kernel responsible for performing pre-emptive multitasking, schedules for scheduling task, multiple threads etc.

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

Ex:- Windows CE, PSOS, Thread X, Embedded linux, Symbian Micro C/OS-II etc. Most of the mobile phones are built around the popular RTOS 'Symbian'.

Embedded firmware development languages:

Firmware can be in two ways.

① Target Processor/Controller specific language (low level or) assembly level language) Ex:- ALP

② Target Processor/Controller independent language (High level language) Ex:- C, C++, Java, Python etc.

① Assembly level language programming (low level):

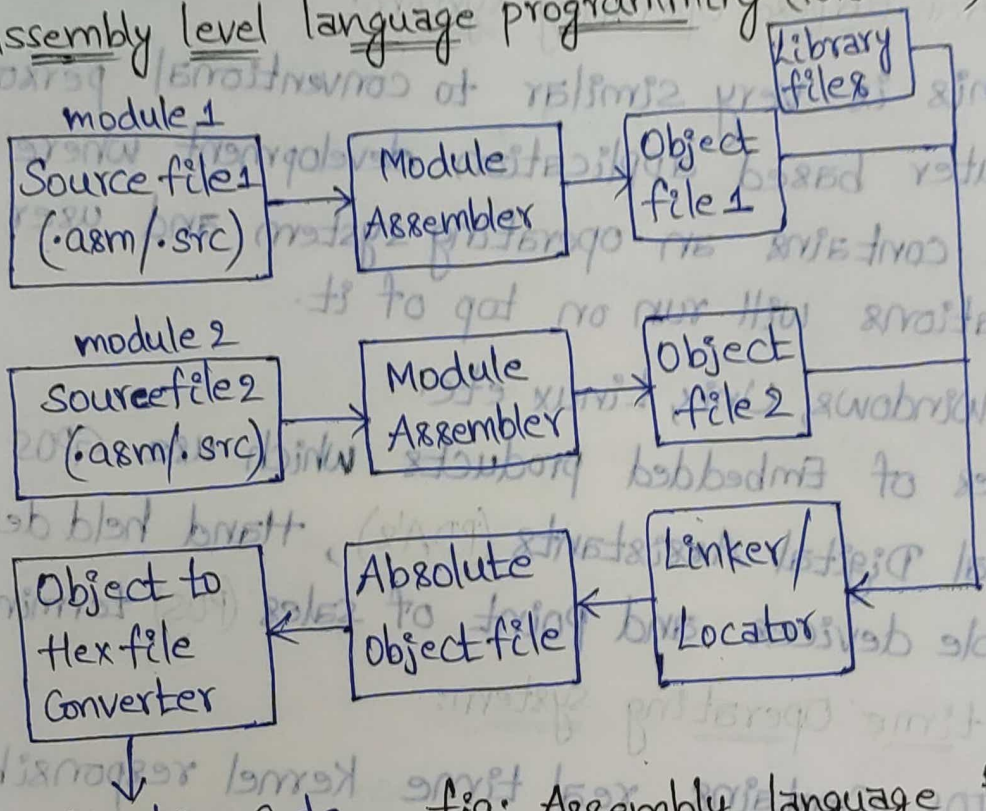


fig: Assembly language to Machine language Conversion Process.

* Assembly language is the human readable notation of "Machine language". They use specific symbols called "Mnemonics".
* A machine level language is processor understandable language. Processor deals with only 1's and 0's.
* Machine language and Assembly language are

processor/controller dependent. A program written for one processor/controller family will not work with others.

* General format for Assembly level language is Opcode followed by operands. Opcode specifies what to do by processor/controller and operands provide the data and information required to perform action specified by opcode.

Some of the opcode is implicitly contains the operand and in such case no operand is required.

Ex:- Consider 8051 ASM instruction `MOV A, #30`
(opcode operand)
This moves the value 30 into Accumulator 'A' (single)
The same instruction written in Machine language is expressed as 01110100 00011110
(opcode operand)

Ex-2: Inc A : This increments accumulator by value '1'.
(opcode, no operand)

Each line of assembly level language is split into 4 fields.

Label	Opcode	Operand	Comments
1	2	3	4

Label: (optional) It represents memory location, address of a program, subroutine, code portion etc. They can contain numbers from 0 to 9 and '_' (under-score). They are always sufixed by Colon and begin with valid character.

Ex:- SUBROUTINE FOR GENERATING DELAY
DELAY PARAMETER PASSED THROUGH REGISTER R1
RETURN VALUE NONE
REGISTERS USED : R0, R1

```
DELAY: MOV R0, #255 ; load R0 with 255
       DJNZ R1, Delay ; Decrement R1 & loop till R1=0
       RET ; Return to calling program
```

Operation of assembly level language programming:

* The Assembly level language is written in Assembly code is saved as .asm (assembly file) (or) .src (source) file. Any text editor like wordpad (or), Notepad can provide an IDE (Integrated Development Environment) tool can be used for writing assembly code.
* Modular programming is employed when programming is too complex. Here the entire code is divided into submodules. Each module is made re-usable.
* Source file to object file translation: Translation of source code to assembler code is performed by assembler.
  Ex:- A51 Macro assembler from Keil Software is a popular assembler for 8051 Micro Controllers.
* On successful conversion of .src/.asm file, a corresponding (.obj) object file is generated. Object file is also called relocatable segment because absolute address of generated code is to be placed in program memory.

* Thus a linker/locator is required to place the code memory in the respective absolute address. Each module can share variables/functions among them. Exporting modules/subroutines is done by declaring them as PUBLIC, EXTRN (Extern)
* Library is the collection of pre-defined object modules with (lib) extension.
  Ex:- LIB51 from KEIL Software for A51 Assemblers/ C51 Compiler for 8051 Controller.

Linker/Locator: It assign absolute address to each module. Here all code and data reside in fixed memory locations. This is used for generation of .Hex file.
Ex:- BL51 from Keil Software is example of linker/locator for A51 Assembler/C51 Compiler for 8051 Specific Controller.

Object to Hex file Conversion: This is final stage of conversion of Assembly level language (Mnemonics) to Machine understandable language (Machine code).
Ex:- For Intel processor the hex file will be "Intel Hex".
For Motorola the hex file is "Motorola Hex".
Hex files are ASIC files that contain Hexadecimal representation of target application.
Ex:- OH51 is "absolute object file" to Hex file Converter for Keil Software.

Advantages:

1. Efficient code and data memory utilization.
2. Provides low level hardware access.
3. Easy for reverse engineering the code memory.

Drawbacks:

1. It requires High development time.
2. Developer dependency makes code complex and non-productive
3. Non-portable because of re-writing is required for different processors.

② High level language based development:

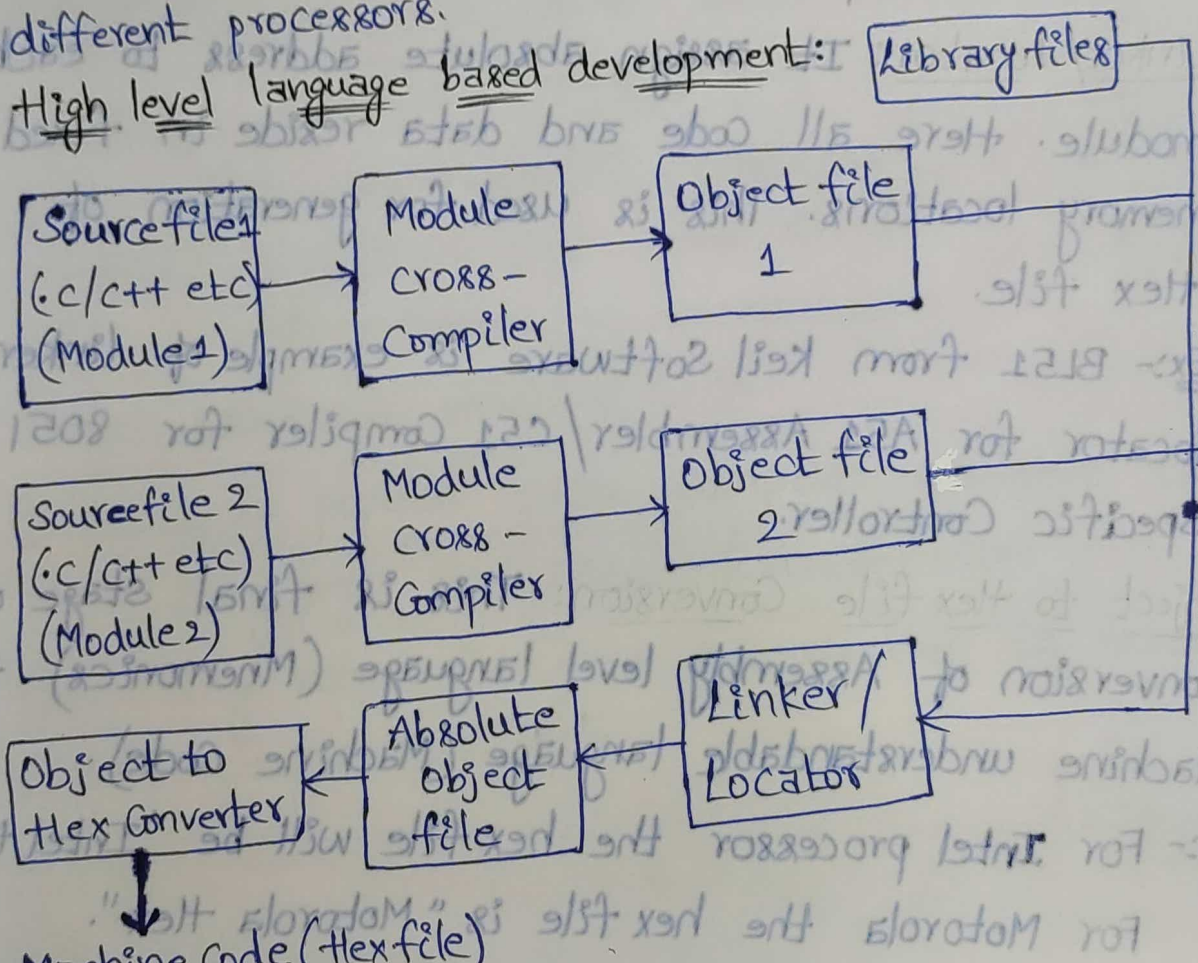


fig: High level language to Machine language Conversion process.

* Any high level language (C, C++, Java, .net, python, oracle, DBMS, R-programming etc.) with a support of cross-compiler (converts application developed in

high level language to target processor specific assembly code for target processor) can be used for embedded firmware development.

* Various steps involved in high level language based embedded systems firmware development is same as that of assembly level language based development except conversion process.
* The program written in any one of the high-level language is saved with corresponding language extension (.c, .cpp etc).
* These files are arranged into modules and given to cross-compiler. Each cross compiler is responsible for converting high level source code into the target processor machine code.

Ex:- C51 is popular cross compiler available for 'C'-language for 8051 Micro Controller.

* Remaining steps involved are similar to Assembly level language conversion.

Advantages:

① Reduced development time due to less no. of lines of code.
② Developer Independency
③ Portability

Compiler Vs Cross-Compiler:

Compiler	Cross- Compiler
① It is a software tool that converts a source code written in high level language on top of a particular operating system running on a specific target processor architecture.	① These are the software tools used in cross-platform development applications. The Compiler running on particular processor converts source code to machine code of target processor whose architecture & instruction set is different.
② They generate machine code for same machine on which it is running.	② They generate machine code for different machine (processor) on which it is running.
③ They are native Compilers.	③ They are non-native Compilers.
④ Ex:- Intel X86, Pentium Processors etc.	④ Ex:- 8051 Controllers, PIC (Programmable Intelligent Computer) Controller, ARM (Advanced RISC Machine) etc.
⑤ It helps to convert the high level source code into machine level code.	⑤ It can create executable code for different machines other than the machine it runs on.
⑥ It is non-retargeting Compiler.	⑥ It is a retargetable Compiler.

C-language Vs Embedded-C language

C-language	Embedded-C language
① 'C' is a well structured, well defined standardized general purpose language.	① It is a subset of C-language which supports all C-language instructions and incorporates few target specific functions/ instructions.
② C-language is used for desktop computers.	② Embedded-C is for micro-controller based applications.
③ C-language programming uses resources like desktop, memory, OS etc.	③ It has to use limited resources like RAM, ROM, I/O's on Embedded processor
④ Extra features are not provided in C-language.	④ It includes extra features over C such as fixed point, multiple memory areas and I/O register mapping.
⑤ Compilers in C-language generates (ANSI C) OS dependent variables.	⑤ Embedded C requires to generate executable files for micro controllers/processors
⑥ It cannot be associated with real time embedded systems.	⑥ It is more convenient for real time embedded systems.

ISR Concept (Interrupt Service Routine):

There are two types of Interrupt sources.

① Software Interrupt sources
- * Error related sources
- * Instruction related sources

② Hardware Interrupt Sources
- * Internal hardware sources
- * External hardware sources

① Software Error related interrupts:

Each processor has specific instruction set. Any illegal code which doesnot corresponds to the set leads to illegal code interrupts. These are software related hardware interrupts.

Ex:- Division by zero detection (or) trap), overflow by hardware, underflow by hardware, illegal opcode by hardware etc.

② Software Instruction related interrupts:

A program can have computational errors (or) run-time conditions which generate Instruction related interrupts.

Ex:- Programmer defined exceptions, signals from device driver functions, square root of negative numbers etc.

③ Hardware Interrupt related internal devices:

These are processor/controller/internal device hardware specific.

Ex:- Timer overflow, ADC to start Conversion, ADC to end Conversion, pulse accumulator overflow, TDRE (Transmit Data Receive Empty) etc.

④ Hardware Interrupt related to external devices:

External hardware interrupts providing vector address (or) Interrupt type through the data bus.

Ex:- NMI, INTO, INT in 8051 etc.

Interrupt Service Mechanism (ISM):

* On an Interrupt, processor vectors to new address ISR - VECT ADDR
* It means the PC (Program Counter) which contain next

address saves it on stack memory (or) in a temporary register called link register. and processor loads ISR-VECTADDR into the PC.

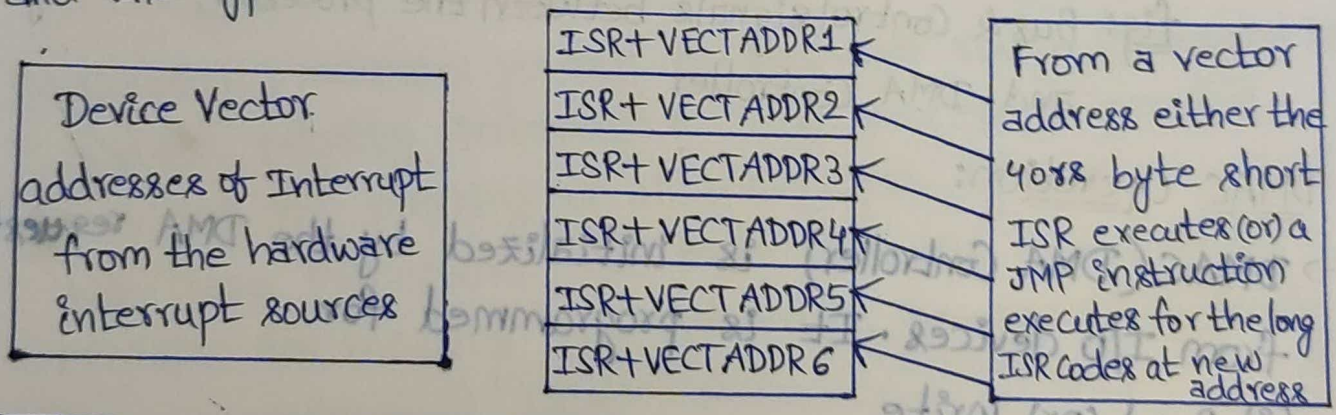
* The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack.
* When PC saves at the link register, it is part of CPU register set. ISR last instruction is RETI.

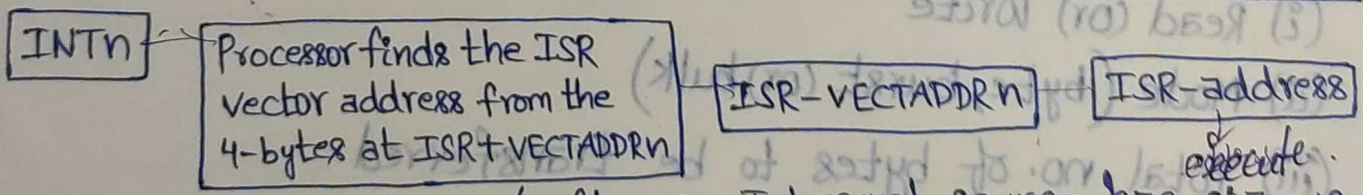
Processor Vector Address:

* A system has internal devices [ext on-chip times, A/D converter etc.] which generate internal interrupts. Each internal device interrupt source (or) source group has a separate ISR-VECTADDR address each.

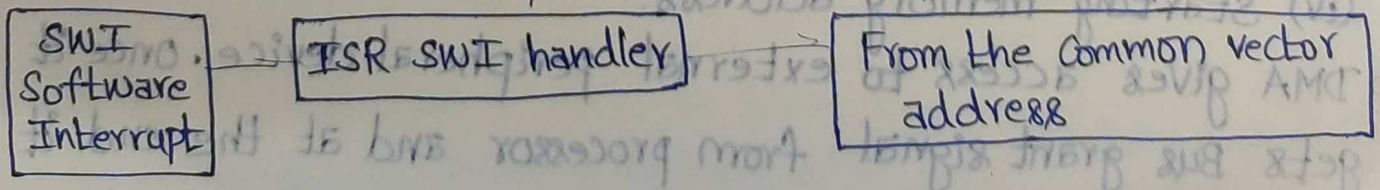
Each external interrupt pin has separate ISR+VECTADDR.

In 8086 Microprocessor architecture, a software instruction for example INTn explicitly also defines the type of interrupt and the type defines the ISR-VECTADDR.





Each hardware/software Interrupt source has size n.



Direct Memory Access (DMA):

The transfer of data between fast storage device such as Magnetic disk and memory is often limited by the speed of CPU. Removing the CPU from the path and letting the peripheral devices managing the memory bus directly by using DMA Controller.

→ DMA Controller interface provides I/O transfer of data directly to and from the memory and I/O device.

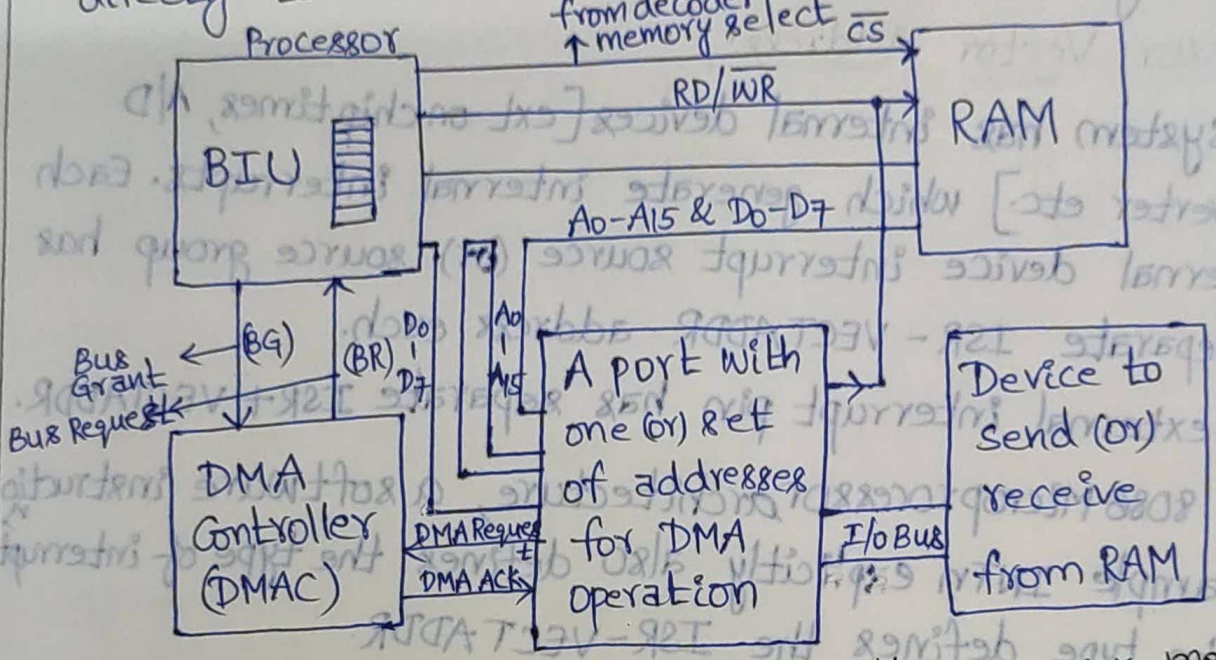


fig: Bus & Control signals between the processor, memory and DMA Controller

DMAC Operation:

→ DMAC (DMA Controller) is initialized by the DMA request from I/O devices. It is programmed for

(i) Read (or) write

(ii) Mode (byte, burst (or) bulk)

(iii) Total no. of bytes to be transferred

(iv) Starting memory address.

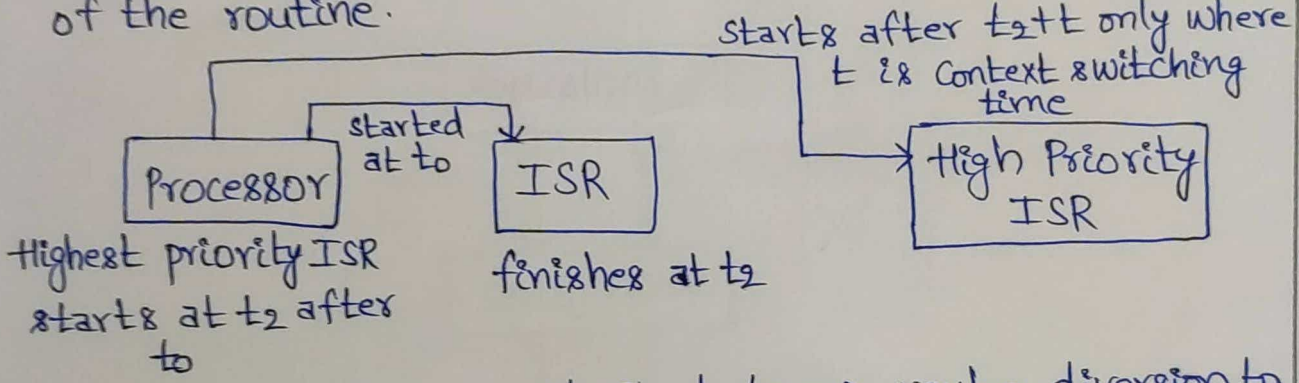
→ DMA gives access to external peripheral device. Once it gets Bus grant signal from processor and at the end it

communicates to processor that the task is completed.

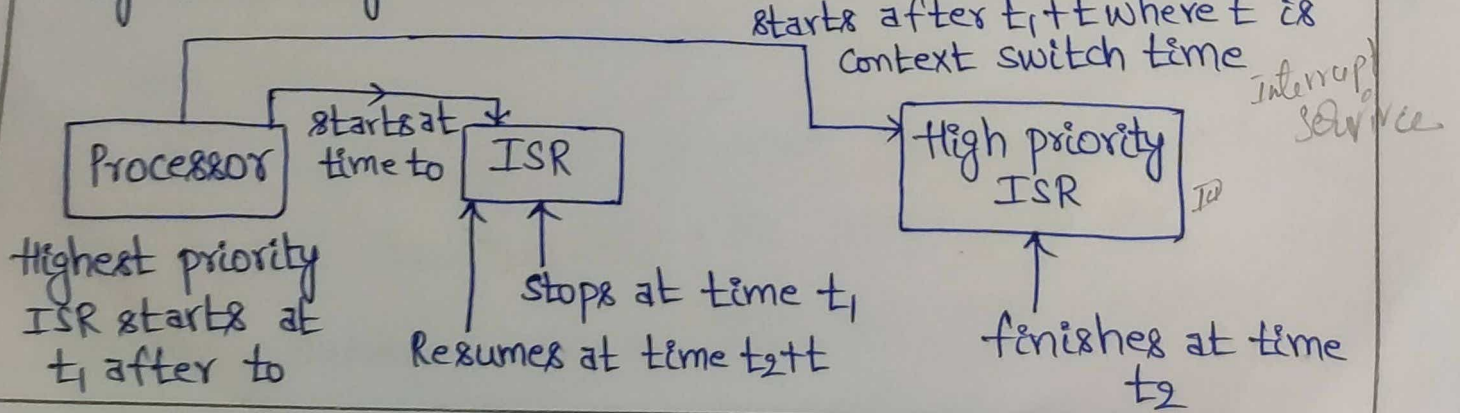
Multiple Interrupts:

When there are multiple interrupt sources, each interrupt is identified by the status register. When multiple interrupts are given then processor services the highest priority interrupt first after it's completion then it returns to lowest priority pending ISR (Interrupt Service Routine).

1. Certain processors do not provide for in-between routine diversion to higher priority interrupts and presume that all interrupts of priority greater than presently running routine are marked till the end of the routine.



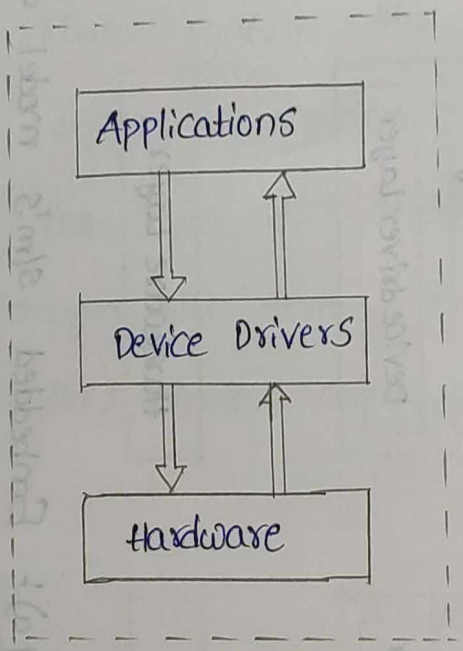
2. Certain processors permit in-between route diversion to highest priority interrupts. These processors provide provision for masking of all interrupts by a primary level bit. These processors also have selective diversion by a provision for masking the interrupt service selectively by secondary-level bits.



Device Drivers:

Most embedded hardware requires some type of Software initialization and management. The software that directly interfaces with and controls this hardware is called a 'Device drivers'.

∴ Device drivers are the software Libraries that initialize the hardware. It is proved between the application and Hardware.



* A driver is a software component that lets the operating system and a device communicate with each other.

* Almost every system operation eventually maps to a physical device, with the exception of the Processor, memory and a very few other entities,

any and all device control operations are performed by code that is specific to the device being addressed that code is called a device driver.

Fig a): Embedded s/m's model and Device drivers

* Device Drivers are typically considered either 'architecture-specific' (or) 'generic'.

* Architecture-specific drivers that initialize and enable components with in a master processor include on-chip memory, integrated memory managers and floating Point hardware.

* Generic drivers manages hardware that is located on the board and not integrated on to the master processor.

* Generic driver can be configured to run on a variety of architectures that contain the related board hardware for which the driver is written.

* Regardless of the type of device driver (or) the hardware it manages, all device drivers are generally made up of all (or) some combination of the following functions.

* Hardware Setup
* Hardware Shutdown
* Hardware disable
* Hardware Enable
* Hardware Acquire
* Hardware Release
* Hardware Read
* Hardware write
* Hardware Install
* Hardware Uninstall

3. On-Board Bus device drivers:

* Bus start up → Initialization of the bus upon power-on (or) reset
* Bus shutdown → bus into its power off state
* Bus disable → Allowing other s/w to disable bus on the fly
* Bus Enable → " " " " enable " " " "
* Bus Acquire → locking to bus
* Bus release → unlock
* Bus read → Read data on bus
* Bus write → write data on bus
* Bus Install → Install new bus

4. Board Input-output Drivers:

* I/o startup
* I/o shutdown
* I/o Enable
* I/o Disable
* I/o Acquire
* I/o Release
* I/o Read
* I/o write
* I/o Install
* I/o uninstall