UNIT-3

EMBEDDED FIRMWARE DESIGN

1. Embedded Firmware design approaches :-

The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of operations required, etc.

* Two basic approaches are used for Embedded firmware design.

  (i) Conventional Procedural Based Firmware Design (or) Super Loop Based Approach.

  (ii) Embedded operating System (OS) Based Designs.

(i) The Super Loop Based Approach :-

* The super loop based firmware development approach is adopted for applications that are not time critical and where the response time is not so important.
* In this the code is executed task by task, The task listed at the top of the program code is executed first and the tasks just just below the top are executed after completing the first task.
* The tasks are running inside an infinite loop, the only way to come out of the loop is either a hardware reset or an interrupt assertion.
* Hardware reset brings the program execution back to the main loop.
* Interrupt request suspends the tasks execution temporarily, and it resumes the task execution from the point where it got interrupted.
* It doesn't require an operating system,
* In this design, the priorities are fixed and the order in which the tasks to be

executed are also fixed.

* This type of design is deployed in low-cost embedded products and products where response time is not time critical.
* For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authenticating the operation, reading/writing, etc..
* A typical example for a super loop based product is an electronic video game - toy containing keypad and display unit.

Drawbacks:-

* Any failure in any part of a single task will affect the total system.
* If the program hangs up at some while executing a task, it will require H/w and S/w watchdog timers (WDTs), may cause additional h/w cost and firmware overheads.

(ii) The Embedded Operating Systems (OS) Based Approach :-

* The operating systems (OS) based approach contains operating systems, which can be either a General Purpose operating System (GPOS) or a Real Time operating systems (RTOS).
* The GPOS based design is very similar to a conventional PC based application development where the device contains an operating system (Windows/Linux/unix)
  Ex:- personal Digital Assistants (PDA's), Handheld/Portable devices & point of sale
* RTOS based design approach is employed in embedded products demanding real time response.
* It responsible for performing pre-emptive multitasking, Scheduler for scheduling tasks, multiple threads etc.
  Ex:- windows CE, pSOS, VxWorks, Threadx, Micro C/OS-II, Embedded Linux, Symbian etc..

2. Embedded Firmware Development Languages :-

(i) Assembly language based development.

(ii) High Level language based development.

(iii) Mixing Assembly and High Level language.

(i) Assembly language based development :-

* 'Assembly language' is the human readable notation of 'machine language', whereas 'machine language' is a processor understandable language.
* Assembly language programming is the task of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler
* Each line of an assembly language program is split into four fields as given

LABEL OPCODE OPERAND COMMENTS.

* A 'LABEL' is an optional field, is commonly used for representing A memory location, address of a program, sub-routine, code portion etc.
* 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.
* The assembly language program written in assembly code is saved as .asm (Assembly file) file or an .src (source) file.
* The following figure shows the assembly language to machine language conversion process.
* Translation of assembly code to machine code is performed by assembler.

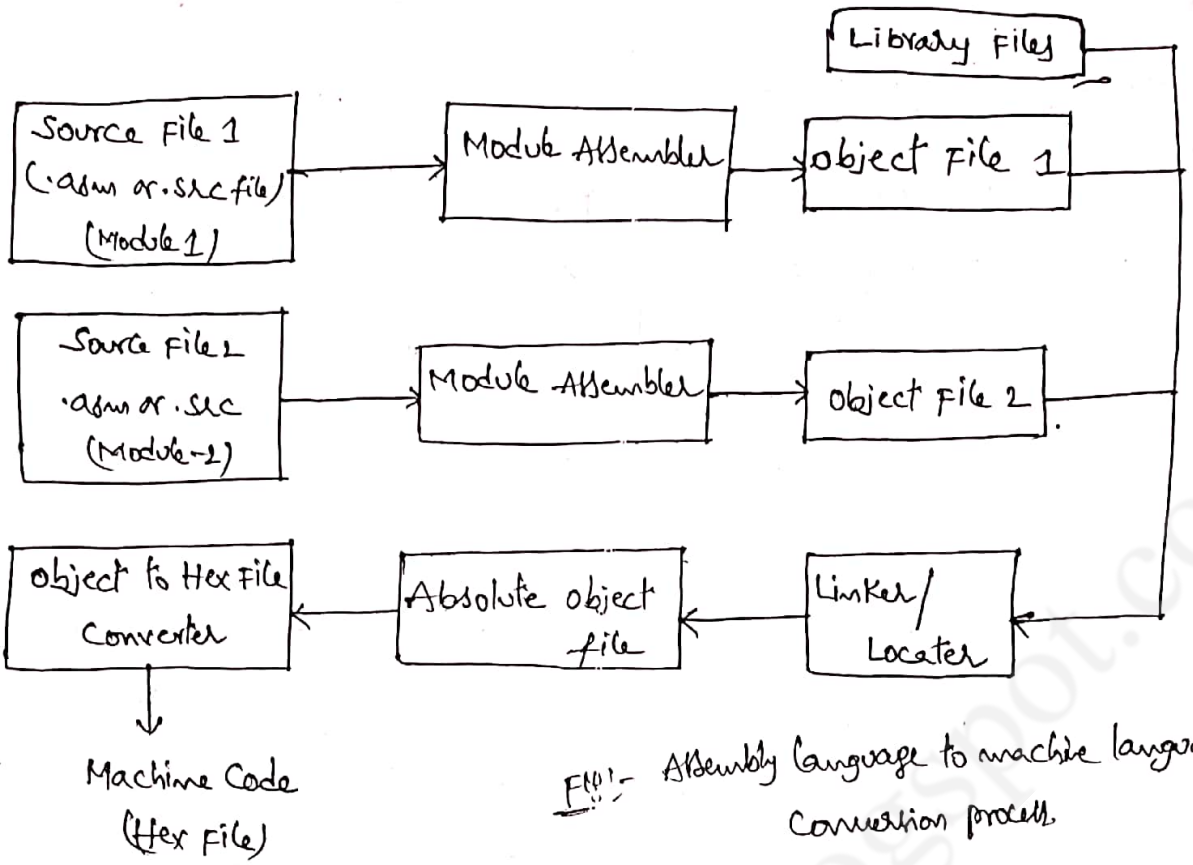


Fig:- Assembly language to machine language conversion process.

Library file creation and usage :- Libraries are specially formatted, ordered programs collections of object modules that may be used by the linker at a later time.

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

object to Hex file Converter :- This is the final stage in the conversion of Assembly language (mnemonics) to machine understandable language.

Advantages :-

* Efficient Code Memory and Data Memory Usage (Memory optimisation) :- Since the developer is well versed with the target processor architecture and memory organisation, optimized Code can be written for performing operations.
* High performance :- Optimised Code not only improves the code memory usage but also improves the total system performance.

* Low Level Hardware Access :- Most of the code for low level programming like accessing external device specific registers from the operating system kern device drivers, and low level interrupt routines etc.

* Code Reverse Engineering :- Reverse engineering is the process of understan the technology behind a product by extracting the information from a finished product.

Drawbacks :-

* High Development Time :- Assembly language is much harder to program than high level languages. The developer must pay attention to more details and must have through knowledge of the architecture, memory organization and register details of the target processor in use.

* Developer Dependency :- There is no common written rule for developing assembly language based applications whereas all high level languages instruct certain set of rules for application development.

* Non-portable :- The applications written in assembly instructions are valid only for that particular family of processors.

(ii) High Level Language Based Development :-

* High level language (C, C++ or Java) with a supported Cross Compiler for the target processor can be used for embedded firmware development.

* The various steps involved in high level language based embedded firmware development is shown in figure.

* The cross-compilers for different high level languages for the same target processor are different.

* It should be noted that each high-level language should have a cross-compiler for converting the high level source code into the target processor machine code

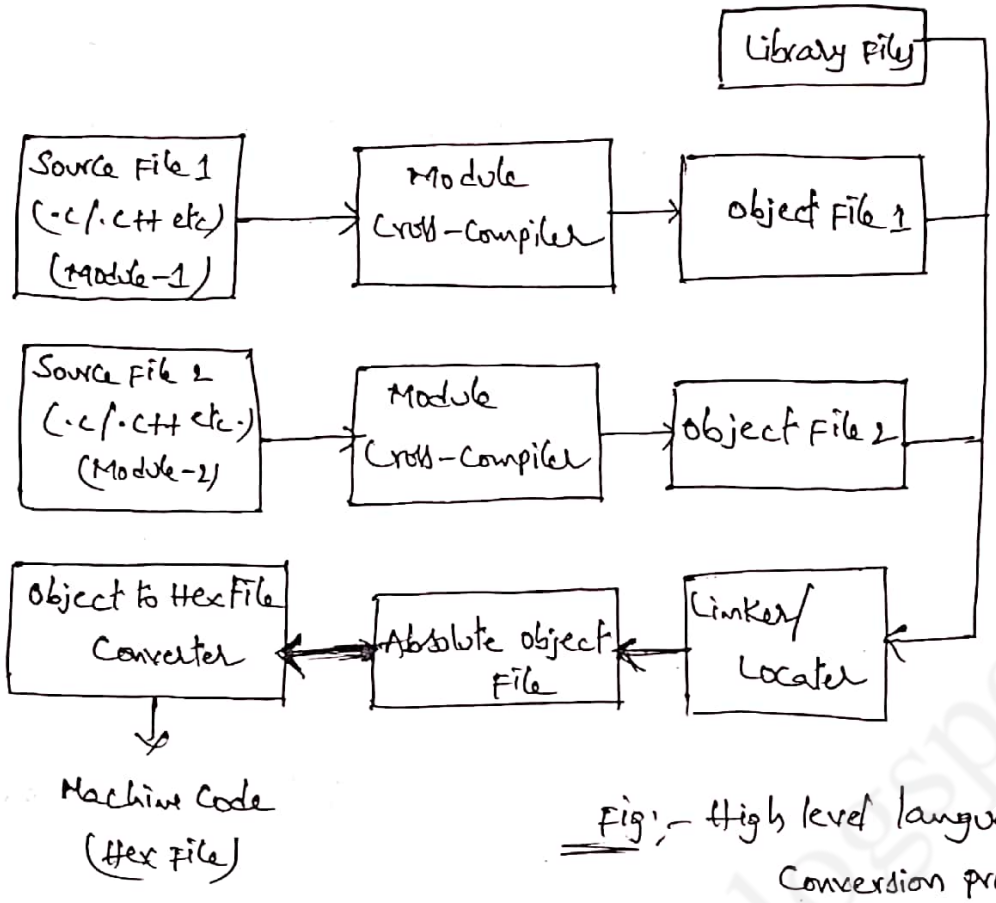


Fig:- High level language to machine language Conversion process.

* Advantages :-

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

(ii) Developer Independency :- The syntax used by most of the high level languages are universal and a programs written in the high level language can easily be understood by a second person knowing the syntax of the language

(iii) portability :- An application written in high level language for a particular target processor can easily be converted to another target processor/Control application

* Limitation :- Some Cross-Compilers available for high level languages may not be so efficient in generating optimized target processor specific instructions.

(iii) Mixing Assembly and High level Language :-

* Certain embedded firmware development situations may demand the mixing of high level language with Assembly and vice versa.
* High level language and assembly languages are usually mixed in three ways; namely
  a) Mixing Assembly language with high level language
  b) Mixing High level language with Assembly
  c) In-line Assembly programming.

a) Mixing Assembly with High level language :-

(Eg: Assembly language with 'C')

* Mixing 'C' and assembly is little complicated in the sense - the programmer must be aware of how parameters are passed from the 'C' routine to Assembly and values are returned from assembly routine to 'C' and how 'Assembly - routine' is invoked from the 'C' code.

b) Mixing High level language with Assembly :-

(Eg: 'C' with Assembly Language)

Mixing the code written in a high level language like 'C' and Assembly language is useful in the following scenarios:

(i) The source code is already available in Assembly language and a routine written in a high level language like 'C' needs to be included to the existing code.

(ii) The entire source code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory etc.

(iii) To include built in library functions written in 'C' language provided by the cross compiler.

(c) Inline Assembly :-

* Inline assembly is another technique for inserting target processor/controller specific Assembly instructions at any location of a source code written in high level language 'C'.
* This avoids the delay in calling an assembly routine from a 'C' code
* Special keywords are used to indicate that the start and end of Assembly instructions.

ISR CONCEPT :-

* Interrupt means event, which invites the attention of processor for some action on the hardware or software event.

1. When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. This interrupt is called hardware interrupt.
2. When software run-time exception condition is detected, either processor h/w or a s/w instruction generates an interrupt. This interrupt is called software interrupt or trap or exception.

* In response to the interrupt, the routine or program, which is running at present gets interrupted and an ISR is executed.

Features of ISR :-

1. An ISR call due to interrupt executes an event. Event can occur at any moment and event occurrences are asynchronous.
2. ISR call is event-based diversion from the current sequence of instructions to the another sequence of instructions. This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or s/w computational exceptional condition detected by h/w or detected by program, which throws the exception.
4. An event can be signalled by s/w interrupt instruction SWI used in device driving functions create(), open(), etc.
5. An interrupt service mechanism exists in a system to call the ISRs from multiple sources.
6. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine.

INTERRUPT SOURCES :-

* Hardware sources can be from internal devices or external peripherals, which interrupt the ongoing routine and thereby cause diversion to corresponding ISR.
* Software sources for interrupt are related to
  (i) processor detecting computational error for an illegal op-code during execution or
  (ii) execution of an SWI instruction to cause processor-interrupt of ongoing routine.
* There may be some other special types of sources provided in the system as

1. Hardware Interrupts Related to Internal Devices :-

There are No of h/w interrupt sources which can interrupt an ongoing programs. These are processor or microcontroller or internal device h/w specific

Ex:- parallel port, UART Serial receiver port 3. Synchronous receiver byte Completion, etc ADC start of conversion, ADC end.

2. Hardware Interrupts Related to External Devices-1 :-

There can be external hardware interrupt source for interrupting an ongoing program that also provides the ISR address or vector address or interrupt-type information through the data bus.

Ex:- INTR in 8086 and 80x86.

3. Hardware Interrupts Related to External Devices-2 :-

External hardware-interrupts with their ISR vector address are processor or microcontroller-specific interrupts of an ongoing program. External interrupting source does not send interrupt-type or ISR address-related information.

Ex:- Non-maskable pins, within first few clock cycles unmaskable declarable pin but otherwise maskable, Maskable pins [In 8086 INTO & INTR]

4. Software Error-Related Hardware interrupts :- There can be the software-error related interrupts generated by processor hardware. Each processor has a specific instruction set. It is designed for that set only.

* An illegal code (instruction in the s/w) is an instruction, which does not correspond to any instruction in this set. whenever the processor fetches illegal code, an interrupt occurs in certain processors. The error-related interrupts are also called hardware-generated software traps (or software exceptions).

Ex:- Division by zero detection (or trap) by hardware, Over-flow by h/w, under flow by h/w, Illegal opcode by h/w.

5. Software Instruction-Related Interrupts Sources :- A program can also handle specific computational errors or run-time conditions or signaling some conditions. Processors provide for software instructions related to the traps, signals or exceptions.

(i) There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling to another routine from an ongoing routine or task or thread.

(ii) Software instructions are also used for trapping some run-time error conditions and executing exceptional handlers on catching the exceptions.

INTERRUPT SERVICING (HANDLING) MECHANISM :-

* Each system has an interrupt servicing (handling) mechanism.
* The OS also provides for mechanisms for interrupt-handling.

(i) Interrupt vector :-

* Interrupt vector is a memory address to which the processor vectors.
* The processor transfers the programs counter to the interrupt vector new address on an interrupt.
* Using this address, the processor services that interrupt by executing corresponding ISR.

* Vectoring is as per the provisions in interrupt-handling mechanism.

The various mechanisms are as follows:

Processor Vectoring to the ISR_VECTADDR :-

* on an interrupt, a processor vectors to a new address, ISR_VECTADDR.
* It means that the PC (Program counter), which has the instruction address of next instruction, saves that address on stack or in some CPU register, called link register and the processor loads the ISR_VECTADDR in to the PC.
* The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack.

A processor provides for one of the following ways of using the ISR_VECTADDR based addressing mechanism.

Processor Vector Address

1. A systems has internal devices like the on-chip timer and A/D Converter. In a given microcontroller, each internal device interrupt source or source-group has a separate ISR_VECTADDR address. Each external interrupt pin has separate ISR_VECTADDR.
2. In 80X86 processor architecture, a software instruction, for example, INT n explicitly also defines the type of interrupt and the type defines the ISR_VECTADDR. This mechanism results in the handling of n number of exception handling routin or ISRs for n interrupt types.
3. In ARM processor architecture, the software instruction SWI does not explicitly define the type of interrupt for generating different vector addrieg and instead there is a common ISR_VECTADDR for each exception or signal or trap generated using SWI instruction.

A group of Interrupt Sources having Common Vector Address :-

A source group in the hardware may have the same ISR_VECTADDR.

There are two types of handling mechanisms in processor hardware. The processor-handling mechanism provides for fetching into the PC either

(i) the ISR instruction at the ISR_VECTADDR or

(ii) the ISR address from the bytes at the ISR_VECTADDR.

Interrupt Vector Table :- System software designer must provide for specifying the bytes at each ISR_VECTADDR address.

* The bytes are for either ISR short code or jump instruction to the ISR first instruct or ISR short code with Call to the full code of the ISR or for fetching the bytes for finding the ISR address.

⇒ Classification of Interrupts :-

There are three types of interrupt sources in a system

1. Non-maskable : Examples are RAM parity error in a PC and error interrupts like division by ZERO. These must be serviced.
2. Maskable : maskable interrupts are those for which service may be temporarily disable to let high priority ISRs be executed first uninterrupted.
3. Non-maskable only when defined so within few clock cycles after reset: for example, an external interrupt pin, XIRQ interrupt, in 68HC11.

XIRQ interrupt is non-maskable only when defined so within few clock cycles after 68HC11 is reset.

MULTIPLE INTERRUPTS :-

Multiple Interrupt Calls :- when there are multiple interrupt sources, each occurance of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (interrupt-pending register)

* There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priority ISR is to lower priority Pending ISR.

Let us understand two procesor interrupt service mechanisms for the case of multiple interrupts.

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

2. Certain processors permit in-between routine diversion to higher priority interrupts. These processors provide, in order to prevent diversion in-between, a mechanism as follows: There is provisioning for masking of all interrupts by a primary level bit. These processors also provision selective diversion by provisioning for masking the interrupt service selectively by secondary-level bit.

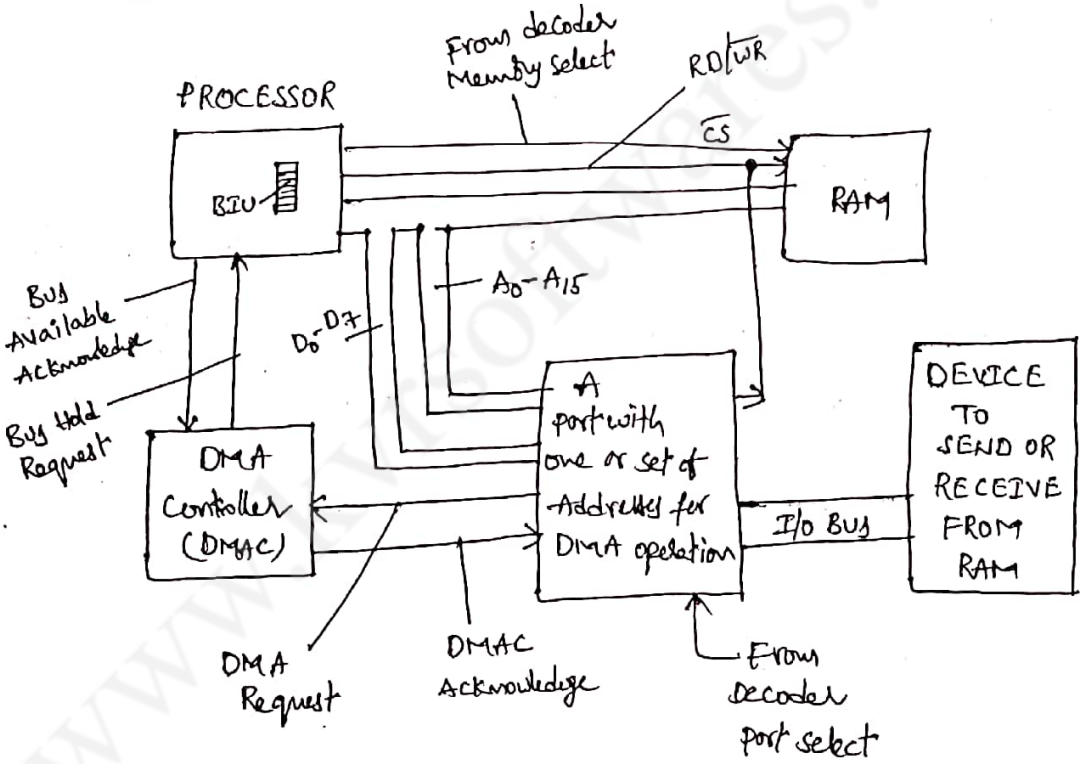
Hardware Assigned priorities :- There is assigned priority order by hardware

* ARM7 provides for two types of external interrupt sources (Requests), IRQs and FIQs (Fast interrupts)
* 8051 provides for priority order in order of interrupt vector address. Lower address has highest and higher has the lower priority.
* Interrupts in 80x86 are assigned priority order according to interrupt-types. Interrupt of type '0' has highest priority and '225' has lowest assigned priority.

—x—

Direct Memory Access :-

* The DMA is used to transfer data between hard disk system memory.
* When the I/O's are needed for large amount data from a peripheral device to the memory address in the system or large amount of data is to be transferred by the I/O's, the interrupt-based mechanism is not suitable.
* A DMA facilitates a multi-byte data set or a burst of data or a block of data transfer between the external device and system or between two systems
* The device DMAC (DMA controller) data transfer occurs efficiently b/w the I/o devices and system memory.
* The following fig. shows the interconnections using the DMAC. It also shows the buses and control signals b/w the processor, memory, DMAC and data-transfer -ring I/o device.



System buses not accessible by processor internal address and data buses during acknowledge active.

Fig:- The buses and control signals between the processor, memory, DMA controller and data-transferring I/o device.

* The DMAC sends a hold request to the CPU and the CPU acknowledges that if the system memory buses are free to use.

* Three modes are usually supported in DMA operations.

(i) Single transfer at a time and then release IO bus hold on the system bus after each transfer.

(ii) Burst transfer at a time and then release of the IO bus hold on the system bus. A burst may be of a few kilobytes.

(iii) Bulk transfer and then release of the IO bus hold on the system bus only after the transfer is completed.

Use of DMAC:-

Whenever a DMA request is made to the DMAC for the I/O's, the DMAC is first initialized. It is programmed for

(i) Read or write
(ii) mode (bytes, burst or bulk) of DMA transfer
(iii) Total No of bytes to be transferred and
(iv) starting memory address.

* whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfer by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC

* A DMAC may also provide memory access to multiple channels.

Device Driver programming :-

* A System has No of physical devices. A device may have multiple functions Each device function requires a driver.

* Examples of multiple functions in a device are as follows.

(i) A timer device performs timing functions as well as counting functions. It also performs the delay function and periodic system calls.

(ii) A transceiver device transmits as well as receives.

(iii) voice-data-fax modems device has transmitting as well as receiving function for voice, fax as well as data.

* The driver has following features

(i) The driver provides a software layer (Interface) between the application and actual device: when running an application, the devices are used. A driver provides a routine that facilitates the use of a device function in the application.

Ex:- For mailing system the ~~device~~ layer network interface card is used

(ii) The driver facilitates the use of a device by executing an ISR; Simple commands from a task or function can then drive the device. Once a driver function is available for writing the codes, the application developer does not need to know anything about the mechanism, addresses, registers, bits and flags used by the device.

Ex:- the system clock is to be set to tick every 10,000 μs.

* Generic device driver functions in high level language are used in high level language programs. The functions are open, close, read, write, listen, accept etc.

* Device driver ISR programming in assembly needs an understanding of the processor, system and I/o buses and the addresses of the device register in the specific hardware.

→ Writing physical Device - Driving ISRs in a System

→ Virtual Device Drivers

→ parallel port Drivers in a System

→ Serial port drivers in a System

→ Device Drivers for Internal programmable Timing Devices

→ Linux Internals of Device Drivers and Network functions.

Programming in Embedded C :-

* Whenever the conventional 'C' language and its extensions are used for programming embedded systems, it is referred as 'Embedded C' progra
* Desktop application development using 'C' language for a particular OS platforms.
* Desktop computers contain working memory in the range of Megabytes and storage memory in the range of Gigabytes.
* Embedded systems are limited in both storage and working memory resources.

'C' V/s 'Embedded C' :-

* 'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support.
* 'C' offers a combination of the features of high level language and assembly and helps in hardware accell programming as well as business package developments.
* The conventional 'C' language follows ANSI standard and it incorporates various library files for different operating systems.
* A platforms (Operating system) specific application, known as Compiler is used for the conversion of programs written in 'C' to the target proces specific binary files.
* Hence it is platforms specific development.

* Embedded 'c' can be considered as a subset of Conventional 'c' lang
* Embedded 'c' supports all 'c' instructions and incorporates a few target processor specific functions/ instructions.
* The implementation of target processor/controller specific functions/instructi depends up on the processor/ controller as well as the supported cross-compiler for the particular Embedded 'c' language
* A Software program called 'Cross-Compiler' is used for the conversion of programs written in Embedded 'c' to target processor/controller specific instructions (machine language)
* The standard ANSI 'c' library implementation is always tailored to the target processor/ controller library files in Embedded 'c'.

Compiler Vs Cross-Compiler :-

* Compiler is a sfw tool that converts a source code written in a high-level language on top of a particular operating system running on a specific target processor architecture
* The development is platforms specific.
* Compilers are generally termed as 'Native compilers'.
* Native compiler generates machine code for the same machine (processor) on which it is running.
* Cross-Compilers are the Software tools used in Cross-platform development applications.

* In Cross-platform development, the compiler running on a particular target processor/OS converts the source code to which machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an OS which is different from the current development environment OS.

- Example is Embedded system development with Intel/AMD.
- Keil 51 is an example of Cross-Compiler

UNIT - III

EMBEDDED FIRMWARE DESIGN

1. Embedded Firmware design approaches

The firmware design approaches for Embedded product is purely dependent on the Complexity of the functions to be performed, the speed of operation required, Etc.,

* Two basic approaches are Used for Embedded firmWare design.
  (i) Conventional procedural Based firmWare Design (or) Super loop Based Approach.
  (ii) Embedded Operating System (OS) Based Designs.

(i) The Super loop Based Approach

* The Super loop based firmWare development approach is adopted for applications that are not time critical and Where the response time is not so important.
* In this the code is executed tasks by task, the task listed at the top of the program code is executed first and the tasks just below the top are execute after completing the first task.
* the tasks are running inside an infinite loop, the Only Way to come Out of the loop is either a hardWare reset or an interrupt assertion.
* hardWare reset brings the program execution back to the main loop.
* Interrupt request suspends the task execution temporarily and it requests the task execution from the point Where it got inte-rrupted.
* It doesn't require an Operating System.
* In this design, the priorities are fixed and the Order in Which the tasks to be executed are also fixed.

* This type of design is deployed in low-cost embedded products and products where response time is not time critical.
* For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authanticating the operation, reading/writing, etc..
* A typical example for a super loop based product is an electronic video game-toy containing keypad and display unit.

Drawbacks

* Any failure in any part of a single tasks will affect the total system.
* If the program hangs up at some while executing a tasks, it will require H/W and S/W watchdog timely (WDTs), may cause additional H/W cost and firmware overheads.

(ii) The Embedded Operating System (OS) Based Approach :-

* The operating system (OS) based approach contains operating systems, which can be either a General/purpose operating system (GPOS) or a Real Time operating system. (RTOS).
* The GPOS based system design is very similar to a conventional PC based applications development where the device contains an Operating System (Windows/linux).
  Ex:- Personal Digital Assistants (PDA's), Hardware held/protalable devices & point of sales.
* RTOS based designs approach is employed in embedded products demanding real time response.
* It responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads etc.
  Ex:- Windows CE, PSOS, Vx Works, Threadx, Microc/os-II, Embedded linux, symbian etc.

Q. Embedded Firmware Development Languages

(i) Assembly language based Development
(ii) High level language based Development
(iii) Mixing Assembly and High level language

(i) Assembly language based Development

* 'Assembly language' is the human readable notation of 'machine language', Where as 'machine language' is a processor understandable language.
* Assembly language programming is the task of Writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler.
* Each line of an assembly language programs is split into four fields as given as

LABEL OPCODE OPERAND COMMENTS

* A 'LABEL' is an optional field, is commonly used for representing A memory location, address of a program, sub-routine, code portion etc.
* 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.
* The assembly language program Written in assembly code is Saved as .asm (Assembly file) file or an .src (source) file.
* The following figure shows the assembly language to machine language Conversion process.
* Translation of assembly code to machine code is performed by assembler.

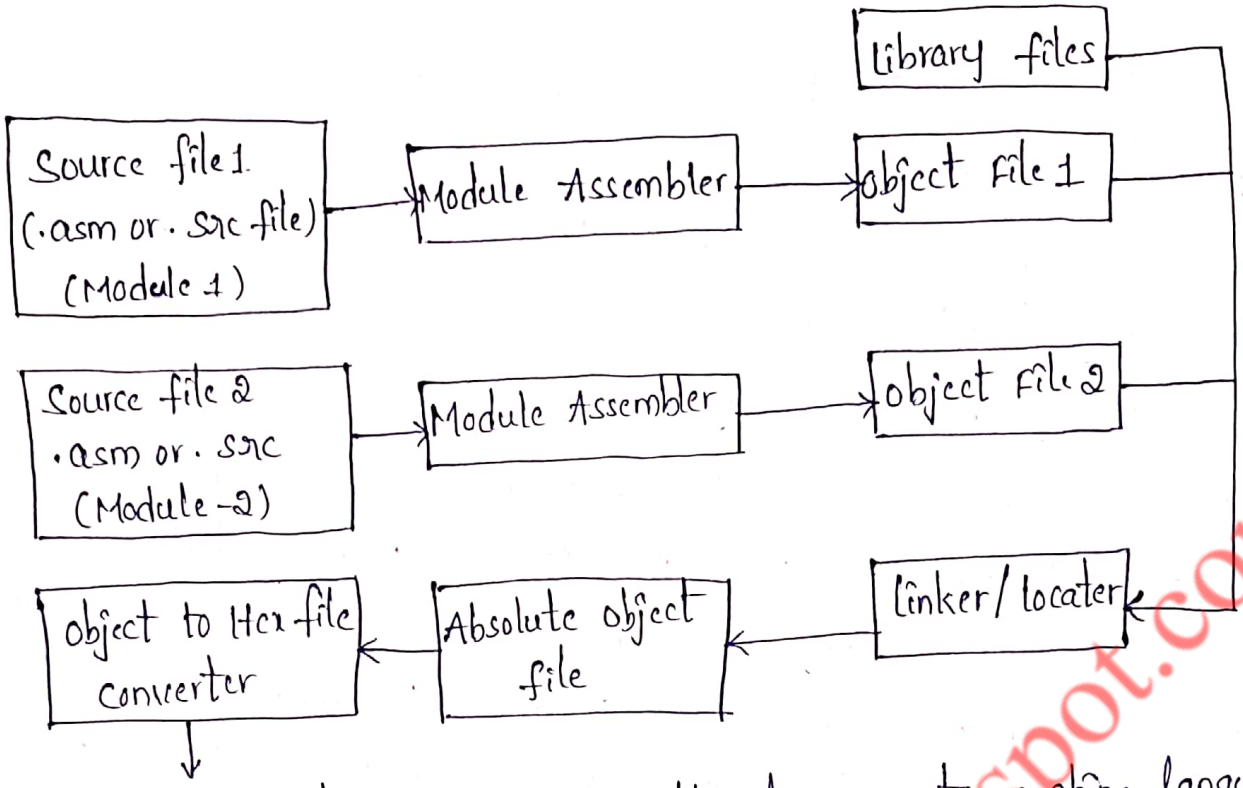


fig:- Assembly language to machine language convertion process

Library file creation using and Usage :- Libraries are specially formatted, Ordered programs collection of object modules that may be used by the linker at a later time.

Linker and locater :- Linker and locator is another software utility respo -nsible for "linking the various object modules in a multi-module project and assigning absolute address to each module".

Object to Hex file converter :- This is the final stage in the Conversion of Assembly language (mnemonics) to machine understandable language.

Advantages

* Efficient code Memory and data Memory Usage (Memory Optimisation):- Since the developer is well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations.
* High performance :- Optimised code not only improves the code Memory usage but also improves the total system performance.
* 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.

* Code Reverse Engineering :- Reverse Engineering is the process of understa... the technology behind a product by extracting the information from a finished product.

Drawbacks :-

* High Development Time :- Assembly language is much harder to programs than high level languages. The developer must pay attention to more details and must have through knowledge of the architecture, Memory Organisation and register details of the target processor in use.
* Developer Dependency :- There is no common written rule for developing assembly language based applications where as all high level languages instruct certain set of rules for application development.
* Non - Portable :- The applications written as in assembly instructions are valid for that particular family of processor.

(ii) High level language Based Development

* High level language (C, C++ or Java) with a supported Cross Compiler for the target processor can be used for Embedded firmware deve -lopment.
* The various steps involved in high level language based Embedded firm ware development is shown in figure.
* The Cross - compilers for different high level languages for the same target processor are different.
* It should be noted that each high - level language should have a Cross - compiler for converting the high - level source into the target processor machine code.

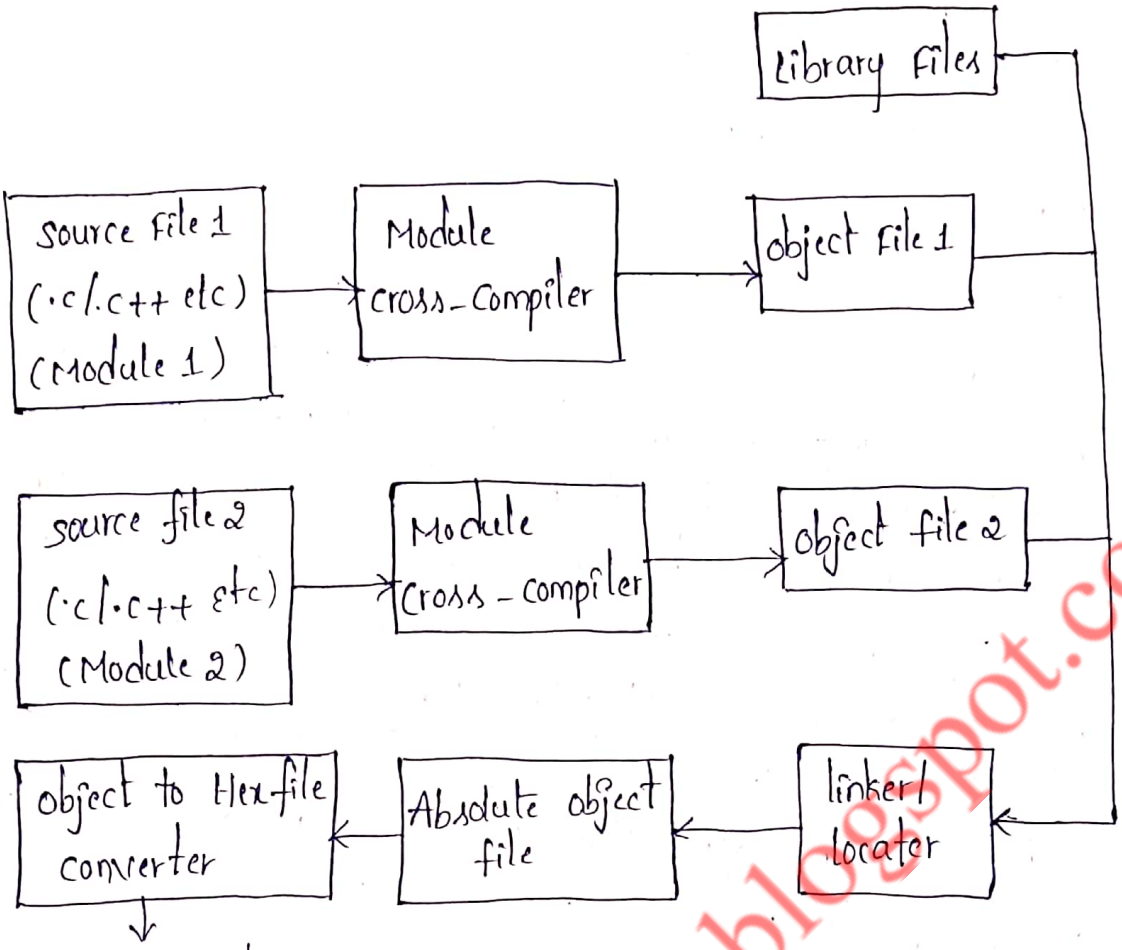


fig :- High level language to machine language Conversion process.

Advantages

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

(ii) Developer Independency :- The syntax used by most of the high level languages are universal and a program written in the high level language can easily be understood by a second person knowing the syntax of the language.

(iii) Portability :- An application written in high level language for a particular target processor can easily be converted to another target processor / control application.

Limitation :- Some Cross - Compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions

(iii) Mixing Assembly and high level language

* Certain embedded firmware developement situations may demand the mixing of high level language with Assembly and Vice Versa.
* High level language and assembly languages are usually mixed in three ways', namely
  (a) Mixing Assembly language with High level language
  (b) Mixing High level language with Assembly
  (c) In-Line Assembly programming.

(a) Mixing Assembly language with High level language

(eg : Assembly language with 'c')

* Mixing 'c' and assembly is little complicated in the sense. the programmer must be aware of how parameters are passed from the 'c' routine to assembly and values are returned from assembly routine to 'c' and how 'Assembly - routine' is invoked from the 'c' code.

(b) Mixing high level language with Assembly

(eg : 'c' with assembly language)

Mixing the code written in a high level language like 'c' and assembly language is useful in the following scenarios :

(i) The source code is already available in assembly language and a routine written in a high level language like 'c' needs to be included to the existing code.

(ii) The entire source code is planned in Assembly code for various reasons like optimised code, optimal performance, efficient code memory etc.

(iii) To include built in library functions written in 'c' language provided by the cross compiler.

(c) In line assembly

* In line assembly is another technique for inserting target processor/ controller specific Assembly instructions at any location of a source code Written in high level language 'c'.
* This avoids the delay in clalling an assembly routine from a 'c' code
* Special keywords are Used to indicate that the start and End of assembly instructions.

ISR concept

* Interrupt means Event, Which invites the attention of processor for Some action on the hard Wlare or soft Wlare event.

1. When a device or port is ready, a device or port generates an interrupt or When it completes the assigned action, it generates an interrupt. This interrupt is called hard Wlare interrupt.
2. When soft Wlare run-time exception conditions is detected, either proce -ssor h/w or s/w instruction generates an interrupt. This interrupt is called Software interrupt or trap or exception.

* In response to the interrupt, the routine or program, Which is running at present gets interrupted and an ISR is executed.

Features of ISR

1. An ISR call due to interrupt excutes an event. Event can occur at any moment and event occurences are asynchronous.
2. ISR call is event-based diversion from the current Sequence of instructions to the another sequence of instructions. This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or s/w computational exceptional condition detected by H/w or detected by program, Which throws the exception.

4. An event can be signalled by slw interrupt instruction SWI used in device driving functions create ( ), open ( ), etc.
5. An interrupt service mechanism exists in a system to call the ISRs from multiple sources.
6. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine.

Interrupt Sources

* Hardware sources can be from internal devices or external peripherals, which interrupt the ongoing routine and there by cause diversion to corresponding ISR.
* software sources for interrupt are related to
  (i) processor detecting computational error for an illegal op-code during execution or
  (ii) execution of an SWI instruction to cause processor - interrupt of on going routine.
* There may be some other special types of sources provided in the system are

1. Hardware Interrupts Related to Internal devices

There are no. of H/w interrupt sources which can interrupt an on going program. These are processor or micro controller or internal devices h/w specific.

Ex:- Parallel port, UART serial receiver port, ADC start of conversion, ADC end, 3. synchronous receiver byte completion, Etc.

2. Hardware Interrupts related to External Devices - 1

There can be external hardware interrupt source for interrupting an on going program that also provides the ISR address or Vector address or interrupt - type Information through the data bus.

Ex:- INTR in 8086 and 80x86.

3. Hardware interrupts Related to External Devices - 2

External hard ware interrupts With their ISR Vector address are process or micro controller - specific interrupt of an on going program. External interruption - source does not send interrupt - type or ISR address - related information.

Ex :- Non - maikable pin, Within first few clock cycles Unmaikable declarable pin but otherwise maskable, Maikable pin [M for] INTO & INT1]

4. software Error - Related Hardware interrupts

These can be the software - error related interrupts generated by processor hardware. Each processor has a specific instruction set. It is designed for that set only.

* An illegal Code (instruction in the s/w) is an instruction, which does not correspond to any instruction in this set. Whenever the processor fetches illegal code, an interrupt occurs in certain processors. The error - related interrupts are also called hard ware - generated software traps (or software exceptions).

Ex :- Division by zero detection (or trap) by hard ware, over flow by h/w, Under flow by h/w, Illegal opcode by H/w.

5. Soft ware instruction - Related interrupts sources

A program can also handle specific computational errors or run - time conditions or signalling some conditions. processors provide for software instructions related to the trapes, signals or exceptions.

(i) There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling to another routine from an ongoing routine or task or thread.

(ii) Software instructions are also used for trapping some run - time error conditions and executing exceptional handlers on catching the exceptions.

INTERRUPT SERVICING (HANDLING) MECHANISM

* Each system has an interrupt Servicing (handling) Mechanism.
* The OS also provides for mechanism for interrupt handling.

(i) Interrupt Vector

* Interrupt Vector is a memory address to which the processor Vectors.
* The processor transfers the program Counter to the interrupt Vector new address on an interrupt.
* Using this address, the processor services that interrupt by executing Corresponding ISR.
* Vectoring is as per the provisions in interrupt-handling mechanism.

The Various mechanism are as follows:

Processor Vectoring to the ISR - VECTADDR

* on an interrupt, a processor Vectors to a new address, ISR-VECTADDR.
* It means that the PC (program Counter), which has the instruction address of next instruction, saves that address on stack or in some CPU register, called link register and the processor loads the ISR-VECTADDR into the PC.
* The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack.

A processor provides for one of the following Ways of using the ISR-VECTADDR based addressing mechanism.

Processor Vector Address

1. A system has internal devices like the on-chip timer and A/D Converter. In a given micro controller, each internal device interrupt source or source-group has a separate ISR-VECTADDR address. Each external interrupt pin has seperate ISR-VECTADDR.

2. In 80x86 processor architecture, a software instruction, for example 'INTn' explicitly also defines the type of interrupt and the type of defined the ISR-VECTADDR this mechanism results in the handling of n no. of exception handling routine or ISRs for n interrupt types.

3. In ARM processor architecture, the software instruction SWI does not explicitly define the type of interrupt for generating different vector address and instead there by is a common ISR-VECTADDR for each exception or signal or trap generated using SWI instruction.

A group of Interrupt sources having common Vector Address

A source group in the hardware may have the same ISR-VECTADDR. There are two types of handling mechanisms in processor hardware. The processor handling mechanism provides for fetching into the PC either.

(i) The ISR instruction at the ISR-VECTADDR or

(ii) The ISR address from the bytes at the ISR-VECTADDR.

Interrupt Vector Table :- System software designer must provide for specifying the bytes at each ISR-VECTADDR address.

* The bytes are for either ISR short code or jump instruction to the ISR first instruction. or ISR short code with call to the full code of the ISR or for fetching the bytes for finding the ISR address.

classification of Interrupt

There are three types of interrupt sources in a system.

1. Non-maskable : Examples are RAM parity error in a PC and error interrupts like division by zero. These must be serviced.
2. Maskable : Maskable interrupts are those for which service may be temperarly disable to let high priority ISRs be executed first uninterrupted.

3. Non-maskable only when defined so within few clock cycles after reset: For example, an external interrupt pin, XIRQ interrupt, in 68HC11. XIRQ interrupt is non-maskable only when defined so within few clock cycle after 68HC11 is reset.

Multiple Interrupts

Multiple Interrupt calls

When there are multiple interrupt sources, each occurance of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (Interrupt-pending resistor).

* There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priority ISR is to lower priority pending ISR.

Let us understand two processor interrupt service mechanisms for the case of multiple interrupts.

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

2. Certain processors permit in-between routine diversion to higher priority interrupts. These processors provide, in order to prevent diversion in between, a mechanism as follows: There is provisioning for masking of all interrupts by a primary levels bit. These process -ors also provisions selective diversion by provisioning for masking the interrupt service selectively by secondary level bits.

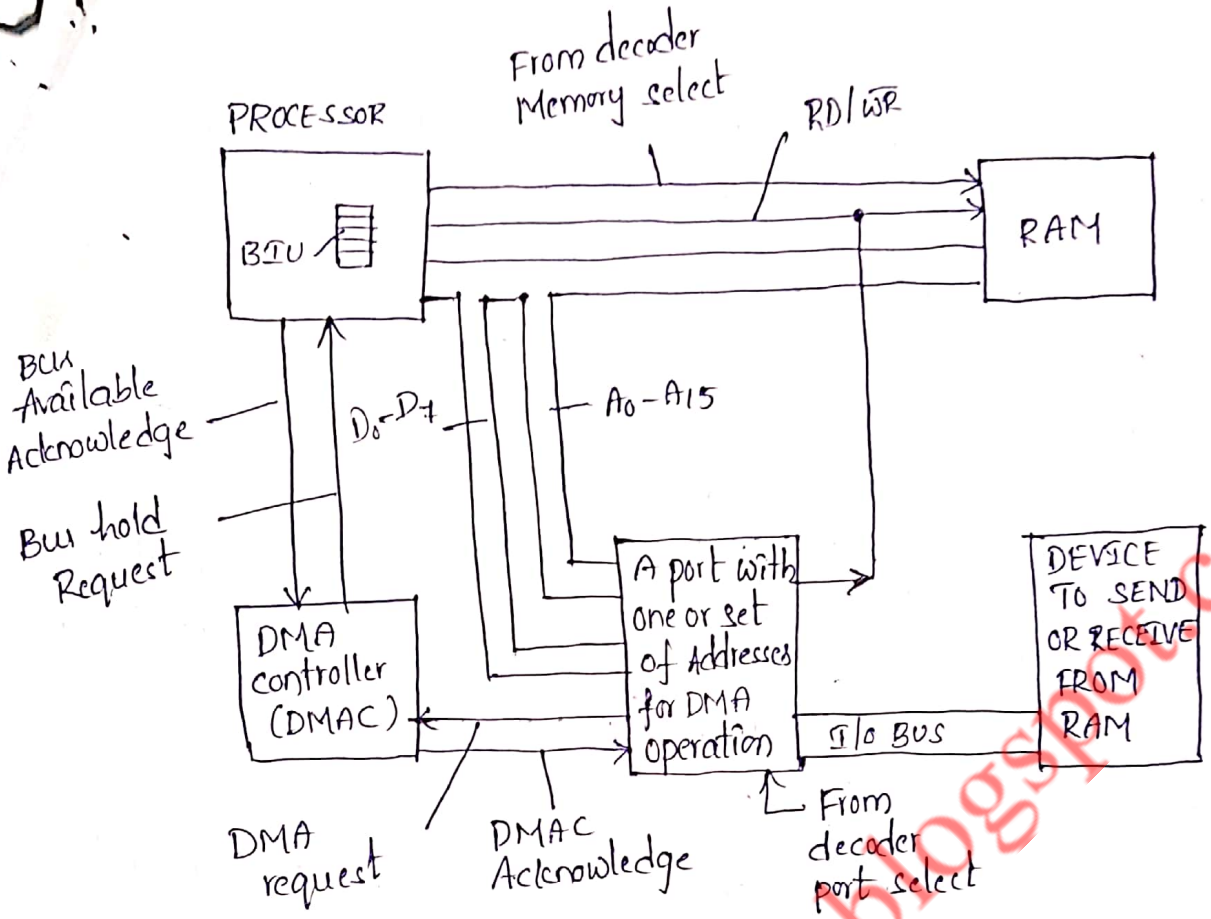
Hardware Assigned priorities

There is assigned priority order by hardware

* ARM7 provides for two types of external interrupt sources (request) IRQs and FIQs (fast interrupt requests).
* 8051 provides for priority order in order of interrupt vector address. lower address has highest and higher has the lower priority.
* Interrupts in 80x86 are assigned priority order according to interrupt-types. Interrupt of type '0' has highest priority and 225 has lowest assigned priority.

Direct Memory Access

* The DMA is used to transfer data b/w hard disk system memory.
* When the I/O's are needed for large amount data from a peripheral device to the memory address in the system or large amount of data is to be transferred by the I/O's, the interrupt-based mechanism is not suitable.
* A DMA facilitates a multi-byte data set or a burst of data or a block of data transfer b/w the external device and system or b/w two systems.
* The Device DMAC (DMA controller) data transfer occurs efficiently b/w the I/O Devices and system memory.
* The following fig. shows the interconnections using the DMAC. It also shows the buses and control signals b/w the processor, memory, DMAC and data transfering I/O device.



System buses not accessible by processor internal address and data buses during acknowledge active.

fig: The buses and control signals between the processor, memory, DMA Controller and data-transfering I/O device.

* The DMAC sends a hold request to the CPU and the CPU acknowledges that if the system memory buses are free to use.
* Three modes are usually supported in DMA operations.

(i) Single transfer at a time and then release IO bus hold on the system bus after each transfer.

(ii) Burst transfer at a time and then release of the IO bus hold on the system bus. A burst may be of a few kilobytes.

(iii) Bulk transfer and then release of the IO bus hold on the system bus only after the transfer is completed.

Use of DMAC

Whenever a DMA request is made to the DMAC for the I/O's, the DMAC is first initialized. It is programmed for

(i) read or Write
(ii) mode (bytes, burst or bulk) of DMA transfer
(iii) Total No. of bytes to be transfered and
(iv) starting memory address.

* Whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfers by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC.
* A DMAC may also provide memory access to multiple channels.

Device Driver programming

* A system has no. of physical devices, A device may have multiple functions each device function requires a driver.
* Examples of multiple functions in a device are as follows.
(i) A timer device performs timing functions as well as Counting functions. It also performs the delay function and periodic System calls.
(ii) A transceiver device transmits as well as receives.
(iii) Voice - data - fax modem device has transmitting as well as receiving as functions for Voice, fax as well as data.
* The drivers has following features
(i) The driver provides a software layer (Interface) b/w the application and actual device: When running an application, the devices are used. A driver provides a routine that facilitates the use of a device function in the application.
Ex:- For mailing system b/w layers, Network interface Card is used.

(iii) The driver facilitates the use of device by executing an ISR: simple commands from a task or function can them drive the device. Once a driver function is available for writing the codes, the application developer does not need to know anything about the mechanism, address, registers, bits and flags used by the device.

Ex:- The system clock is to be set to tick every 10,000μs.

* Generic device driver functions in high level language are used in high level language program. The functions are open, close, read, write, listen, accept etc.
* Device driver ISR programming in assembly needs an understanding of the processor, system and I/o buses and the addresses of the device register in the specific hardware.
  - → Writing physical device - Driving ISRs in a system
  - → Virtual Device drivers
  - → Parallel port drivers in a System
  - → Serial port drivers in a system
  - → Device drivers for Internal programmable Timing devices.
  - → Linux Internals as device drivers and Network functions.

Programming in Embedded C

* Whenever the convential 'c' language and its extensions are used for programming embedded systems, it is referred as "Embedd C' program.
* Desktop application development using 'c' language for a particular os platformer.
* Desktop computers contain working memory in the range of Megaby and storage memory in the range of Giga bytes.
* Embedded systems are limited in which both storage and working memory resources.

'c' V/s 'Embedded c':-

* 'c' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support.
* 'c' offers a combination of the features of high level language and assembly and helps in hard ware access programming as well as business packages developments.
* The conventional 'c' language follows ANSI standard and it incorporates various library files for different operating systems.
* A plat form (os) specific application, known as compiler is used for the conversion of program written in 'c' to the target processor specific binary files.
* Hence it is plat form specific development.
* Embedded 'c' can be considered as a subset of conventional 'c' language.
* Embedded 'c' supports all 'c' instructions and incorporates a few target processor specific functions / instructions.
* The implementation of target processor / controller specific functions / instructions depends up on the processor / controller as well as the supported cross - compiler for the particular Embedded 'c' language.
* A software program called 'cross - compiler' is used for the conversion of programs written in Embedded 'c' to target proce -ssor / controller specific instructions. (machine language).
* The standard ANSI 'c' library implementation is always tailored to the target processor / controller library files in Embedded 'c'.

Compiler Vs Cross - compiler

* Compiler is a s/w tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture.
* The development is platform specific.
* Compilers are generally termed as 'Native Compilers'.
* Native compiler generates machine code for the same machine (processor) on which it is running.
* Cross - compilers are the software tools used in cross - platform development applications.
* In cross - platform development, the compiler running on a particular target processor / os converters the source code to which machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an os which is different from the current deve -lopment environment os.
* Example is Embedded system development with Intel / AMD.
* Kail 51 is an example of Cross - Compilers.