MODULE-1 ARM 32-BIT MICROCONTROLLER.

B.S.Balaji
Asst. Prof
BGSIT.

Introduction – ARM CORTEX M3 - PROCESSOR.

→ The requirement for higher performance microcontrollers has been driven globally by the industry's changing needs; for example, microcontrollers are required to handle more work without increasing a product's frequency or power.

→ Microcontrollers are becoming increasingly connected whether by Universal Serial Bus (USB), Ethernet, or wireless radio, and hence, the processing needed to support these communication channels and advanced peripherals are growing.

→ General application complexity is on the rise due to more sophisticated user interfaces, multimedia requirements, system speed, and convergence of functionalities.

→ The ARM Cortex-M3 (32 bit) processor, the first of the Cortex generation of processors released by ARM (Advanced RISC (Reduced Instruction Set Computing) Machines) in 2006.

→ It provides excellent performance at low gate count and comes with many new features previously available only in high end processors.

→ It addresses the requirements of 32 bit embedded processor like–

(i) Greater performance efficiency - It allows more work to be done without increasing the frequency or power requirements.

(ii) Low power consumption - It enables longer battery life, especially critical in portable products like wireless networking applications.

(iii) Enhanced determinism - It guarantees that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles.

(iv) Improved code density - It ensures that code fits in even the smallest memory footprints.

(v) Ease of use - It provides easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32 bits.

(vi) Lower cost solutions - It reduces 32-bit based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32 bit micro controllers.

(vii) Wide choice of development tools - The availability of low-cost or free compilers to full-featured development suites from many development tool vendors.

→ Cortex M3 processor builds on the success of the ARM7 processor to deliver devices that are significantly easier to program and debug and yet deliver a higher processing capability.

→ It can be easily programmed using the C language and are based on a well-established architecture, application code can be ported and reused easily, reducing development time and testing costs.

→ It introduces a no. of features and technologies that meet the specific requirements of the microcontroller applications such as
* non maskable interrupts for critical tasks,
* highly deterministic nested vector interrupts,
* atomic bit manipulation, and
* an optional Memory Protection Unit (MPU).

Background of ARM and ARM Architecture

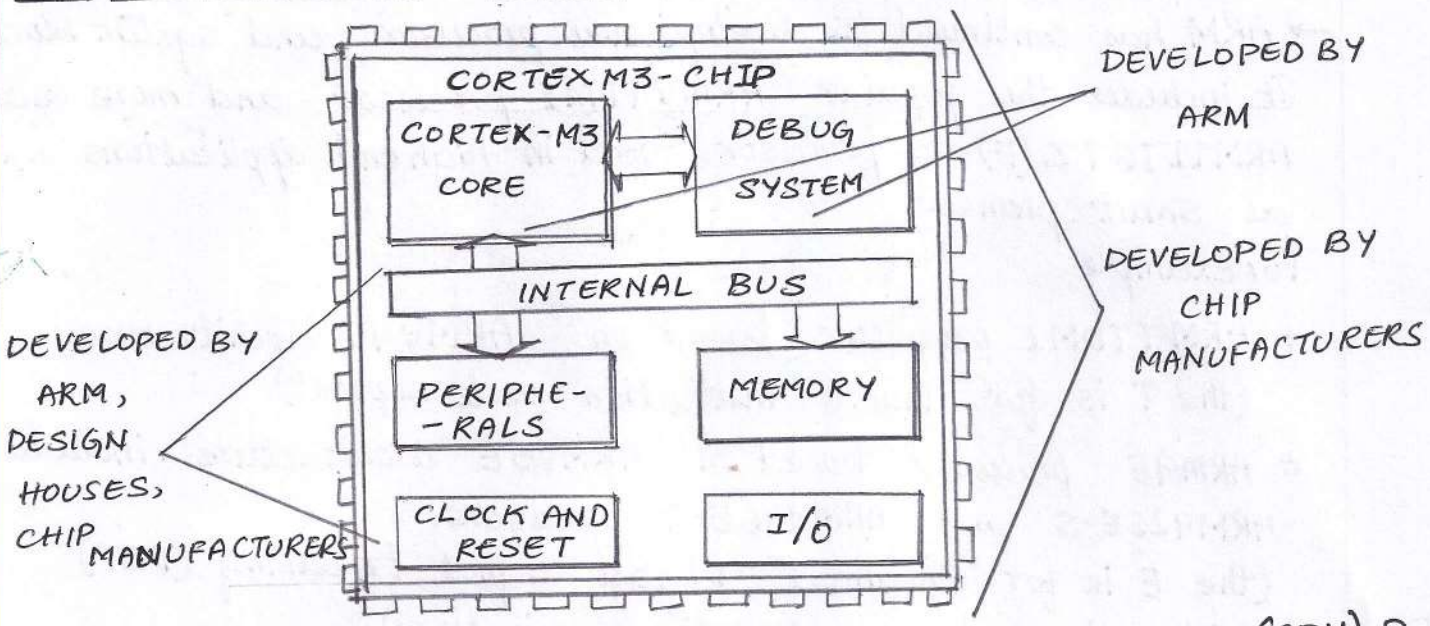
→ ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology.

→ In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee.

→ Additional companies, including Texas instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs.

→ Applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems etc.

The Cortex-M3 Processor versus Cortex-M3-Based MCUs.



→ The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a no. of other components are required for the whole Cortex-M3 processor-based microcontroller.

→ After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features.

→ Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features.

→ ARM does not manufacture processors or sell the chips directly.

→ ARM licenses the processor designs to business partners, including a majority of the world's leading semiconductor companies.

→ Based on the ARM low-cost and power efficient processor designs, these partners (such as NXP (Philips), Texas Instruments, Atmel, OKI etc.) create their processors, microcontrollers, and system on chips solutions. It is called as intellectual property (IP) licensing.

→ ARM also licenses systems-level IP and various software IPs.

→ ARM has developed a strong base of development tools, hardware and software products to enable partners to develop their own products.

ARCHITECTURE VERSIONS

→ ARM has continued to develop new processors and system blocks. It includes the popular ARM7TDMI processor and more recently ARM1176TZ(F)-S processor used in high end applications such as smart phones.

For example

* ARM7TDMI processor based on ARMv4T architecture (the T is for Thumb instruction mode support).
* ARM9E processor based on ARMv5E architecture includes ARM926E-S and ARM946E-S processors. (the E is for "Enhanced" Digital Signal Processing (DSP) instructions for multimedia applications).
* ARM11 processor based on ARMv6 architecture includes new features memory system features and Single Instruction-Multiple Data (SIMD) instructions.

ARMv6 architecture also includes the ARM1136J(F)-S, the ARM1156T2(F)-S and the ARM1176JZ(F)-S.

The architecture is divided into three profiles -

1) A-Profile (ARMv7-A): It is ~~def~~ designed for high-performance open application platforms.
   * Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) e.g., Symbian, Linux and Windows embedded.
   * These processors requires the highest processing power, virtual memory system support with memory management units (MMUs).
2) R-Profile (ARMv7-R): It is designed for high end embedded systems in which real-time performance is needed.
   * Real-Time, high ~~processors~~ performance processors targeted primarily at the higher end real time market applications like high-end breaking systems and hard drive controllers with high processing power and high reliability.
3) M-Profile (ARMv7-R): It is designed for deeply embedded micro-controller type systems.

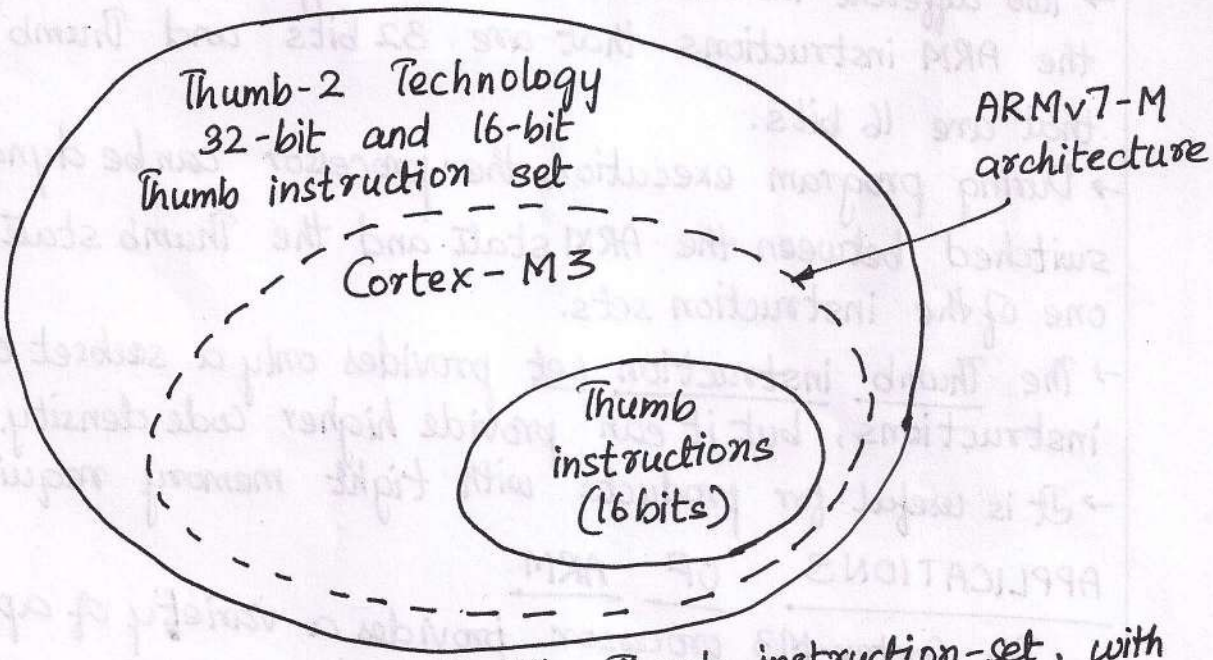
* It targets low-cost applications with processing efficiency important and cost, power consumptions, low interrupt latency and ease of use are critical as well as industrial control applications, including real-time control systems.

THE THUMB-2 TECHNOLOGY

→ The Thumb-2 technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set.

→ It delivers significant benefits in terms of ease of use, code size, and performance.

→ The Relationship between the Thumb Instruction Set in Thumb-2 Technology and the Traditional Thumb.



→ It is a superset of the previous 16-bit Thumb instruction-set, with additional 16-bit instructions alongside 32-bit instructions.

→ In 2003, ARM announced the Thumb-2 instruction set, which is a new ~~instruction~~ super set of Thumb instructions that contains both 16 bit and 32 bit instructions.

→ It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state.

→ It focuses on small memory system devices such as microcontrollers and reducing the size of the processor.

→ The Cortex-M3 supports only the Thumb-2 (and Traditional Thumb) instruction set; especially uses Thumb-2 instruction set for all operations instead of using ARM instructions for some operations as in traditional ARM processors.

→ With support for both 16 bit and 32 bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions).

→ The Cortex M3 processor also supports unaligned data accessels, a feature previously available only in high-end processors.

Instruction Set development

→ Two different instruction sets are supported on the ARM processor: the ARM instructions that are 32 bits and Thumb instructions that are 16 bits.

→ During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets.

→ The Thumb instruction set provides only a seebset of ARM instructions, but it can provide higher code density.

→ It is useful for products with tight memory requirements.

APPLICATIONS OF ARM

The Cortex-M3 processor provides a variety of applications -

1) Low-cost microcontrollers - The Cortex-M3 processor is well suited for microcontrollers, which are commonly used in consumer products, from toys to electrical appliances.

→ Its lower power, high performance, and ease of use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture.

2) Automotive - The Cortex-M3 processor has very high-performance efficiency and low latency, allowing it to be used in real time systems.

→ Hence it is ideal application for Cortex-M3 processor in automotive industry.

→ It supports upto 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional MPU which makes easily available as highly integrated and cost-sensitive automotive applications.

3) Data Communications -

→ The processor's low power and high efficiency, coupled with instructions in Thumb-2 for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.

4) Industrial Control -

→ In Industrial control ~~applications~~ applications, simplicity, fast response, and reliability are key factors.

→ The Cortex-M3 processor's interrupt feature, low interrupt latency, and enhanced fault-handling features (make it a strong candidate in this area).

5) Consumer products -

→ In many consumer ~~products~~ products, a high performance microprocessor is used.

→ The Cortex-M3 processor, is a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection.

ARCHITECTURE OF ARM CORTEX-M3

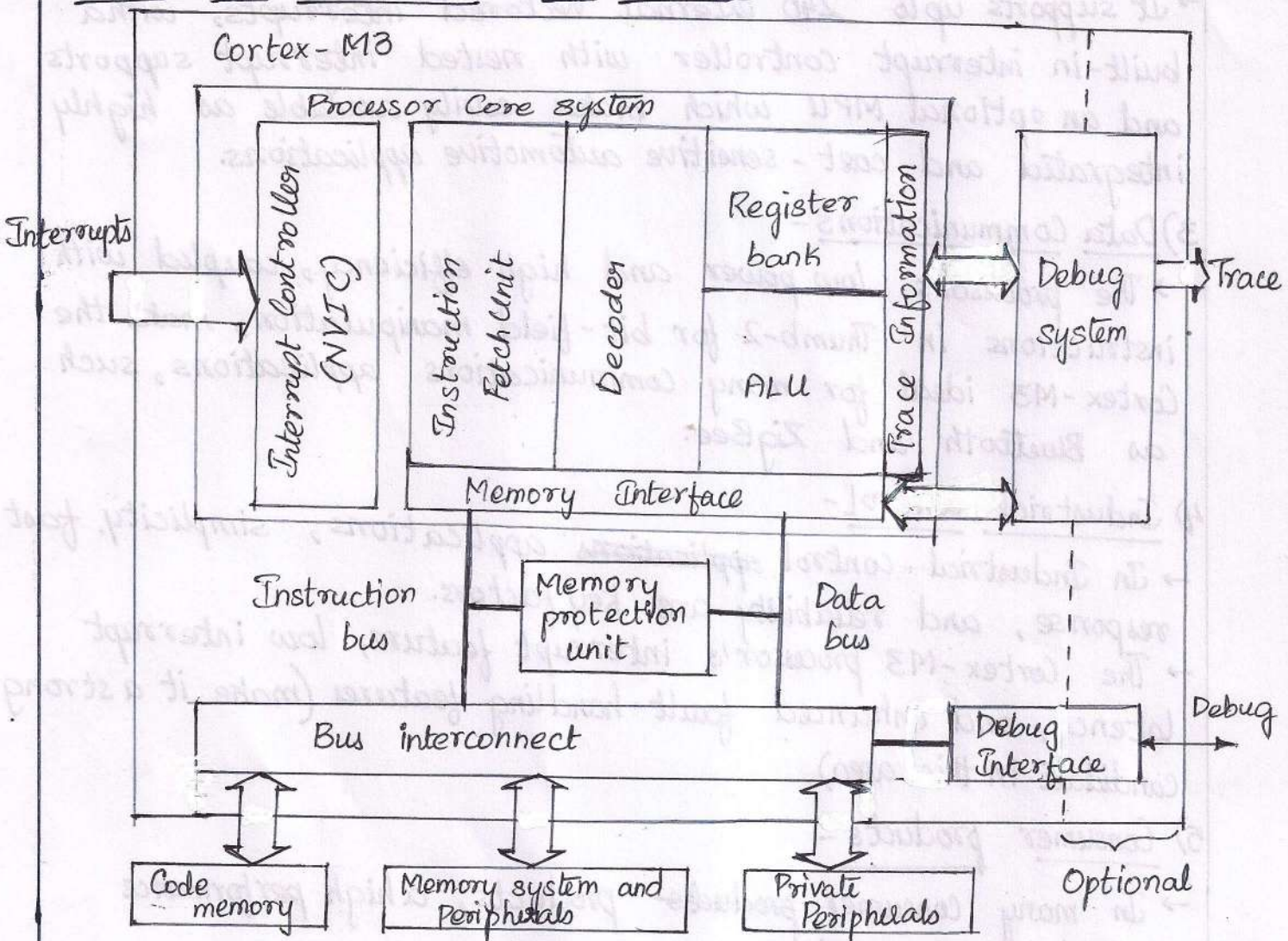
→ Arm Cortex-M3 is a 32 bit microprocessor. It has a 32-bit datapath, a 32-bit register bank, and 32-bit memory interfaces.

→ It has a Harvard architecture which means a separate instruction bus and data bus. It allows instructions and data accesses to take place at the same time.

→ As a result, the processor performance increases because data accesses do not affect the instruction pipeline.

→ It allows multiple bus interfaces in Cortex-M3 with optimized usage and ability to be used simultaneously.

Cortex M3 Architecture - an overview



→ The instruction and data buses share the same memory space (a unified memory system).

→ For complex applications, which requires more memory system features, where the Cortex-M3 processor has an optional Memory Protection Unit (MPU) and external cache is can be used. if its required

→ It supports both Little endian and big endian memory systems.

→ It includes a no. of fixed internal debugging components. These components provide debugging operation supports and features such as breakpoints and watchpoints.

→ In addition, optional components provide debugging features, such as instruction trace, and various types of debugging interfaces.

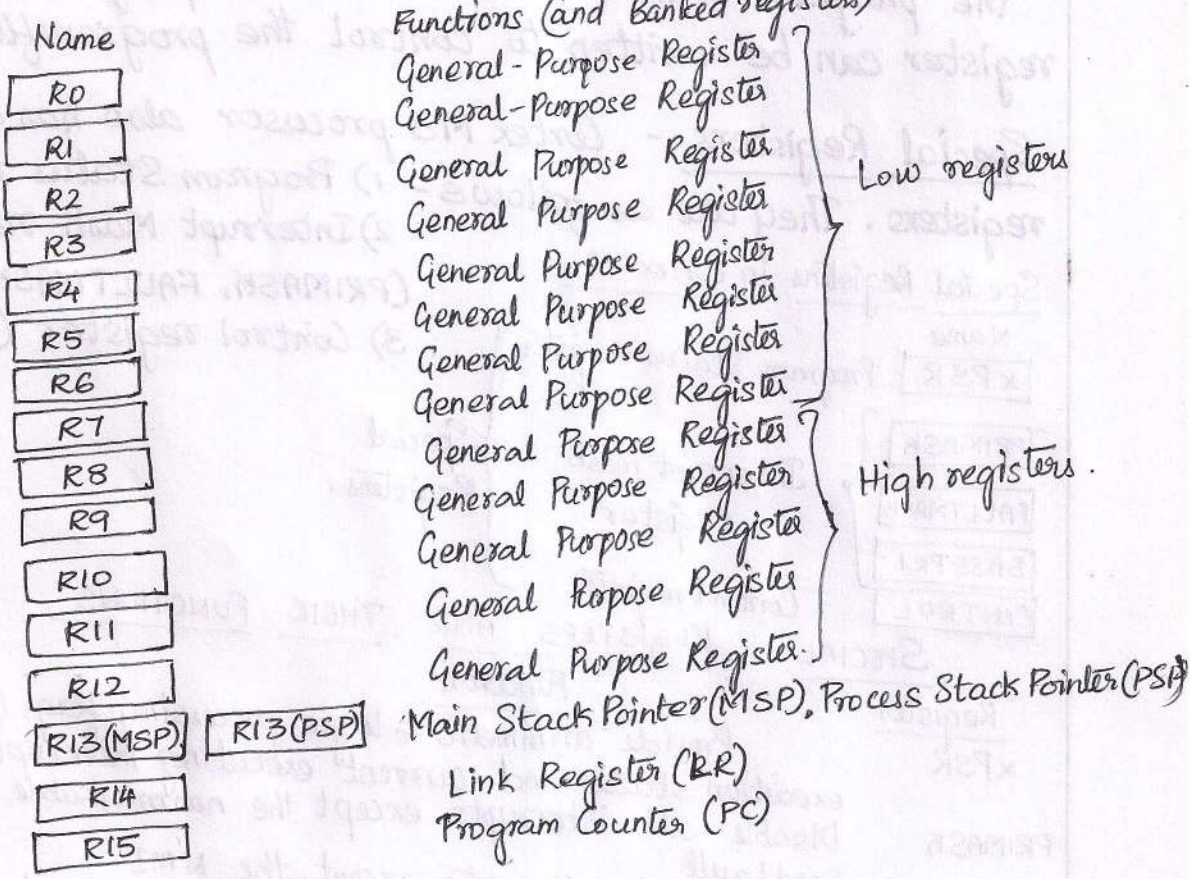
VARIOUS UNITS IN THE ARCHITECTURE-

The Cortex-M3 processor - architecture ~~xxx~~ includes various units like -

1) Registers
2) Operation Modes
3) Nested Vectored Interrupt Controller (NVIC).
4) Memory Map
5) Bus Interface
6) Memory Protection Unit.
7) Instruction Set
8) Interrupts and Exceptions
9) Debugging support.

1) Registers - ~~R0 - R12 are 32 bit~~

The Cortex-M3 processor ~~R0~~ has registers R0 through ~~R15~~, R15, R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time, R14 - link register and R15 is Program Counter.



Registers in the Cortex-M3

R0-R12 General Purpose Registers

These are 32 bit General purpose registers for data operations. Some 16-bit instructions can only access a subset of these registers (low registers, R0-R7) and 32-bit ~~registers~~ instructions can ~~only~~ access (R0-R12).

R13 - Stack Pointers

The Cortex M3 contains two stack pointers (R13). They are banked so that only one is visible at a time.

1) Main Stack pointer (MSP).- The default stack pointer, used by the operating system (OS) kernel and exception handlers.
2) Process Stack pointer (PSP)- Used by user application code.

R14 - Link Register

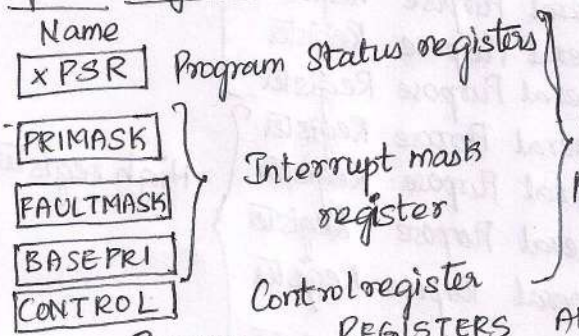
When a subroutine is called, the return address is stored in the link register.

R15- The Program Counter

The program counter is the current program address. This register can be written to control the program flow.

Special Registers - Cortex M3 processor also has a no. of special registers. They are as follows- 1) Program Status registers (PSRs)
2) Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
3) Control register (CONTROL)

Special Registers in Cortex-M3



SPECIAL REGISTERS AND THEIR FUNCTIONS

Register	Function
xPSR	Provide arithmetic & logic processing flags (zero flag & carry flag), execution status and current executing interrupt number.
PRIMASK	Disable all interrupts except the non maskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

These registers have special functions and can be accessed only by special ~~instruction~~ instructions.

2) Operation modes

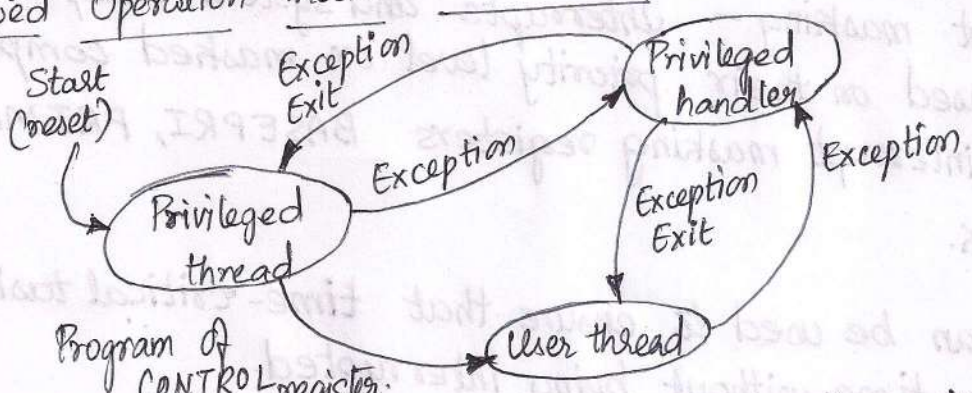
→ The Cortex-M3 processor has two modes and two privelege levels.
→ The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler.

	Privileged	User
When running an exception handler	Handler mode	
When not running an exception handler (e.g. main program)	Thread mode	Thread mode

Operation modes and Privilege levels in Cortex M3.

→ The privilege levels (privileged level and user level) provide a mechanism for safeguard memory accesses to critical regions as well as providing a basic security model.
→ Software in the privileged access level can switch the program into the user access level using the control register.
→ When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler.

Allowed Operation mode Transitions.



→ A user program cannot change back to the privileged state by writing to the control register.
→ It has to go through an exception handler that programs the control register to switch the processor back into privileged access level when returning to thread mode.

3) Built-in NESTED VECTORED INTERRUPT CONTROLLER.

→ Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC).

→ It provides a no. of features - 1) Nested interrupt support,
2) Vectored interrupt support
3) Dynamic priority changes support
4) Reduction of interrupt latency
5) Interrupt masking.

1) Nested Interrupt Support - The NVIC provides nested interrupt support. All the external interrupts and most of the system exceptions can be programmed to different priority levels.

2) Vectored interrupt support - When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory.

3) Dynamic Priority changes support - Priority levels of interrupts can be changed by software during run time.

4) Reduction of interrupt latency - It includes number of advanced features to lower the interrupt latency like automatic saving and restoring some register contents, reducing delay in switching from one ISR to another and handling of late arrival of interrupts.

5) Interrupt masking - Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK and FAULTMASK.

They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

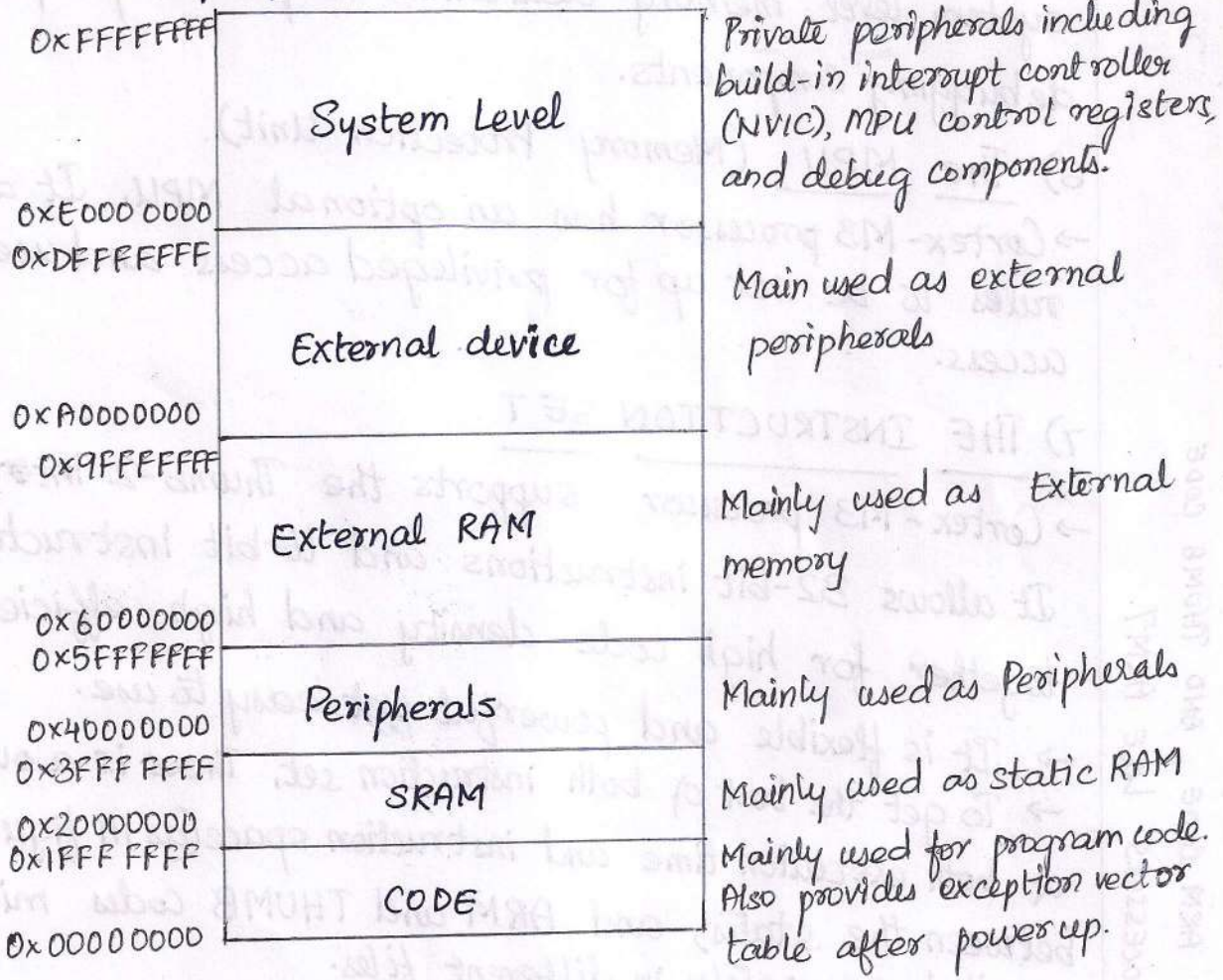
4) Memory Map

→ Cortex M3 has a predefined memory map. It allows the built-in peripherals, such as interrupt controller and debug components to be accessed by simple memory access instructions.

→ The predefined memory map also allows the processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

Overall, the 4GB memory space can be divided in to ranges as shown in figure.

The Cortex-M3 memory Map



5) The BUS Interface -

Cortex-M3 processor has ~~many~~ Three bus interfaces which allows to carry instruction fetches and data accesses at the same time.

1) Code memory buses
2) System bus
3) Private peripheral bus.

→ The Code memory region access is carried out on the Code memory buses, which physically consist of two buses - I-Code and D-Code.

These are optimized for instruction fetches for best instruction execution speed.

→ The system bus is used to access memory and peripherals.

It provides access to the ~~Static~~ SRAM, peripherals, External ~~RAM~~ RAM, external devices, and part of the system-level memory regions.

→ The private peripheral bus provides access to a part of the system level memory dedicated to private peripherals, such as debugging components.

6) The MPU (Memory Protection Unit).

→ Cortex-M3 processor has an optional MPU. It allows access rules to be set up for privileged access and user program access.

7) THE INSTRUCTION SET

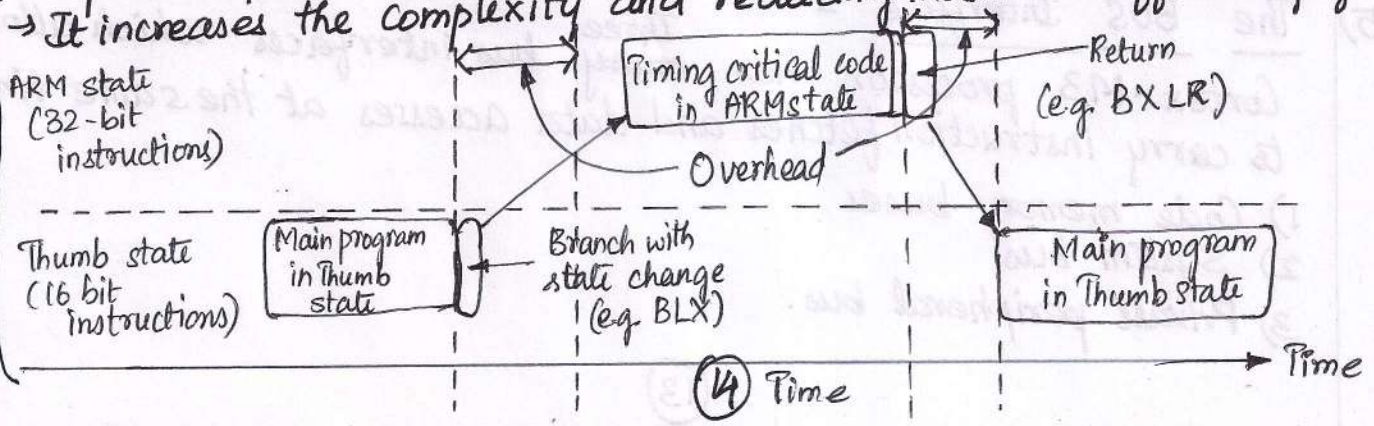
→ Cortex-M3 processor supports the Thumb-2 ~~into~~ instruction set. It allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency.

→ It is flexible and powerful yet easy to use.

→ To get the best of both instruction set, There is a overhead (interms of both execution time and instruction space(as in figure)) to switch between the states, and ARM and THUMB codes might need to be compiled separately in different files.

→ It increases the complexity and reducing maximum efficiency of core.

SWITCHING BETWEEN ARM CODE AND THUMB CODE IN TRADITIONAL ARM PROCESSORS LIKE ARM7.



8) Interrupts and Exceptions

→ Cortex M3 processor implements a new exception model, enabling very efficient exception handling.
→ It has a no.of system exceptions plus a no.of external Interrupt Request (IRQs) (external interrupt inputs).
→ There is no fast interrupt (FIQ).
→ It supports nested interrupts (a higher-priority interrupt can override or preempt a lower-priority Interrupt handler). behaves just like FIQ.

9) Low power Consumption

The Cortex-M3 processor is suitable for various low-power applications:
→ The Cortex-M3 processor is suitable for low power designs because of the low gate count.
→ It has power-saving mode support (SLEEPING and SLEEPDEEP).
→ The processor can enter sleep mode using WFI or WFE instructions.
→ The design has separated clocks for essential blocks, so clocking circuits for most parts of the processor can be stopped during sleep.
→ The fully static, synchronous, synthesizable design makes the processor easy to be manufactured using any low power or standard semiconductor process technology.

10) Debug supports

The Cortex M3 processor includes ~~de~~ Comprehensive debug features (to help software developers design their products)-
→ Supports JTAG or Serial-wire debug interfaces
→ Based on the Coresight debugging solution, processor status or memory contents can be accessed even when the core is running.

→ Built-in support for six breakpoints and four watchpoints.
→ Optional ETM for instruction trace and data trace using DWT.
→ New debugging features, including fault status registers, new fault exceptions, and Flash Patch operations, ~~making~~ make debugging much easier.
→ ITM provides an easy-to-use method to output debug information from test code.
→ PC sampler and counters inside the DWT provide code-profiling information.

GENERAL PURPOSE REGISTERS

→ Cortex-M3 processor has registers R0 through R15 and a no. of special registers.

→ R0 through R12 are general purpose, but some of 16 bit Thumb instructions can only access R0 through R7 (low registers), whereas 32 bit Thumb-2 instructions can access all these registers.

→ Special registers have predefined functions and can only be accessed by Special register access instructions.

Registers in the Cortex-M3.

Name	Functions (and banked registers)	
R0	General Purpose Register	Low registers
R1	General Purpose Register	
R2	General Purpose Register	
R3	General Purpose Register	
R4	General Purpose Register	
R5	General Purpose Register	
R6	General Purpose Register	
R7	General Purpose Register	
R8	General Purpose Register	High registers
R9	General Purpose Register	
R10	General Purpose Register	
R11	General Purpose Register	
R12	General Purpose Register	
R13 (MSP) / R13 (PSP)	Main Stack Pointer (MSP) / Process Stack Pointer (PSP)	
R14	Link Register (LR)	
R15	Program Counter (PC)	
xPSR	Program status Registers (PSRs)	Special registers.
PRIMASK	Interrupt mask registers	
FAULTMASK		
BASEPRI		
CONTROL	Control register	

General Purpose Registers R0 through R7.

→ These are also called as low registers. They can be accessed by all 16 bit Thumb instructions and all 32-bit Thumb-2 instructions.

→ They are all 32 bits; the reset value is unpredictable.

General Purpose Registers R8 through R12 — Thumb-2

→ These are also called as High registers. They are accessible by all instructions but not by all 16-bit instructions.

→ These registers are all 32 bits; the reset value is unpredictable.

Stack Pointer R13

→ R13 is the stack pointer. There are two SPs in Cortex-M3 Processor.

→ Its duality allows two separate stack memories to be set up.

→ The use of Special instructions allows the access of current SP, and other one is inaccessible through move to special register from general purpose register (MSR) and move special register to general-purpose register (MRS).

1) Main Stack Pointer or SP_main - It is the default SP. It is used by the operating system (OS) kernel, exception handlers, and all application codes that require privileged access.

2) Process Stack Pointer or SP_process - It is used by the base-level application code (when not running an exception handler).

→ The instructions for accessing stack memory are PUSH and POP.

example -

```
PUSH  [R0];   R13 = R13 - 4, then memory [R13] = R0
POP   [R0];   R0 = Memory [R13], then R13 = R13 + 4
```

→ The Cortex-M3 uses a full-descending stack arrangement.

Link Register R14

→ R14 is the link register (LR). It is used to store the return program Counter (PC) when a subroutine or function is called. for example

```
main : Main program (In assembly language, it can be used as R14 or LR)
       ....
       BL function1 ; Call function1 using Branch with link instruction
                    ; PC = function1 and
                    ; LR = the next instruction in main
       ....
       function1    ; Program code for function1
       ....
       BX LR ;  Return.
```

Program Counter R15

→ It is the Program Counter (PC) and used in assembler code as R15 or PC.

→ The value in PC is different than the location of executing instruction, normally by 4. ex- 0x1000 : MOV R0, PC ; R0 = 0x1004

SPECIAL REGISTERS

The special registers in the Cortex-M3 processor include ~~the~~ -

1) Program Status Registers (PSRs)
2) Interrupt Mask Registers (PRIMASK, FAULTMASK, and BASEPRI).
3) Control Register.

→ Special registers can only be accessed via MSR and MRS instructions; they do not have memory addresses:

MRS <reg>, <special_reg> ; Read special register
MSR <special_reg>,<reg>; write to special register.

Program Status Register

The PSRs are subdivided into three status registers:

→ All3 can be accessed together or separately using the special register access instructions MSR and MRS.

→ When they are accessed as a collective term, the name xPSR is used.

Combined Program Status Register (xPSR) in the Cortex-M3.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception number				

Bit fields in Cortex-M3 Program Status registers.

Bit	Description
N	Negative
Z	Zero
C	Carry/Borrow
V	Overflow

Bit	Description
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits/IF-THEN instruction status bit.
T	Thumb state, always 1; trying to clear this bit will cause exception (fault)
Exception number	Indicates which exception the processor is handling.

→ PSRs ~~be~~ can read using the MRS instruction. whereas APSR can also be changed using the MSR instruction, but EPSR and IPSR are readonly. for example -

```
MRS  r0, APSR; Read flag state into R0
MRS  r0, IPSR; Read Exception/Interrupt state
MRS  r0, EPSR; Read Execution state
MSR  ~~~,APSR,r0; Write flag state.
```

In ARM assembler, when accessing xPSR (all 3 PSRs as one), the symbol PSR is used:

```
MRS  r0, PSR ; Read the combined Program status word
MSR  PSR, r0 ; Write ~~the~~ combined Program state word.
```

2 Interrupt Mask registers consists of PRIMASK, FAULTMASK, and BASEPRI registers which are used to disable exceptions.

→ The PRIMASK and BASEPRI registers are useful to temporarily disabling interrupts in timing-critical tasks.

→ The FAULTMASK gives the OS kernel time to deal with fault conditions and are used to temporarily disable fault handling when a task crashed.

→ In assembly language, the MSR and MRS instructions are used. For example:

```
MRS  r0, BASEPRI ;  Read BASEPRI register into R0
MRS  r0, PRIMASK;  Read PRIMASK register into R0
MRS  r0, FAULTMASK; Read FAULTMASK register into R0
MSR  BASEPRI, r0 ;  Write R0 into BASEPRI register
MSR  PRIMASK, r0;  Write R0 into PRIMASK register
MSR  FAULTMASK, r0; Write R0 into FAULTMASK register
```

These registers cannot be set in the user access level.

Cortex-M3 Interrupt Mask Registers

PRIMASK - A 1 bit register, when it is set, it allows NMI and hard fault exception; all other interrupts and exceptions are masked.
The default value is 0, which means that no masking is set.

FAULTMASK - A 1 bit register, when it is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled.
The default value is 0, which means that no masking is set.

BASEPRI - A register of upto 8 bits where it defines masking priority level. When it is set, it disables all interrupts of the same or lower level (larger priority value). Higher priority interrupts can still be allowed.
The default value is set to 0, the masking function is disabled.

The Control Register

→ It is a 2bit register and is used to define the privilege level and the SP selection.

CONTROL[1] bit - is always 0 in handler mode and however in thread or base level, it can be either 0 or 1.

→ It is writable only when the core is in thread mode and privileged.

→ In the user state or handler mode, writing to this bit is not allowed.

→ Another way to change it ~~is through~~ by changing bit 2 of the LR when in exception return.

CONTROL[0] bit - is writable only in a privileged state. Once it enters the user state, the only way to switch back to privileged is to trigger an interrupt and change this in the exception handler.

To access the control register, in assembly - ~~the~~

```
MRS   r0, CONTROL; Read CONTROL register into R0.
MSR   CONTROL, r0; Write R0 into CONTROL register.
```

Cortex-M3 CONTROL REGISTER.

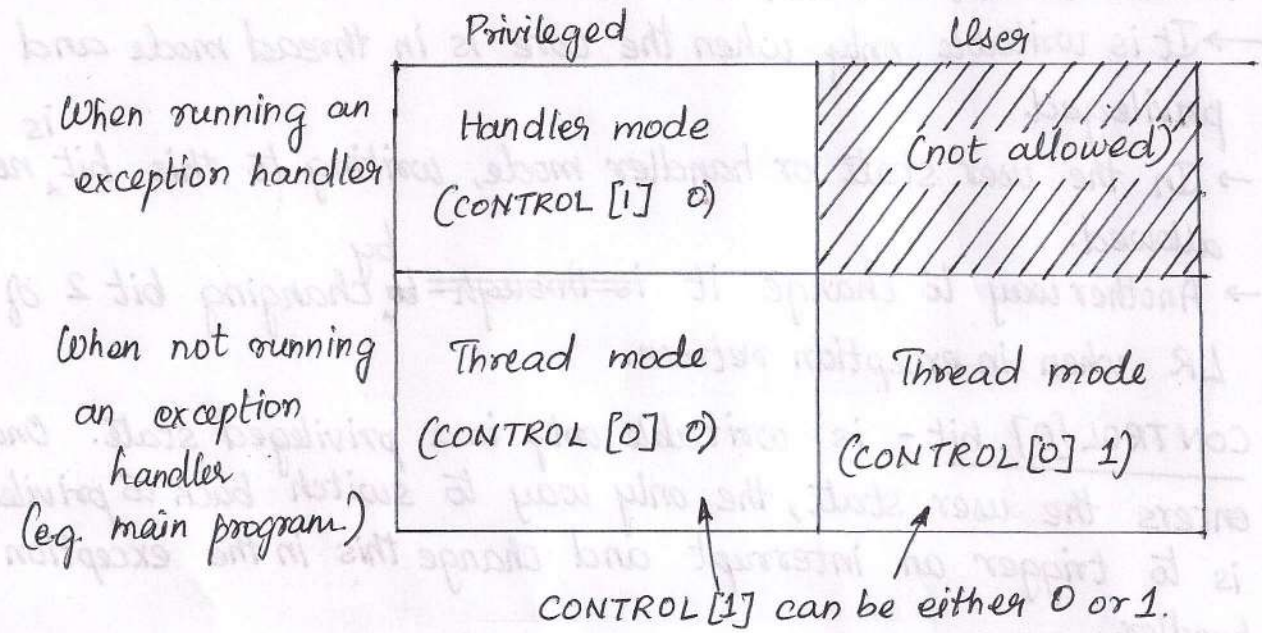
Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used (PSP) 0 = Default stack (MSP) is used. In thread or base level, the alternate stack is PSP. No alternate stack in handler mode. Processor is in handler mode, when the bit is 0.
CONTROL[0]	0 = Privileged in thread mode. 1 = User state in thread mode.

If in handler mode (not thread mode), the processor operates in privileged mode.

OPERATION MODE

The Cortex-M3 processor supports two modes and two privilege levels.

Operation Modes and Privilege Levels in Cortex-M3.



→ When the processor is running in thread mode, it can be in either the privileged or user level, but handlers can only be in the privileged level.

→ When the processor exits reset, it is in thread mode, with privileged access rights.

→ In the user level (thread mode), access to the system control space (SCS) – a part of the memory region for configuration registers and debugging components – is ~~blocked~~ blocked.

→ Instructions that access special registers (like MSR, except when accessing APSR) cannot be used.

→ When a programming is running, at user level tries to access SCS or special registers, a fault exception will occur.

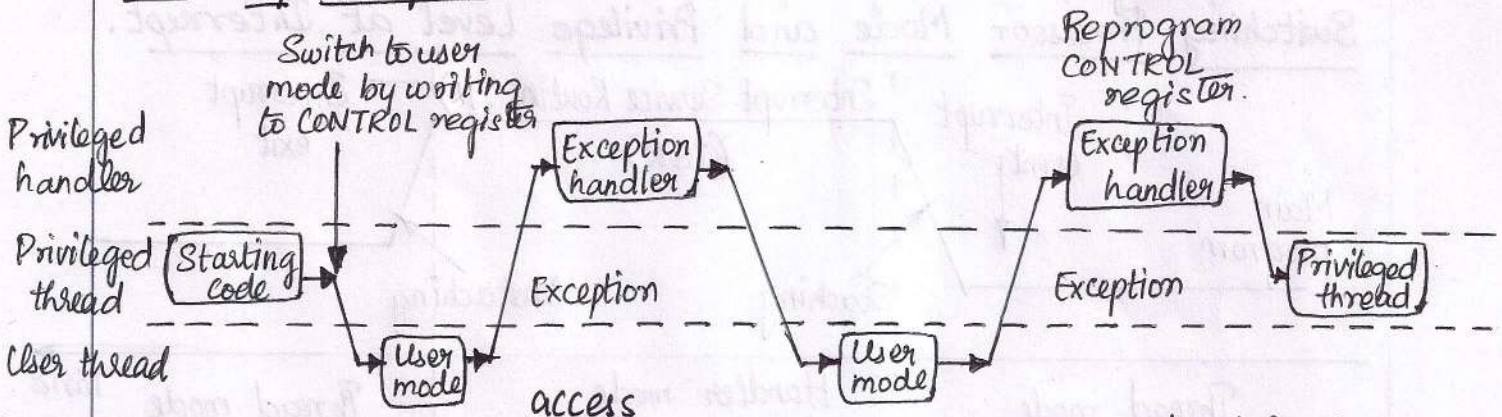
→ Software in a privileged access level can switch the program into the user access level using the control register.

→ When an exception takes place, the processor will always switch to a privileged state and return to the previous state when exiting the exception handler.

→ A user program cannot change back to the privileged state directly by writing to the control register.

⇒ It has to go through an exception handler that programs the control register to switch the processor back into privileged access level when returning to thread mode.

Switching of Operation Mode by Programming the Control Register or by Exceptions.



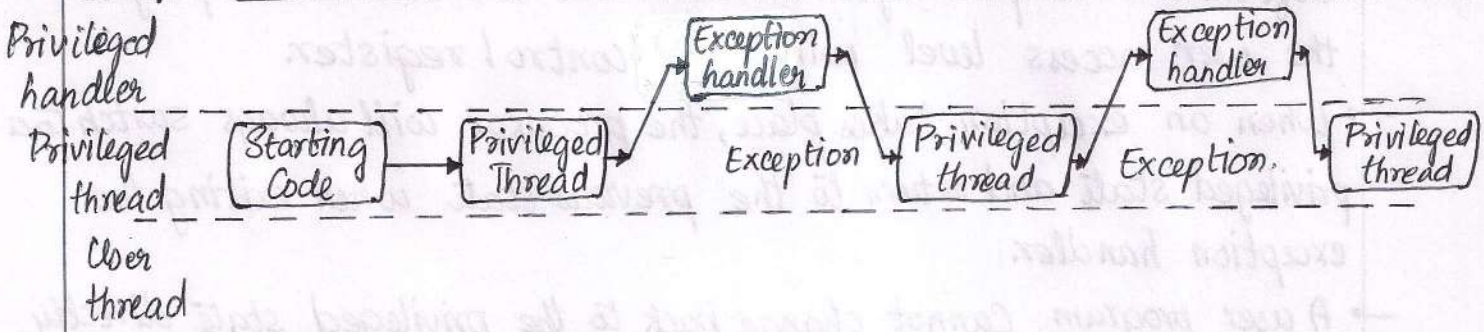
→ The support of both levels provides a more secure & robust architecture. Ex- When a user program goes wrong due to NVIC, it will not be able to corrupt control registers. MPU avoids the user from accessing memory regions used by privileged processes and blocks user programs.

→ In simple applications, there is no need to separate the privileged and user access levels. There is no need to use user access level & no need to program the control register.

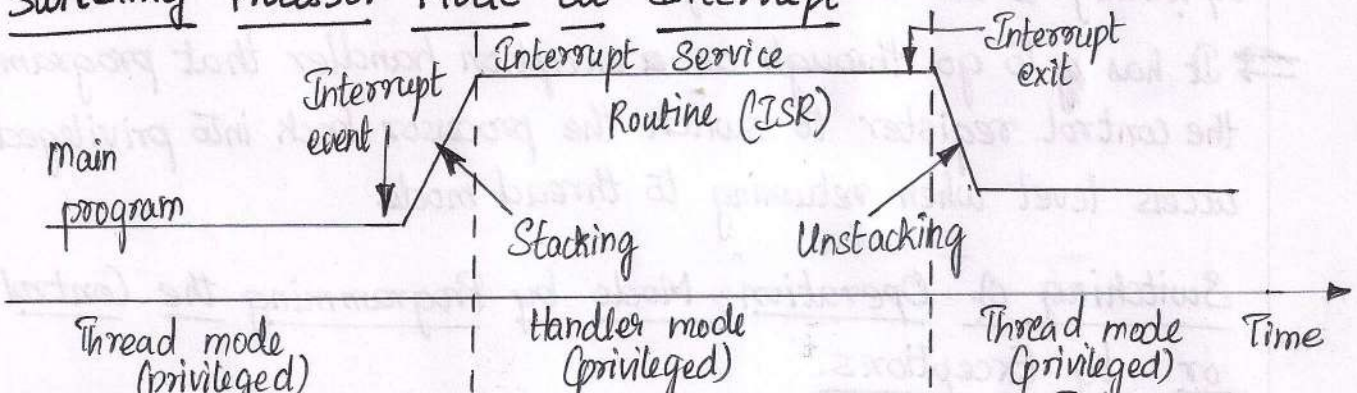
→ The user application stack is can be separated from the kernel stack memory to avoid crashing a system due to stack operation errors in user program.

→ Acc. to diagram, User program running in thread mode uses PSP, & the exception handlers use the MSP. The switching of SPs is automatic upon entering or leaving the exception handlers.

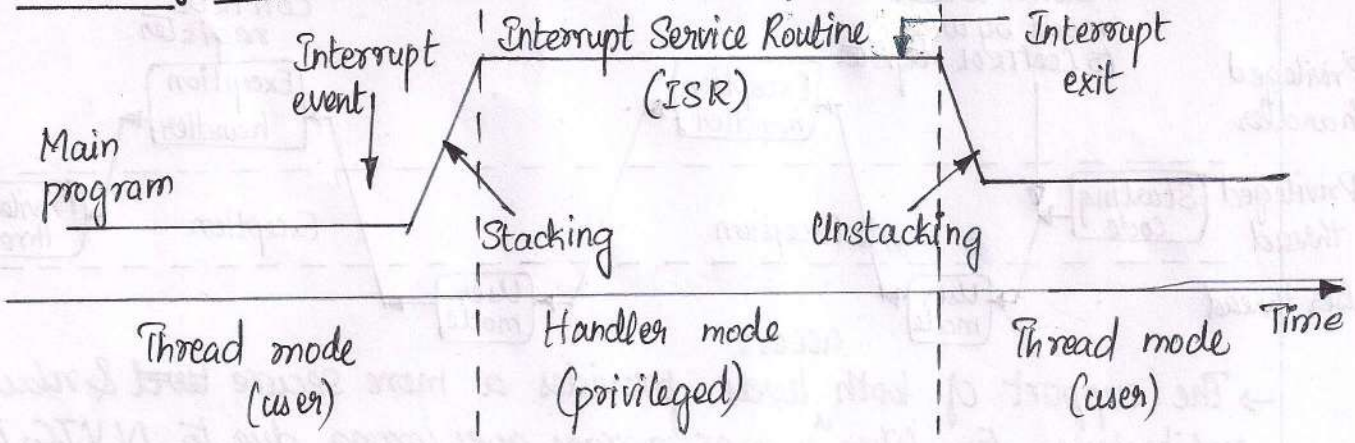
Simple Applications Do not Require User Access Level in Thread mode



Switching Processor Mode at Interrupt



Switching Processor Mode and Privilege Level at Interrupt.



→ The mode & access level of the processor are defined by the control register. When the CR bit 0 is 0, the processor mode changes when an exception takes place.

→ When control register bit 0 is 1 (thread running user application), both processor mode and access level change when an exception takes place.

→ CR bit 0 is programmable only in the privileged level. For a user-level program to switch to privileged state, it has to raise an interrupt.
Ex- Supervisor call [SVC], and write to CONTROL[0] within the handler.

DEBUGGING SUPPORT -

→ Cortex-M3 processor includes a no. of debugging features such as program execution controls including halting and stepping, instruction breakpoints, data watchpoints, registers and memory accesses, profiling and traces.

→ Debugging hardware in Cortex M3 processor is based on Core sight architecture.

→ It does not have a JTAG interface like Traditional ARM processor instead DAP (Debug Access Port) is a debug interface module is decoupled from the core and a bus interface at core level.

→ External debuggers can access control registers to debug hardware as well as System memory. When the processor is running.

→ Debug Port (DP), device controls the DAP bus interface and it is available as SW-DP (Serial wire - JTAG). It supports traditional JTAG protocol as Serial wire protocol.

→ JTAG DP module from ARM core sight product family can also be used.

→ Chip manufacturers can use/choose attach one of these Debug port module to provide the debug interface.

→ It also includes -

ETM (Embedded Trace Macro cell) allows instruction trace and traced information is output through TPIU (Trace Port Interface Unit) and the debug host (PC) can collect the executed instruction information via external trace capturing hardware.

Events - In Cortex-M3 processor, a no. of events can be used to trigger ~~bebu~~ debug actions.

→ Debug events can be breakpoints, watchpoints, fault conditions, or external debugging request i/p signals.

→ When a debug event takes place, the Cortex M3 processor can either halt mode (or) execute the debug monitor exception handler.

→ Data watch point function is provided by a Data Watch Point and Trace (DWT) in Cortex M3 processor

→ It can be used to stop the processor (or trigger the debug exception routine) or to generate data trace information.

→ When data trace is used, traced data can be output via the TPIU. Multiple trace devices can share one single trace port (TPIU).

→ It ~~allows~~ also provides a Flash Patch and Break point (FPB) unit which can perform a simple breakpoint function (or) remap an instruction access from Flash to a different location in SRAM.

→ ITM (Instrumentation Trace Macro cell) provides a new way for developers to output data to a debugger by writing data to register memory.

→ Debugger can collect the data via trace ~~information interface~~ interface and display or process them.

→ It is easy to use and Faster than JTAG output.

→ The Debugging components can be controlled by DAP interface bus (or) Program running on the Processor core.

→ All Trace information is accessible from the TPIU.

EXCEPTIONS AND INTERRUPTS

→ The Cortex-M3 supports a no. of exceptions, including a fixed no. of system exceptions and a no. of Interrupts commonly called IRQ.

→ The no. of interrupt inputs on a Cortex-M3 microcontroller depends on the individual design.

→ The typical no. of interrupt inputs is 16 or 32 bit.

→ It is generated by peripherals, except System Tick Timer also connected to interrupt input signals.

→ There is also a NMI (non maskable interrupt) input signal which depends on the design of the microcontroller or system-on-chip (SoC).

→ NMI could be connected to a Watch dog timer or a voltage-monitoring block that warns the processor when voltage ~~drops~~ drops below a certain level.

→ NMI can be activated at any time, even right after the core exits reset.

→ A no. of the system exceptions are fault-handling exceptions that can be triggered by various error conditions.

→ NVIC also provides a no. of fault status registers so that error handlers can determine the cause of the exceptions.

VECTOR TABLES

→ When an exception event takes place on the Cortex-M3 and is accepted by the Processor Core, the corresponding exception handler is executed.

→ Exception handler address is identified by Vector Table mechanism.

→ The Vector table is an array of word data inside the system memory, each representing the starting address of one exception type.

→ The Vector table is relocatable and the relocation is controlled by a relocation register in the NVIC.

→ After reset, Relocation control register is reset to 0. Vector table is located in address 0x00 after reset (exception type 1), the address of reset vector is 1 times of 4 (each word is 4 bytes), ~~and NMI (type 2)~~ is located at ~~0x04~~. 0x000004 and NMI (type 2) is located in $2 \times 4 = 0x00000008$.

→ The address 0x000 00000 is used to store the starting value for the MSP.

→ The LSB of each exception vector indicates whether the exception is to be executed in the Thumb state.

→ Cortex-M3 supports only Thumb instructions, the LSB of all the exception vectors should be set to 1.

Exception Types in Cortex M3

Exception Number	Exception Type	Priority	Function.
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Nonmaskable interrupt
3	Hard fault	-1	All classes of fault, when corresponding fault handler cannot be activated because it is currently disabled or masked by exception mating
4	Mem Manage	Settable	Memory management fault
5	Bus fault	Settable	Error response received from the bus system
6	Usage fault	Settable	Usage fault;
7-10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16-255	IRQ	Settable	IRQ input #0-239

Vector Table Definition after Reset.

Exception Type	Address Offset	Exception Vector
18 - 255	0x48 - 0x3FF	IRQ #2-239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug monitor
11	0x2C	SVC
7-10	0x1C - 0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x0C	Starting value of the MSP.

STACK MEMORY OPERATIONS :-

→ In Cortex-M3 processor, besides normal software-controlled stack PUSH and POP, the stack PUSH and POP operations are also carried out automatically when entering or exiting an exception / interrupt handler.

Basic operations of the Stack.

→ In general, stack operations are memory write or read operations with the address specified by SP.

→ Data in registers is saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation.

→ The SP is adjusted automatically in PUSH and POP so that multiple data PUSH will not cause old stacked data to be erased.

→ The function of the stack is to store register contents in memory so that they can be restored later, after a processing task is completed.

Stack Operation Basics: One Register in Each Stack Operation

Main program

```
    ....
    ; R0 = X, R1 = Y, R2 = Z
    BL function1
```

Subroutine

```
function1
PUSH {R0}; store R0 to stack & adjust SP
PUSH {R1}; store R1 to stack & adjust SP
PUSH {R2}; store R2 to stack & adjust SP
....; Executing task (R0, R1, and R2
    ; could be changed)
POP  {R2}; restore R2 and SP re-adjusted
POP  {R1}; restore R1 and SP re-adjusted
POP  {R0}; restore R0 and SP re-adjusted
BX   LR ; Return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
....; next instructions
```

→ For each store (PUSH), there must be a corresponding read (POP) and the address of the POP operation should match that of the PUSH operation.

→ When PUSH/POP instructions are used, the SP is incremented/decremented automatically.

→ When program control returns to the main program, the R0-R2 contents are the same as before.

→ Notice the order of PUSH and POP: The POP order must be the reverse of PUSH.

Stack Operation Basics: Multiple Register Stack operation.

```
Main program
    ....
    ; R0 = X, R1 = Y, R2 = Z
    BL function1 ------------------> function1
                                     PUSH {R0-R2}; store R0, R1, R2 to stack
                                     ...; Executing task (R0, R1 and R2
                                        ; could be changed)
                                     POP {R0-R2}; restore R0, R1, R2
                                     BX  LR ; Return
; Back to main program
; R0 = X, R1 = Y, R2 = Z
....; next instructions
```

→ These operations can be simplified, ~~thanks~~ to with PUSH and POP instructions allowing multiple load and store.

→ In this case, the ordering of a register POP is automatically reversed by the Processor.

Stack Operation Basics: Combining Stack POP and RETURN

```
Main program
    ....
    ; R0 = X, R1 = Y, R2 = Z
    BL function1 ------------------> function1
                                     PUSH {R0-R2, LR}; Save registers
                                        ; including link register
                                     .... ; Executing task (R0, R1 and R2
                                        ; could be changed)
                                     POP  {R0-R2, PC}; Restore registers and
                                        ; return
; Back to main program
; R0 = X, R1 = Y, R2 = Z
....; next instructions
```

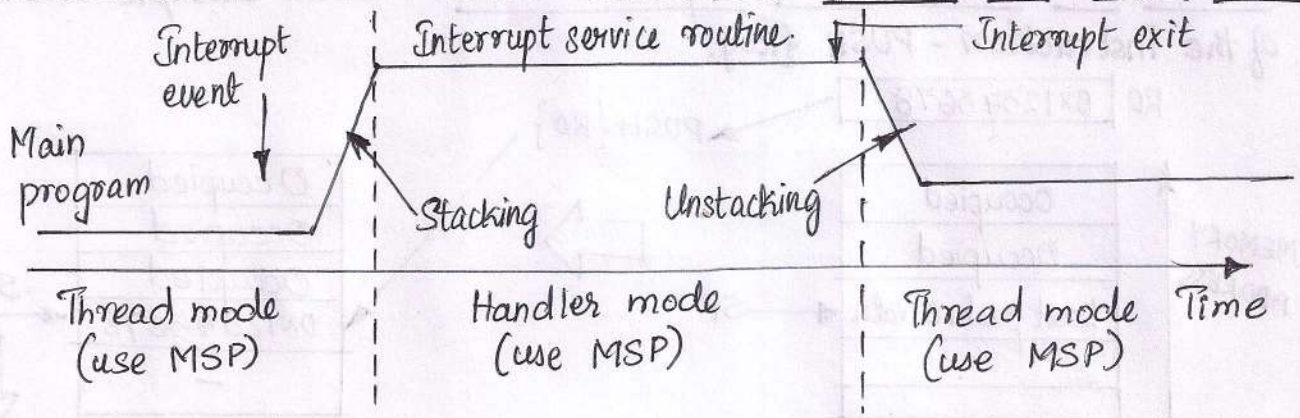
Subroutine

→ It combines RETURN with a POP operation. It is done by pushing the LR to the stack and popping it back PC at the end of the subroutine.

The Two-stack model in the Cortex-M3

The Cortex M3 has two SPs: the MSPs and the PSP. The SP register to be used is controlled by the ~~control~~ control register bit 1. (CONTROL[1]).

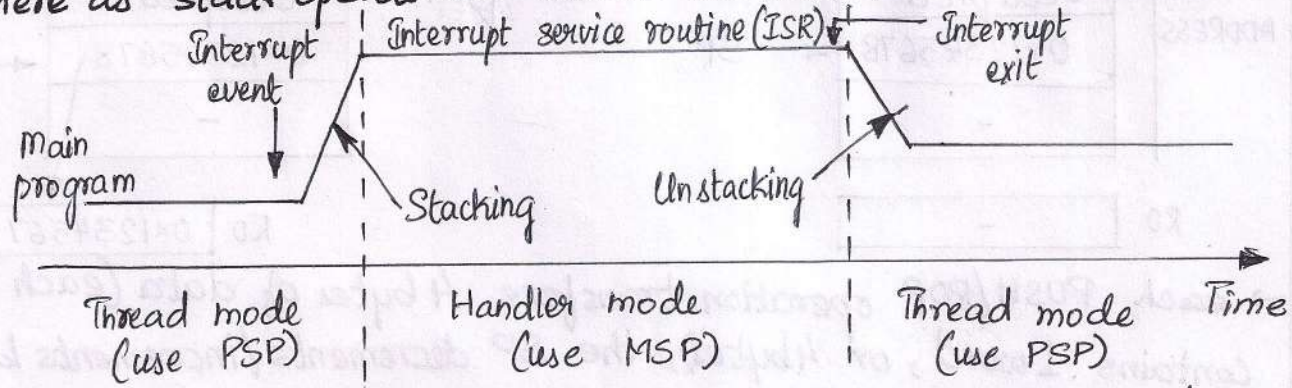
CONTROL[1]=0: Both Thread level and Handler Use Main Stack.



→ When ~~control~~ CONTROL[1] is 0, the MSP is used for both thread mode and handler mode.
→ Here, the main program and the exception handlers share the same stack memory region. It is the default setting after power-up.

CONTROL[1]=1: Thread level Uses Process Stack and Handler Uses Main Stack.

(Note - the automatic stacking and unstacking mechanism will use PSP, where as stack operations inside the handler will use MSP).



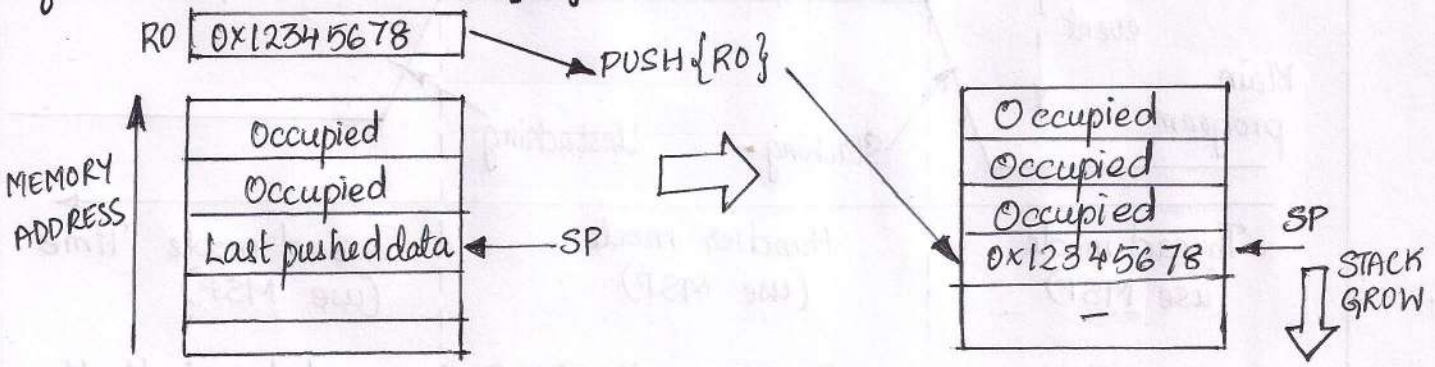
→ When the CONTROL[1] is 1, the PSP is used in thread mode.
→ Here the main program and the exception handler can have separate stack memory regions.
→ It can prevent a stack error in a user application from damaging the stack used by the OS.
→ It is assumed that the user application runs only in thread mode and the OS kernel executes in handler mode.

Cortex-M3 Stack implementation.

→ The Cortex-M3 uses full-descending stack operation model

→ The SP points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation

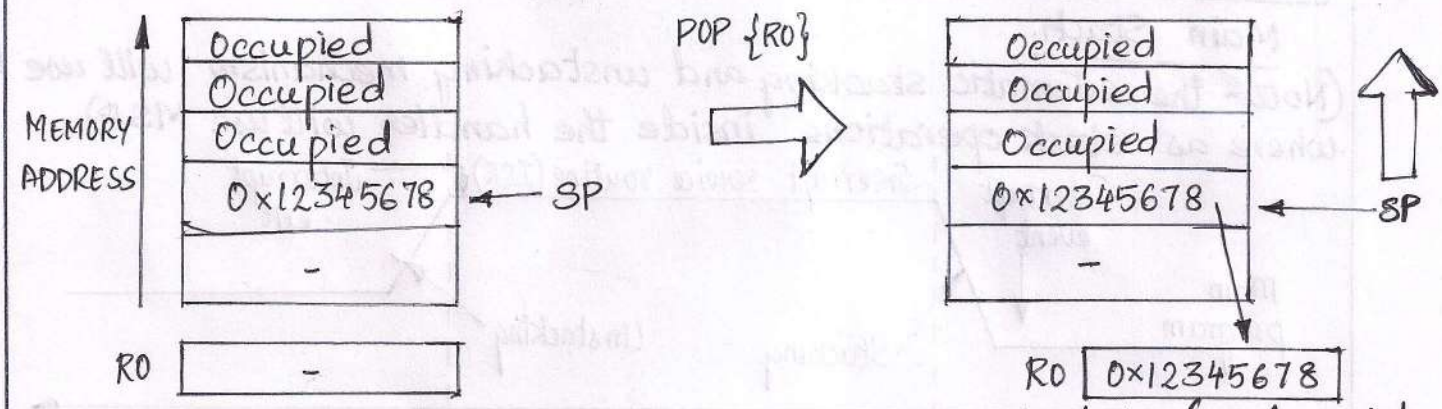
Cortex-M3 Stack PUSH implementation - shows example execution of the instruction - PUSH {R0}.



→ For POP operations, the data is read from the memory location pointer by SP, and then, then SP is incremented.

→ The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place.

Cortex-M3 Stack POP implementation - example - POP {R0}

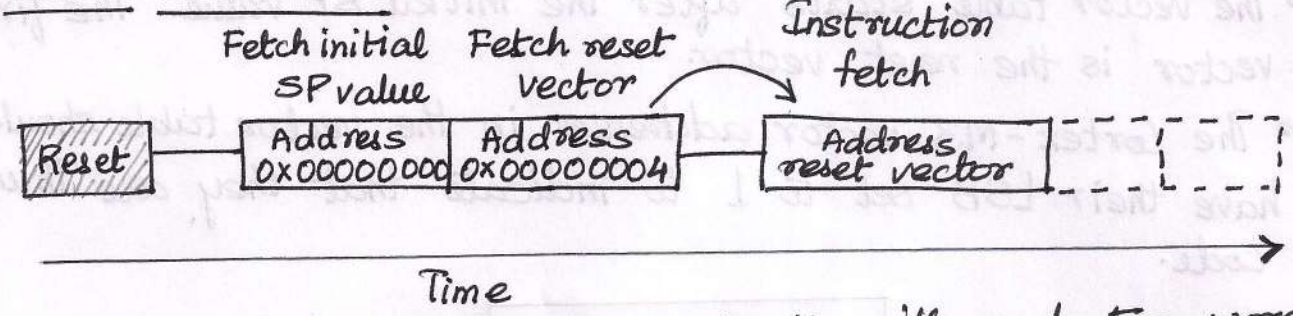


→ each PUSH/POP operation transfers 4 bytes of data (each register contains 1 word, or 4 bytes), the SP decrements/increments by 4 at a time or a multiple of 4 if more than 1 register is pushed or popped.

→ In the Cortex-M3, R13 is defined as the SP. When an interrupt takes place, a no. of registers will be pushed automatically, & R13 will used as the SP for this stacking process.

→ Similarly, the pushed regs will be restored/popped automatically when exiting an interrupt handler, & the SP will also be adjusted.

RESET SEQUENCE



After the processor exits reset, it will read two words from memory.

→ Address 0x00000000 : starting value of R13 (the SP)

→ Address 0x00000004 : Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state).

→ It differs from Traditional ARM processor behavior. Previous ARM processors executed program code starting from address 0x0. as in vector table in previous ARM devices.

→ In the Cortex-M3, the initial value for the MSP is put at the beginning of the memory map, followed by the vector table, which contains vector address values.

→ The vector table can be relocated to another location later, during program execution.

→ In addition, the contents of the vector table are address values not branch instructions.

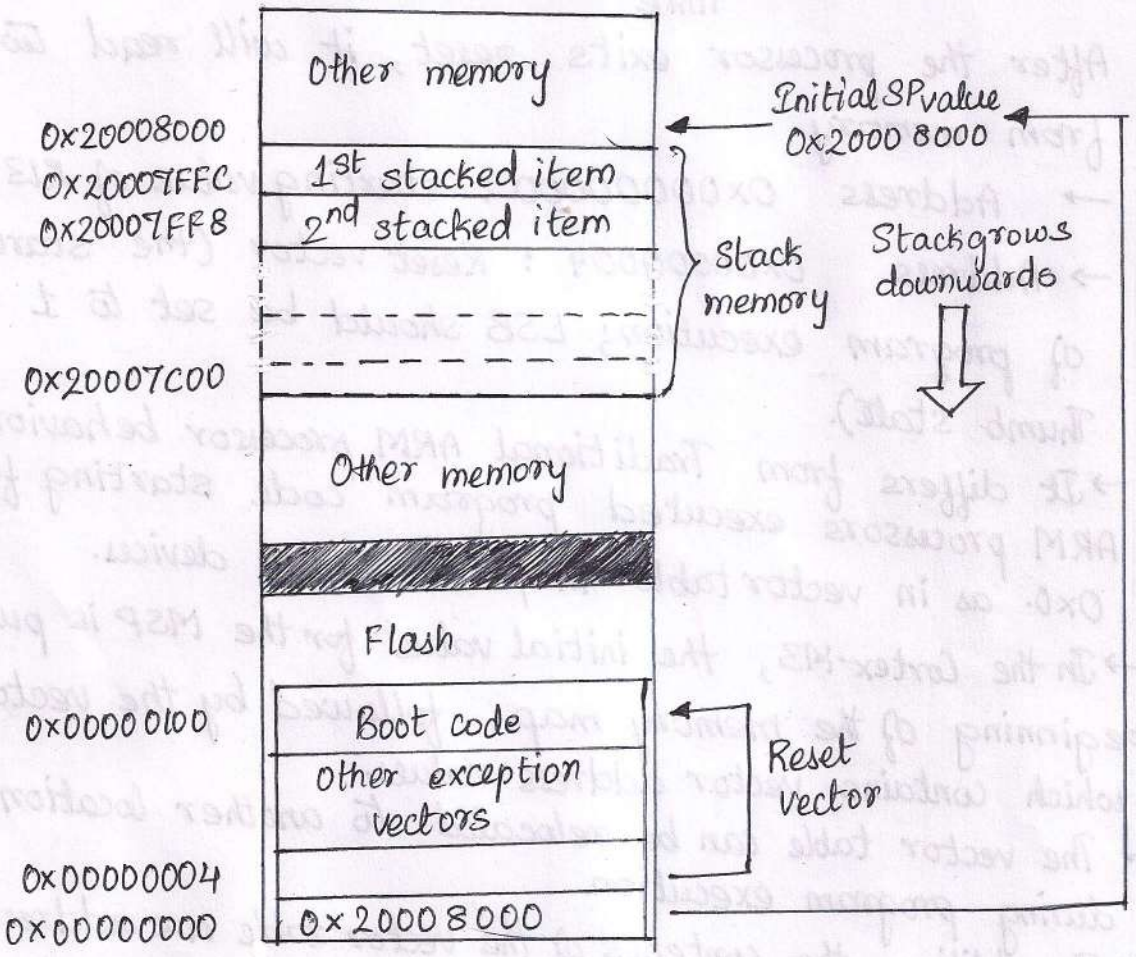
→ The first vector in the vector table (exception type I) is the reset vector, which is the second piece of data fetched by the processor after reset.

→ The stack operation in the Cortex M3 is a full descending stack (SP decrement before store), the initial SP value should be set to the first memory after the top of the stack region.

→ For example, if you have a stack memory range from 0x20007C00 to ~~the~~ 0x20007FFF (1KB), the initial stack value should be set to 0x20008000.

→ The vector table starts after the initial SP value. The first vector is the reset vector.

→ The Cortex-M3 vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code.



Initial Stack pointer Value and Initial Program Counter Value Example

→ For that reason, the previous example has 0x101 in the reset vector, where as the boot code starts at address 0x100.

→ After the reset vector is fetched, the Cortex-M3 can then start to execute the program from the reset vector address and begin normal operations.

→ It is necessary to have the SP initialized, because some of the exceptions (such as NMI) can happen right after reset, and the stack memory could be required for the handler of those exceptions.

MODULE-2 ARM CORTEX M3 INSTRUCTION SETS AND PROGRAMMING

B.S.Balaji
Asst. Prof
BGSIT.

ARM BASICS

Assembler language: Basic Syntax

```
label
        opcode  operand1, operand2, .... ; Comments
```

→ The label is optional. Some of the instructions might have a label in front of instruction so that the address of the instructions can be determined using the label.

→ The opcode (the instruction) followed by a number of operands, where the first operand is the destination of the operation.

→ The no. of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.

For example -

1) immediate data are usually in the form #number,

```
MOV  R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
MOV  R1, # 'A' ; Set R1 = ASCII character A
```

The text after each semicolon (;) is a comment. Comments make programs easier for humans to understand and do not affect the program operation.

2) Define constants using EQU. (or) EQU - defines constants.

```
NVIC_IRQ_SETENO   EQU   0xE000E100
NVIC_IRQ0_ENABLE  EQU   0x1
....
LDR  R0, =NVIC_IRQ_SETENO; ; LDR here is pseudo instruction
                           ; that convert to a PC
                           ; relative load by assembler.

MOV R1, #NVIC_IRQ0_ENABLE  ; Move immediate data
                           ; to register

STR  R1, [R0]              ; Enable IRQ 0 by writing
                           ; R1 to address in R0
```

3) A no. of data definition directives are available for insertion of constants inside assembly code.

For example,

(i) DCI (Define Constant Instruction) can be used to code an instruction if your assembler cannot generate the exact instruction that you want and if you know the binary code for the instruction.

ex-
```
DCI 0xBE00 ; Breakpoint (BKPT 0), a 16-bit instruction
```

(ii) DCB (Define Constant Bytes) for byte size constant values, such as characters.

ex-
```
HELLO_TXT
        DCB "Hello\n", 0  ; null terminated string
```

(iii) DCD (Define Constant Data) for word size constant values to define binary data in your code.

ex-
```
MY_NUMBER
        DCD 0x12345678
```

example code -
```
LDR R3, =MY_NUMBER  ; Get the memory address value of MY_NUMB-ER
LDR R4, [R3]        ; Get the value code 0x12345678 in R4
....
LDR R0, =HELLO_TXT  ; Get the starting memory address of
                    ; HELLO_TXT
BL PrintText        ; Call a function called PrintText
                    ; display string
```

Instruction List - The instructions supported by the Cortex-M3 processor can be categorized as -

1) Memory access instructions
2) General data processing instructions
3) Multiply and divide instructions
4) Saturating instructions
5) Bit field instructions
6) Branch and control instructions.
7) Miscellaneous instructions.

1) **Memory Access Instructions**

Mnemonic	Description
ADR	Load PC-relative address
CLREX	Clear exclusive
LDM (mode)	Load multiple registers
LDR (type)	Load register using immediate offset
LDR (type)	Load register using register offset
LDR (type)T	Load register with unprivileged access
LDR (type)	Load register using PC-relative address.
LDRD	Load register using PC-relative address (two words)
LDREX (type)	Load register exclusive
POP	Pop registers from stack
PUSH	Push registers onto stack.
STM (mode)	Store multiple registers.
STR (type)	Store register using immediate offset
STR (type)	Store register using register offset
STR (type) T	Store register with unprivileged access
STREX (type)	Store register exclusive.

General Data Processing Instructions

Mnemonic	Description
ADC	Add with carry
ADD	Add
ADDW	Add
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the complement of another value).
CLZ	Count leading zeros.
CMN	Compare negative (compare one data with two's complement of another data).
CMP	Compare
EOR	Exclusive OR
LSL	Logical Shift Left
LSR	Logical Shift Right
MOV	Move
MOVT	Move top
MOVW	Move 16-bit constant
MVN	Move NOT (copy logical inverted value)
ORN	Logical OR NOT
ORR	Logical OR
RBIT	Reverse bits
REV	Reverse byte order in a word
REV16	Reverse byte in each halfword
REVSH	Reverse byte order in bottom halfword and sign extend
ROR	Rotate right
RRX	Rotate right with extend
RSB	Reverse subtract
SBC	Subtract with Carry
SUB	Subtract
TEQ	Test equivalence (use as EX-OR, Z flag is updated but result is not stored).
TST	Test (use as logical AND; Z flag is updated but AND result is not stored).

Multiply and Divide Instructions

Mnemonic	Description
MLA	Multiply with accumulate, 32-bit result
MLS	Multiply and Subtract, 32-bit result
MUL	Multiply, 32-bit result
SDIV	Signed divide
SMLAL	Signed multiply with accumulate (32x32 +64), 64 bit result.
SMULL	Signed multiply (32x32), 64 bit result
UDIV	Unsigned divide
UMLAL	Unsigned multiply with accumulate (32x32+64), 64 bit result
UMULL	Unsigned multiply (32x32), 64 bit result.

Saturating Instructions

Mnemonic	Description
SSAT	Signed saturate
USAT	Unsigned saturate.

Bitfield Instructions

Mnemonic	Description
BFC	Bit Field Clear
BFI	Bit Field Insert
SBFX	Signed bit field extract
SXTB	Sign extend a byte
SXTH	Sign extend a halfword
UBFX	Unsigned bit field extract
UXTB	Unsigned extend a byte
UXTH	Unsigned extend a halfword

Branch and Control Instructions

Mnemonic	Description
B	Branch
BL	Branch with link
BLX	Branch indirect with link
BX	Branch indirect
CBNZ	Compare and branch if non-zero
CBZ	Compare and branch if zero
IT	If-Then
TBB	Table branch byte
TBH	Table branch halfword

Miscellaneous Instructions

Mnemonic	Description
BKPT	Breakpoint
CPSID	Change processor state, disable interrupts
CPSIE	Change processor state, enable interrupts
DMB	Data memory barrier
DSB	Data synchronization barrier
ISB	Instruction Synchronization barrier.
MRS	Move from special register to register
MSR	Move from register to special register
NOP	No operation
SEV	Send event
SVC	Supervisor Call
WFE	Wait For Event
WFI	Wait For Interrupt.

Unsupported Instructions (Thumb instructions for Traditional ARM Processors).

BLX Label – ~~This~~ Branch with link and exchange state. (Ina format with immediate data, BLX always changes to ARM state. ∴ Cortex M3 does not support the ARM state, instructions like this one that attempt to switch to the ARM state will result in a fault exception called

SETEND – Thumb instruction, introduced in architecture V6, switches the endian configuration during run time. Since Cortex-M3 does not support dynamic endian using the SETEND instruction will result in a fault exception.

Unsupported Coprocessor Instructions

Instruction	Description
MCR	Move to coprocessor from ARM processor
MCR2	Move to coprocessor from ARM processor
MCRR	Move to coprocessor from two ARM register.
MRC	Move to ARM register from coprocessor.
MRC2	Move to ARM register from coprocessor.
MRRC	Move to two ARM register from coprocessor
LDC	Load coprocessor; load memory data from a sequence of consecutive memory addresses to a coprocessor.
STC	Store coprocessor; stores data from a coprocessor to a sequence of consecutive memory addresses.

Unsupported Change Process State Instructions

Instruction	Description
CPS<IE|ID>.WA	There is no A bit in the Cortex-M3
CPS.W #mode	There is no mode bit in the Cortex-M3 PSR.

Unsupported Hint Instructions

Instruction	Description
DBG	A Hint instruction to debug and trace system.
PLD	Preload data; It is a hint instruction for cache memory, however, since there is no cache in the Cortex-M3 processor, it behaves as NOP.
PLI	Preload instruction; It is a hint instruction for cache memory, however, since there is no cache in the Cortex-M3 processor, it behaves as NOP.
YIELD	A hint instruction to allow multithreading software to indicate to hardware that it is doing a task that can be swapped out to improve overall system performance.

4/5 Instruction Descriptions

① Moving Data - The basic functions in a processor is transfer of data. In Cortex-M3, The types of data transfers are -

(i) Moving data between register and register
(ii) Moving data between memory and register
(iii) Moving data between Special register and register
(iv) Moving an immediate data value into a register.

→ MOV - is the command to move data between registers. Example- moving data from register R3 to register R8.

MOV R8, R3.

→ MVN - is the instruction which can generate the negative value of the original data.

→ Basic instructions for accessing memory are Load and Store.

* Load (LDR) transfers data from memory to registers, and
* Store (STR) transfers data from registers to memory.

The transfers can be in different data sizes (byte, half word, word and double word).

* Multiple Load and Store operations can be combined into single instructions - LDM (Load Multiple) and STM (Store Multiple).

→ The exclamation mark (!) in the instruction specifies whether the register Rd should be updated after the instruction is completed.

→ ARM processors also support memory accesses with pre indexing and post indexing.

→ For example - In post indexing, if R8 equals 0x8000:

*STMIA.W R8!, [R0-R3]; R8 changed to 0x8010 after store (increment by 4 words).

*without exclamation mark (!)
STMIA.W R8, [R0-R3]; R8 unchanged after store.

→ For preindexing, the register holding the memory address is adjusted. The memory transfer then takes place with the updated address.

For example,

```
LDR.W   R0, [R1, #offset]! ; Read memory [R1+offset], with R1
                           ; update to R1+offset.
```

→ The use of the "!" indicates the update of base register R1. It is optional; without it, the instruction would be just a normal memory transfer with offset from a base address.

→ Post indexing memory access instructions carry out the memory transfer using the base address specified by the register and then update the address register afterward.

For example,

```
LDR.W   R0, [R1], #offset ; Read memory [R1], with R1
                          ; updated to R1 + offset
```

Here, all post indexing instructions update the base address register without "!" but in pre-indexing instructions, (programmer might choose) whether to update the base address register or not.

Simple Memory Access Instructions

Example	Description
LDRB Rd, [Rn, #offset]	Read byte from memory location Rn+offset
LDRH Rd, [Rn, #offset]	Read half word memory location Rn+offset
LDR Rd, [Rn, #offset]	Read word from memory location Rn+offset
LDRD Rd1, Rd2, [Rn, #offset]	Read double word from memory location Rn+offset
STRB Rd, [Rn, #offset]	Store byte to memory location Rn+offset
STRH Rd, [Rn, #offset]	Store half word to memory location Rn+offset
STR Rd, [Rn, #offset]	Store word to memory location Rn+offset
STRD Rd1, Rd2, [Rn, #offset]	Store double word to memory location Rn+offset

Multiple Memory Access Instructions

Example	Description
LDMIA Rd!, <reg list>	Read multiple words from memory location specified by Rd; address increment after (IA) each transfer (16-bit Thumb instruction).
STMIA Rd! <reg list>	Store multiple words to memory location specified by Rd; address increment after (IA) each transfer (16-bit Thumb instruction).
LDMIA.W Rd(!), <reg list>	Read multiple words from memory location specified by Rd; address increment after each read. (.W specified it is a 32-bit Thumb2 instruction).
LDMDB.W Rd(!), <reg list>	Read multiple words from memory location specified by Rd; address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb2 instruction).
STMIA.W Rd(!), <reg list>	Write multiple words to memory location specified by Rd; address increment after each read. (.W specified it is a 32-bit Thumb-2 Instruction)
STMDB.W Rd(!), <reg list>	Write multiple words to memory location specified by Rd; address DB each read (.W specified it is a 32-bit Thumb-2 Instruction).

Preindexing Memory Access Instructions

Example	Description
LDR.W Rd, [Rn, #offset]! LDRB.W Rd, [Rn, #offset]! LDRH.W Rd, [Rn, #offset]! LDRD.W Rd1, Rd2, [Rn, #offset]!	Preindexing load instructions for various sizes (word, byte, half word and double word
LDRSB.W Rd, [Rn, #offset]! LDRSH.W Rd, [Rn, #offset]!	Preindexing load instructions for various sizes with sign extend (byte, half word).

```
STR.W  Rd, [Rn, #offset]!
STRB.W Rd, [Rn, #offset]!
STRH.W Rd, [Rn, #offset]!
STRD.W Rd1, Rd2, [Rn, #offset]!
```

} Pre indexing store instructions for various sizes (word, byte, half word and double word)

Post indexing Memory Access Instructions – Examples

```
LDR.W  Rd, [Rn], #offset
LDRB.W Rd, [Rn], #offset
LDRH.W Rd, [Rn], #offset
LDRD.W Rd1, Rd2, [Rn], #offset
```

} Post indexing load instructions for various sizes (word, byte, half word, and double word)

```
LDRSB.W Rd, [Rn], #offset
LDRSH.W Rd, [Rn], #offset
```

} Post indexing load instructions for various sizes with sign extend (byte, half word).

```
STR.W  Rd, [Rn], #offset
STRB.W Rd, [Rn], #offset
STRH.W Rd, [Rn], #offset
STRD.W Rd1, Rd2, [Rn], #offset
```

} Post indexing store instructions for various sizes (word, byte, half word, and double word).

→ Two other types of memory operations are stack PUSH and stack POP.

PUSH and POP. – example

```
PUSH   [R0-R3, LR] ; Save register contents at beginning of
                   ; Subroutine
                   ; Processing
....
POP    [R2, R3]    ; Pop R2 and R3 from stack.
```

→ Both PUSH instruction and POP instruction corresponds to same register list but it is mandatory. For example, a common exception is when POP is used as a function return:

```
PUSH [R0-R3, LR] ; Save register contents at beginning of
                 ; Subroutine
                 ; Processing
....
POP  [R0-R3, PC] ; Restore registers and return.
```

Instead of popping the LR register back and then branching to the address in LR, where as the POP instructions provides the address value directly in the program counter.

Immediate Data Transfer

→ MOV- Moving immediate data into a register. example- MOV R0, #0x789A

* For small values (8 bits or less), - MOVS instruction is used.

ex - MOVS R0, #0x12 ; Set R0 to 0x12.

* For a larger value (over 8 bits), Thumb-2 move instruction is used.

ex- ~~MOV.W~~ MOVW.W R0, #0x789A; Set R0 to 0x789A.

* If the value is 32-bit, Two instructions are used to set the upper and lower halves.

```
MOVW.W  R0, #0x789A ; Set R0 lower half to 0x789A
MOVT.W  R0, #0x3456 ; Set R0 upper half to 0x3456.
```

Now R0 = 0x3456789A.

(or)

A pseudo-instruction provided in ARM assembler - LDR, example.

```
LDR R0, =0x3456789A
```

where it is used to set registers to a program address value.

② LDR and ADR Pseudo-instructions -

LDR and ADR are pseudo instructions which can be used to set registers to a program address value. They have different syntaxes and behaviours.

For example- ① if the address is a program address value, the assembler will automatically set the LSB to 1. which indicates it is thumb code.

```
    LDR R0, =address 1; R0 set to 0x4001
    ...
address 1               ; address here is 0x4000
    MOV R0, R1          ; address 1 contains program code
    ...
```

(2) If address1 is a data address, LSB will not be changed.

```
LDR  R0, =address1 ; R0 set to 0x4000
...
address 1          ; address here is 0x4000
DCD  0x0           ; address1 contains data
                                    loaded
...
```

For ADR, - the address value of a program code into a register without setting the LSB automatically.

Note ~~that~~ - there is no equal sign (=) in the ADR statement.

Example -

```
ADR  R0, address1
...
address1           ; (address here is 0x4000)
  MOV  R0, R1      ; address 1 contains program code
...
```

→ LDR obtains the immediate data by putting the data in the program code and uses a PC relative load to get the data into the register.

→ ADR tries to generate the immediate value by adding or subtracting instructions based on current PC value.

→ Using ADR, it requires target address must be in a close range as it is not possible to create all immediate values.

→ ADR can generate smaller code sizes compared with LDR.

→ The 16 bit version of ADR requires that the target address must be word aligned (address value is a multiple of 4).

(3) Processing Data Instructions -

Cortex M3 provides many different instructions for data processing. Data operation instructions can have multiple instruction formats.

Example - an ADD instruction can operate between two registers or between one register and an immediate data value:

```
ADD   R0, R0, R1 ;    R0 = R0 + R1;
ADDS  R0, R0,#0x12;   R0 = R0 + 0x12
ADD.W R0, R1, R2 ;    R0 = R1 + R2;
```

→ With Traditional Thumb instruction syntax, when 16-bit Thumb code is used, an ADD instruction can ~~operate~~ ~~instruction~~ change the flags in the PSR.

→ 32-bit Thumb-2 code can either change a ~~Keep~~ flag or keep it unchanged. The 'S' suffix should be used where the operation depends on the flags-

Example-
```
ADD.W   R0, R1, R2 ; Flag unchanged
ADDS.W  R0, R1, R2 ; Flag change.
```

→ Cortex M3 supports more arithmetic functions include subtract (SUB), multiply (MUL), and unsigned and signed divide (UDIV/SDIV).

Examples of Arithmetic Instructions.

Instruction		Operation
ADD Rd, Rn, Rm ;	Rd = Rn + Rm	ADD operation
ADD Rd, Rd, Rm ;	Rd = Rd + Rm	
ADD Rd, #immed;	Rd = Rd + #immed	
ADD Rd, Rn, #immed;	Rd = Rn + #immed	
ADC Rd, Rn, Rm ;	Rd = Rn + Rm + carry	ADD with carry
ADC Rd, Rd, Rm;	Rd = Rd + Rm + carry	
ADC Rd, #immed ;	Rd = Rd + #immed + carry	
ADDW Rd, Rn, #immed;	Rd = Rn + #immed	ADD register with 12 bit immediate value.
SUB Rd, Rn, Rm;	Rd = Rn - Rm	Subtract
SUB Rd, #immed ;	Rd = Rd - #immed	
SUB Rd, Rn, #immed;	Rd = Rn - #immed	
SBC Rd, Rm ;	Rd = Rd - Rm - borrow	Subtract with borrow (not carry)
SBC.W Rd, Rn, #immed;	Rd = Rn - #immed - borrow	
SBC.W Rd, Rn, Rm ;	Rd = Rn - Rm - borrow	
RSB.W Rd, Rn, #immed;	Rd = #immed - Rn	Reverse subtract
RSB.W Rd, Rn, Rm ;	Rd = Rm - Rn	
MUL Rd, Rm ;	Rd = Rd * Rm	Multiply
MUL.W Rd, Rn, Rm ;	Rd = Rn * Rm	
UDIV Rd, Rn, Rm;	Rd = Rn / Rm	Unsigned and Signed divide.
SDIV Rd, Rn, Rm;	Rd = Rn / Rm	

Note: with or without

→ 16 bit Thumb instructions used "S" suffix to determine if APSR should be updated

→ In Unified Assembly Language (UAL) syntax, 32 bit instructions and 'S' suffix is not used then it will be selected as 16 bit Thumb instructions, update APSR.

→ The Cortex M3 also supports 32-bit multiply instructions and multiply accumulate instructions that gives 64 bit results.
It supports both signed or unsigned values.

32-Bit Multiply Instructions

Instruction	Operation
SMULL RdLo, RdHi, Rn, Rm;	{RdHi, RdLo} = Rn * Rm } 32 bit multiply instructions for signed values
SMLAL RdLo, RdHi, Rn, Rm;	{RdHi, RdLo} += Rn * Rm }

→ Another group of data processing instructions are the logical operations instructions and logical operations such as AND, ORR (or), and shift and rotate functions.

Logic Operations Instructions

Instruction		Operation
AND Rd, Rn	; Rd = Rd & Rn	} Bitwise AND
AND.W Rd, Rn, #immed	; Rd = Rn & #immed	
AND.W Rd, Rn, Rm	; Rd = Rn & Rd	
ORR Rd, Rn	; Rd = Rd | Rn	} Bitwise OR
ORR.W Rd, Rn, #immed	; Rd = Rn | #immed	
ORR.W Rd, Rn, Rm	; Rd = Rn | Rm	
BIC Rd, Rn	; Rd = Rd & (~Rn)	} Bit clear
BIC.W Rd, Rn, #immed	; Rd = Rn & (~#immed)	
BIC.W Rd, Rn, Rm	; Rd = Rn & (~Rd)	
ORN.W Rd, Rn, #immed	; Rd = Rn | (~#immed)	} Bitwise OR NOT
ORN.W Rd, Rn, Rm	; Rd = Rn | (~Rm)	
EOR Rd, Rn	; Rd = Rd ^ Rn	} Bitwise Exclusive OR
EOR.W Rd, Rn, #immed	; Rd = Rn ^ #immed	
EOR.W Rd, Rn, Rm	; Rd = Rn ^ Rm	

Shift and Rotate Instructions

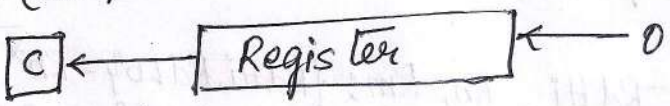
The Cortex-M3 provides rotate and shift instructions.

Note: Why is there rotate right but no rotate left?

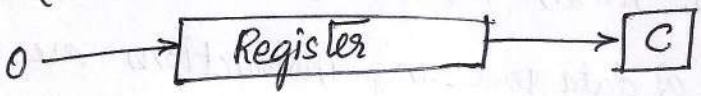
→ The rotate left operation can be replaced by a rotate right operation with a different rotate offset.

→ Example - a rotate left by 4 bit operation can be written as a rotate right by 28 bit instruction, which it takes equal amount of time to execute and provides the same result.

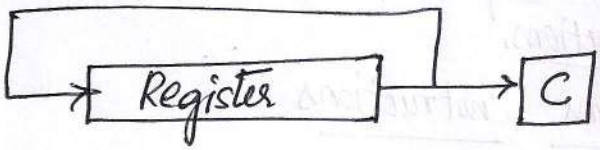
Logical Shift Left (LSL)



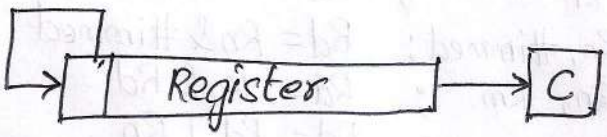
Logical Shift Right (LSR)



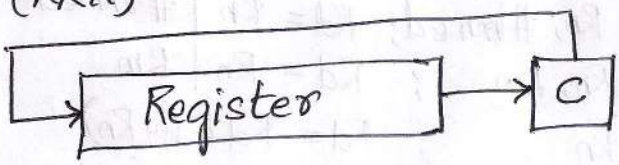
Rotate Right (ROR)



Arithmetic Shift Right (ASR)



Rotate Right eXtended (RRX)



→ In UAL Syntax, if the 'S' suffix is used then the Shift and Rotate operations can also update the carry flag, and

→ If the 16 bit Thumb® code is used, it always update the carry flag.

→ If the shift or rotate operation shifts the register position by multiple bits, the value of the carry flag "C" will be the last bit that shifts out of the register.

Shift and Rotate Instructions

Instruction		Operation
ASR Rd, Rn, #immed;	Rd = Rn >> immed	Arithmetic Shift Right
ASR Rd, Rn ;	Rd = Rd >> Rn	
ASR.W Rd, Rn, Rm ;	Rd = Rn >> Rm	
LSL Rd, Rn, #immed;	Rd = Rn << immed	Logical Shift left
LSL Rd, Rn ;	Rd = Rd << Rn	
LSL.W Rd, Rn, Rm ;	Rd = Rn << Rm	
LSR Rd, Rn;	Rd = Rd >> Rn	Logical Shift right
LSR Rd, Rn, #immed;	Rd = Rn >> immed	
LSR.W Rd, Rn, Rm ;	Rd = Rn >> Rm	
ROR Rd, Rn ;	Rd rot by Rn	Rotate right
ROR.W Rd, Rn, #immed;	Rd = Rn rot by immed	
ROR.W Rd, Rn, Rm ;	Rd = Rn rot by Rm	
RRX.W Rd, Rn ;	{C, Rd} = {Rn, C}	Rotate right extended.

Sign Extend Instructions

The Cortex M3 provides the two instructions for conversion of signed data from byte or half word to word where 16 bit version can only access low registers and 32 bit version can access both registers (top & low).

Sign Extend Instruction

Instruction		Operation.
SXTB Rd, Rm;	signext (Rm[7:0])	Sign extend byte data into word
SXTH Rd, Rm;	signext (Rm[15:0])	Sign extend half word data into word.

Data reverse ordering Instructions – It is another group of data processing instructions is used for reversing data types in a register.

It is usually used for conversation between little endian and big endian data. (where 16 bit version can only access low registers and 32 bit version can access both registers (top and low)).

Data Reverse Ordering Instructions

		Operation
REV	Rd, Rn ; Rd = rev (Rn)	Reverse bytes in word
REV16	Rd, Rn ; Rd = rev16 (Rn)	Reverse bytes in each half word
REVSH	Rd, Rn ; Rd = revsh (Rn)	Reverse bytes in bottom half word and sign extend the result.

Bit Field Processing and Manipulation Instructions

It is a group of data processing instructions is used for ~~reversing data types~~ bit field processing and manipulation instructions.

Instruction	Operation
BFC.W Rd, Rn, #<width>	Clear bit field within a register
BFI.W Rd, Rn, #<lsb>, #<width>	Insert bit field to a register
CLZ.W Rd, Rn	Count leading zero
RBIT.W Rd, Rn	Reverse bit order in register
SBFX.W Rd, Rn, #<lsb>, #<width>	Copy bit field from source and sign extend it.
UBFX.W Rd, Rn, #<lsb>, #<width>	Copy bit field from source register.

(4) Call and Unconditional Branch. – Basic branch instructions are-

B label – Branch to a labeled address
BX reg – Branch to an address specified by a register.

In BX instructions, the LSB value contained in the register determines the next state (Thumb/ARM) of the processor. (If LSB is set to 1, it is always in Thumb state. If it is zero, it will try to switch the processor into ARM state because it will cause a usage fault exception.)

Branch and Link instruction

BL label ; Branch to a labeled address and save return
; address in LR.

~~With these instructions~~, the return address will be stored in the link register (LR) and the function can be terminated using BX LR, which causes program control to return to the calling process.

```
BLX reg  ;  Branch to an address specified by a register and
         ;  save return
         ;  address in LR
```

Here make sure that the LSB of the register is 1. Otherwise the processor will produce a fault exception because it is an attempt to switch to the ARM state.

Branch operation using MOV instructions and LDR instructions - Example

```
MOV   R15, R0 ;  Branch to an address inside R0
LDR   R15, [R0] ; Branch to an address in memory location
                ; specified by R0
POP   [R15]     ; Do a stack pop operation, and change the program
                ; counter value to the result value.
```

Note: Save the LR If you need to call a subroutine.

→ The BL instruction will destroy the current content of your LR whereas if the program code needs the LR later, then it should be saved before using BL.

→ The common method is to push the LR to stack in the beginning of your subroutine.

Example - main

```
...
BL functionA
...
functionA
    PUSH {LR} ; Save LR content to stack
    ...
    BL functionB
    ...
    POP {PC} ; Use stacked LR content to return to main
functionB
    PUSH {LR}
    ...
    POP {PC}; Use stacked LR content to return to functionA
```

→ If the subroutine is a C function, the contents in R0-R3 and R12 need to be saved, if it is required at a later stage.
→ The contents in these registers could be changed by a C function.

(5) Decisions and Conditional Branches

→ ARM processors use flags in the APSR to determine whether a branch should be carried out for conditional branches
→ In APSR, there are five flag bits; four of them are used for branch decisions.

Flag Bits in APSR that can be used for Conditional Branches.

Flag	PSR Bit	Description
N	31	Negative flag (last operation result is a negative value).
Z	30	Zero (last operation result returns a zero value)
C	29	Carry (last operation returns a carry out or borrow)
V	28	Overflow (last operation results in an overflow).
Q	27	Saturation math operations (non Conditional branches).

Q flag is another flag bit [27] is used for saturation math operations and is not used for conditional branches.

Z (Zero) flag - It is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result. (bit 31 is 1)

N (Negative) flag - It is set when the result of an instruction has a negative value.

C (Carry) flag - It is for unsigned data processing - for ex- in add, it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).

V(overflow) ~~bits~~ flag - It is for signed data processing; for example - in an add, when two positive values added together produce a negative ~~value~~, ~~or~~ value, or when two negative values added together produce a positive value.

15 branch conditions are defined with combinations of the four flags (N, Z, C, and V)

Example - BEQ label ; Branch to address 'label' if Z flag is set

Thumb-2 version → BEQ.W label ; Branch to address 'label' if Z flag is set

Conditions for branches or other Conditional Operations

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z Clear
CS/HS	Carry set / Unsigned higher or same	C Set
CC/LO	Carry clear / Unsigned lower	C clear
MI	Minus / Negative	N set
PL	Plus / positive or zero	N Clear
VS	Overflow	V Set
VC	No overflow	V Clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
LT	Signed less than	N set and V clear, or N clear & V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z==0, N==V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z==1 or N!=V)
AL	Always (unconditional)	—

Branch conditions can also be used in IF-THEN-ELSE structures.

```
Ex-  CMP  R0, R1 ;   Compare R0 and R1
     ITTEE GT    ;   if R0 > R1 then
                 ;   if true, first 2 statements execute,
                 ;   if false, other 2 statements execute
     MOVGT R2, R0 ;     R2 = R0
     MOVGT R3, R1 ;     R3 = R1
     MOVLE R2, R0 ; Else R2 = R1
     MOVLE R3, R1 ;     R3 = R0
```

APSR can be affected by the following -

(i) Most of the 16-bit ALU instructions
(ii) 32 bit (Thumb)-2) ALU instructions with the S suffix ; ex- ADDS.W
(iii) Compare (ex- CMP) and Test (ex- TST, TEQ)
(iv) Write to APSR/xPSR directly.

16 bit Thumb instructions (arithmetic) affect the N, Z, C, and V flags.

In 32 bit Thumb-2 instructions, the ALU operation can either change flags or not change flags.

Example -

```
ADDS.W R0, R1, R2 ; This 32 bit Thumb instruction updates flag
ADD.W  R0, R1, R2 ; This 32 bit Thumb instruction does not update flag
```

In Thumb syntax - example "CODE 16" directive is used with ARM assembler.

```
ADD R0, R1 ;    This 16 bit Thumb instruction updates flag
ADD R0, #0x1 ;  This 16 bit Thumb instruction updates flag
```

In UAL syntax -

```
ADD R0, R1 ;    This 16-bit Thumb instruction does not update flag
ADD R0, #0x1 ;  This will become a 32-bit Thumb instruction that
                does not update flag.
```

Compare (CMP) instruction - it subtracts two values and updates the flags (just like SUBS), but the result is not stored in any registers.

example- CMP R0, R1 ; Calculate R0 - R1 and update flag
CMP R0, #0x12 ; Calculate R0 - (-0x12) and update flag

Compare negative (CMN) instruction- it compares one value to the negative (two's complement) of a second value; the flags are updated, but the result is not stored in any registers.

example- CMN R0, R1 ; Calculate R0 - (-R1) and update flag
CMN R0, #0x12 ; Calculate R0 - (-0x12) and update flag.

TST (test) instruction - It is more like the AND instruction. It ANDs two values and updates the flags and the result is not stored in any register.

example- TST R0, R1 ; Calculate R0 AND R1 and update flag
TST R0, #0x12 ; Calculate R0 AND 0x12 and update flag

⑥ Combined Compare and Conditional Branch -

Cortex-M3 with ARM architecture v7-M provides two new instructions to compare with zero and conditional branch operations. These are
① CBZ (Compare and branch if zero) and
② CBNZ (Compare and branch if nonzero)

The compare and branch instructions only support forward branches.

① CBZ - example

```
i = 5;
while (i!=0){
func1(); ; call a function
i--;
}
```

This can be compiled into the following

```
    MOV R0, #5; set loop counter
loop1 CBZ R0, loop1exit; if loop counter=0
⇒   BL func1; call a function ; then exit the loop
    SUB R0, #1; loop counter decrement
    B loop1; next loop
loop1 exit
```

CBZ is the fact that the branch is taken if the Z flag is set (if result is zero).

CBNZ - the fact is similar to CBZ, apart from the branch is taken if the Z flag is not set (result is not zero).

Example (In C Programming language)

```
status = strchr(email_address, '@');
if (status == 0) { //status is 0 if @ is not in email_address
    show_error_message();
    exit(1);
    }
```

This can be compiled into the following -

```
    ...
    BL strchr
    CBNZ R0, email_looks_okay; Branch if result is not zero
    BL show_error_message
    BL exit
email_looks_okay
    ...
```

Note - The APSR value is not affected by the CBZ and CBNZ instructions.

Conditional Execution Using IT Instructions.

→ The IT (If-Then) block is very useful for handling small conditional code.
→ It avoids branch penalties because there is no change to program flow.
→ It can provide a maximum of four conditionally executed instructions.
→ The first line must be the IT instruction, detailing the choice of execution, followed by the condition it checks.
→ The first statement after the IT command must be TRUE - THEN - EXECUTE, which is always written as ITxyz, where T means THEN and E means ELSE.
→ The second ~~statement~~ through fourth statements can be either THEN (true) or ELSE (false):-

IT syntax -

```
IT <x> <y> <z> <cond>          ; IT instruction (<x>, <y>, <z>
                               ;   can be T or E)
instr1 <cond> <operands>       ; 1st instruction (<cond> must be
                                 same as IT)
instr2 <cond or not cond> <operands>; 2nd instruction (can be <cond>
                               ;   or <! cond>
instr3 <cond or not cond> <operands>; 3rd instruction (can be <cond>
                               ;   or <! cond>
instr 4 <cond or not cond> <operands>; 4th instruction (can be <cond>
                               ;   or <! cond>
```

→ If a statement is to be executed when <cond> is false, the suffix for the instruction must be the opposite of the condition.
Example - the opposite of EQ is NE, the opposite of GT is LE, and so on.

→ The following code shows an example of a simple conditional execution-

```
if (R1 < R2) then
    R2 = R2 - R1
    R2 = R2/2
else R1 = R1 - R2
     R1 = R1/2
```

In assembly,

```
    CMP   R1, R2   ; If R1 < R2 (less then)
    ITTEE  LT      ; then execute instruction 1 and 2
                   ; (indicated by T)
                   ; else execute instruction 3 and 4
                   ; (indicated by E)
    SUBLT.W R2,R1  ; 1st Instruction
    LSRLT.W R2,#1  ; 2nd Instruction
    SUBGE.W R1,R2  ; 3rd instruction (notice the GE is opposite of LT)
    LSRGE.W R1,#1  ; 4th instruction
```

(7) Instruction Barrier and Memory Barrier Instructions.

→ The Cortex-M3 supports a no. of barrier instructions. ~~These~~ It is very important as memory systems get more and more complex.

→ if memory barrier instructions are not used, race conditions could occur.

Example - if memory map can be switched by a hardware register, after writing to the memory switching register - DSB (Data Synchronization Barrier) instruction is used.

Barrier Instructions - are the three instructions available in Cortex-M3

Instruction	Description
DMB	Data Memory Barrier; ensures that all memory accesses are completed before new memory access is committed.
DSB	Data Synchronization Barrier; ensures that all memory accesses are completed before next instruction is executed.
ISB	Instruction Synchronization Barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions.

The memory barrier instructions can be accessed in C using Cortex Microcontroller Software Interface Standard (CMSIS) compliant device driver library as follows -

```
void __DMB (void);  // Data Memory Barrier
void __DSB (void);  // Data Synchronisation Barrier
void __ISB (void);  // Instruction Synchronization Barrier
```

→ DMB is very useful for multiprocessor systems. For example, tasks running on separate processors might use shared memory to communicate with each other.

→ It can be inserted between accesses to the shared memory to ensure that the memory access sequence is exactly the same as expected.

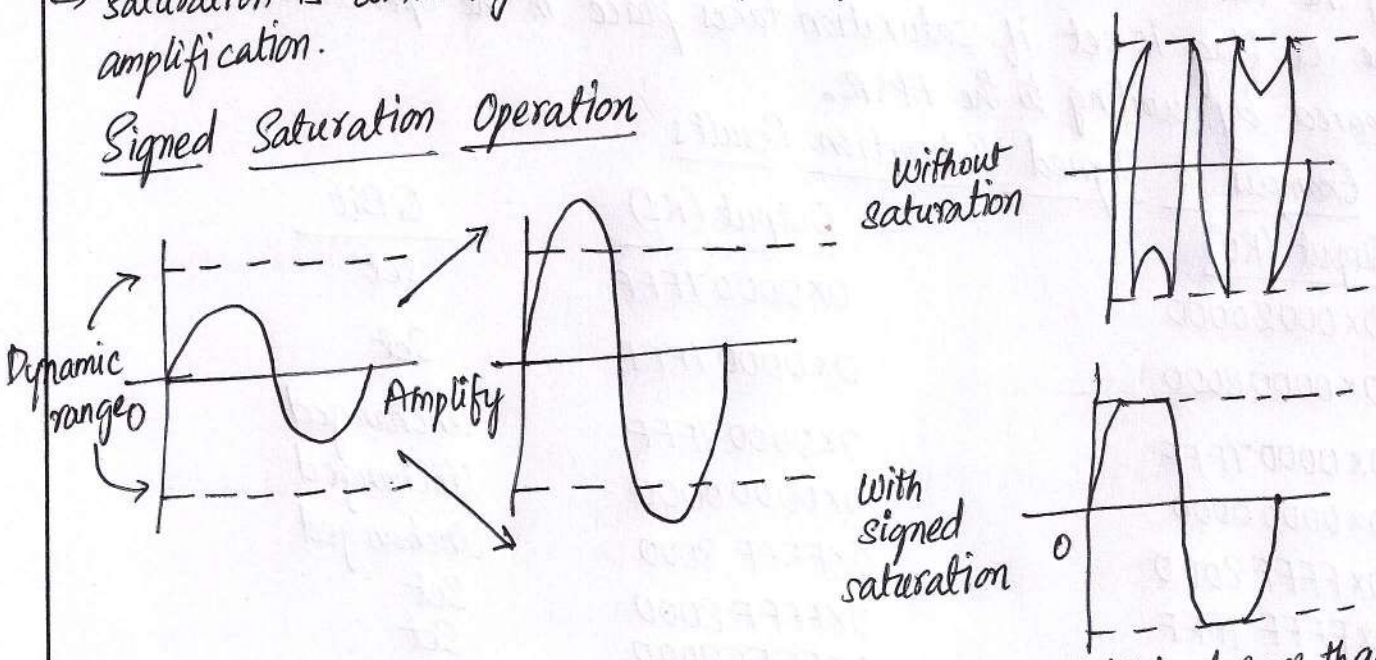
→ The DSB and ISB instructions can be important for self-modifying code. For example, if a program changes its own program code, the next executed should be based on the updated program. Using DSB and then ISB can ensure that the modified program code is fetched again.

(8) Saturation Operations -

→ Cortex M3 supports two instructions that provide signed and unsigned saturation operations - SSAT and USAT (for signed data type and unsigned data type, respectively).

→ Saturation is commonly used in signal processing - example, in signal amplification.

Signed Saturation Operation



→ When an input signal is amplified, where the output will be larger than the allowed output range.

→ If the value is adjusted simply by removing the unused MSB, an overflowed result will cause the signal waveform to be completely deformed as shown in figure.

→ The saturation operation does not prevent the distortion of the signal, but at least the amount of distortion is greatly reduced in the signal waveform.

→ The instruction syntax of the SSAT and USAT instructions

Instruction	Description
SSAT.W <Rd>, #<immed>, <Rn>, {, <shift>}	Saturation for signed value
USAT.W <Rd>, #<immed>, <Rn>, {, <shift>}	Saturation for a signed value into an unsigned value

Note - Rn - Input value, Rd - Destination register

Shift - Shift operation for input value before saturation; optional, can be #LSL N or #ASR N.

Immed - Bit position where the saturation is carried out.

Example - if a 32 bit signed value is to be saturated into a 16-bit signed value -

SSAT.W R1, #16, R0

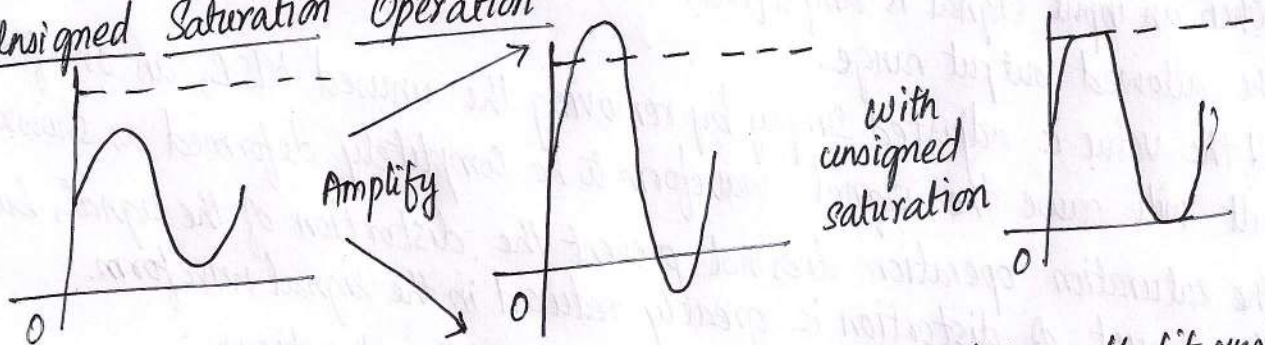
→ Besides the destination register, the Q bit in the APSR can also be affected by the result.

→ The Q flag is set if saturation takes place in the operation, and it can be cleared by writing to the APSR.

Examples of Signed Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x00007FFF	Set
0x00008000	0x00007FFF	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF7FFF	0xFFFF8000	Set
0xFFFE0000	0xFFFF8000	Set

Unsigned Saturation Operation



Example - if a ~~16~~ 32 bit unsigned value is to saturate into a 16-bit unsigned value - USAT.W R1, #16, R0

Examples of Unsigned Saturation results.

INPUT (R0)	OUTPUT (R1)	Q Bit
0x00020000	0x0000FFFF	Set
0x00008000	0x00008000	Unchanged
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0x00000000	Set
0xFFFF8001	0x00000000	Set
0xFFFFFFFF	0x00000000	Set

Several Useful Instructions in the Cortex-M3

1) **MSR and MRS** – are two instructions provide access to special registers.

```
MSR <SReg>, <Rn> ;  Write to Special Register
MRS <Rn>, <SReg> ;  Move from Special Register
```

Example - The code is used to set up the process stack pointer:

```
LDR R0, =0x20008000 ; new value for Process Stack Pointer (PSP)
MSR PSP, R0
```

→ The MRS and MSR instructions can be used in privileged mode only without accessing the APSR. or the operation will be ignored and the returned read data (if MRS is used) will be zero.

→ MSR instruction is used to update the value of the CONTROL register. where ISB instruction is recommended to be added to ensure that the effect of the update takes place immediately.

Example -
```
MRS R0, PSR ;  Read Processor Status word into R0
MSR CONTROL, R1; Write value of R1 into Control register.
```

Special register names for MRS and MSR Instructions.

Symbol	Description
IPSR	Interrupt Status register
EPSR	Execution status register (read as Zero)
APSR	Flags from previous operation.
IEPSR	A composite of IPSR and EPSR
IAPSR	A composite of EPSR and APSR
PSR	A Composite of APSR, EPSR, and IPSR
MSP	Main Stack Pointer
PSP	Process Stack Pointer
PRIMASK	Normal exception mask register
BASEPRI	Normal exception priority mask register
BASEPRI_MAX	Same as normal exception priority mask register, with Conditional write (new level must be higher than the old).
FAULTMASK	Fault exception mask register (disables normal interrupts).
CONTROL	Control register

2) IF-THEN Instruction block

→ It allows up to four succeeding instructions (called an IT block) to be conditionally executed.
→ The <cond> part uses the same condition symbols as conditional branch.
→ Each of <x>, <y> & <z> can be either T (THEN) or E (ELSE), which refers to the base condition <cond> whereas it uses traditional syntax such as EQ, NE, GT or the like.

IT format -

→ <x> specifies the execution condition for the second instruction.
→ <y> specifies the execution condition for the third instruction.
→ <z> specifies the execution condition for the fourth instruction.
→ <cond> specifies the base condition of the instruction block; the first instruction following IT executes if <cond> is true.

Various Length of IT Instruction Block

	IT Block (each of <x>, <y> and <z> can either be T [True] or E [else])	Examples
Only one conditional instruction	IT <cond> instr1 <cond>	IT EQ ADDEQ R0, R0, R1

Two conditional instructions	IT <x> <cond>
	instr1 <cond>
	instr2 <cond or ~(cond)>
		ITE GE
		ADDGE R0, R0, R1
		ADDLT R0, R0, R3

Three conditional instructions	IT <x><y> <cond>
	instr1 <cond>
	instr2 <cond or ~(cond)>
	instr3 <cond or ~(cond)>
		ITET GT
		ADDGT R0, R0, R1
		ADDLE R0, R0, R3
		ADDGT R2, R4, #1

Four conditional instructions	IT <x><y><z> <cond>
	instr1 <cond>
	instr2 <cond or ~(cond)>
	instr3 <cond or ~(cond)>
	instr4 <cond or ~(cond)>
		ITETT NE
		ADDNE R0, R0, R1
		ADDEQ R0, R0, R3
		ADDNE R2, R4, #1
		MOVNE R5, R3

Example -

```
if (R0 equal R1) then {
    R3 = R4 + R5
    R3 = R3/2
} else {
    R3 = R6 + R7
    R3 = R3/2
}
```

```
CMP R0, R1;      ; Compare R0 and R1
ITTEE EQ;        ; If R0 equal R1 Then-Then-Else-Else
⟹ ADDEQ R3, R4, R5; Add if Equal
ASREQ R3, R3, #1; Arithmetic shift right if equal
ADDNE R3, R6, R7; Add if not equal
ASRNE R3, R3, #1; Arithmetic shift right if not equal.
```

Automatic Insertion of IT instruction in ARM assembler

Original Assembly Code

```
CMP   R1, #2
ADDEQ R0, R1, #1
```

Disassembled Assembly code from Generated Object file

```
CMP   R1, #2
IT    EQ
ADDEQ R0, R1, #1
```

→ When ARM assembler is used, and if a conditional execution instruction is used in assembly code without IT instruction,

→ The assembler can insert the required IT instruction automatically.

3) SDIV and UDIV

The syntax for signed and unsigned divide instructions are -

```
SDIV.W <Rd>, <Rn>, <Rm>
UDIV.W <Rd>, <Rn>, <Rm>
```

Result is Rd = Rn/Rm.

example -

```
LDR   R0, = 300 ; Decimal 300
MOV   R1, #5
UDIV.W R2, R0, R1
```

Result is $R2 = R0/R1 = \frac{300}{5} = 60$ (or) (0x3C)

4) REV, REVH, and REVSH -

→ REV reverses the byte order in a data word, and REVH reverses the byte order inside a half word.

Example - R0 = 0x12345678

```
REV   R1, R0 ;    R1 = 0x7856 3412
REVH  R2, R0 ;    R2 = 0x3412 7856
```

→ REV and REVH are particularly useful for converting data between big endian and little endian.

→ REVSH is similar to REVH but it processes the lower half word and then it sign extends the result.

Example - R0 = 0x33448899

```
REVSH  R1, R0 ;   R1 = 0xFFFF9988.
```

5) Reverse Bit (RBIT)

RBIT instruction - reverses the bit order in a data word.

Syntax - RBIT.W <Rd>,<Rn>

→ It is very useful for processing serial bit streams in data communications.

Example - R0 = 0xB4E10C23

binary value (1011_0100_1110_0001_0000_1100_0010_0011),

executing, RBIT.W R0, R1;

result is R0 = 0xC430872D

binary value (1100_0100_0011_0000_1000_0111_0010_1101)

6) SXTB, SXTH, UXTB, and UXTH - are four instructions used to extend a byte or half word into a word. Syntax -

```
SXTB  <Rd>, <Rn>        UXTB <Rd>, <Rn>
SXTH  <Rd>, <Rn>        UXTH <Rd>, <Rn>
```

→ SXTB/SXTH, the data are sign extended using bit[7]/bit[15] of Rn.
→ UXTB/UXTH the value is zero extended to 32-bit.

Example - R0 = 0x55AA8765;

Instruction	Result
SXTB R1, R0;	R1 = 0xFFFFFF65
SXTH R1, R0;	R1 = 0xFFFF8765
UXTB R1, R0;	R1 = 0x00000065
UXTH R1, R0;	R1 = 0x00008765

7) Bit Field Clear and Bit Field Insert

Bit Field Clear (BFC) clears 1-31 adjacent bits in any position of a register.

Syntax - BFC.W <Rd>, <#lsb>, <#width>

Example -
```
LDR   R0, =0x1234FFFF
BFC.W R0, #4, #8
```
Result - R0 = 0x1234F00F.

Bit Field Insert (BFI) copies 1-31 bits (#width) from one register to any location (#lsb) in another register.

Syntax - BFI.W <Rd>, <#lsb>, <#width>

Example -
```
LDR   R0, =0x12345678
LDR   R1, =0x3355AACC
BFI.W R1, R0, #8, #16; Insert R0[15:0] to R1[23:8]
```
Result - R1 = 0x335678CC

8) UBFX and SBFX - are the unsigned and signed bit field extract instructions.

Syntax - UBFX.W <Rd>, <Rn>, <#lsb>, <#width>
SBFX.W <Rd>, <Rn>, <#lsb>, <#width>

8) UBFX and SBFX

UBFX extracts a bit field from a register starting from any location (specified by #lsb) with any width (specified by #width), zero extends it, and puts it in the destination register.

Example - LDR R0, = 0x5678ABCD
UBFX.W R1, R0, #4, #8

Result - R1 = 0x000000BC

SBFX extracts a bit field, but its sign extends it before putting it in a destination register.

Example - LDR R0, = 0x5678ABCD
SBFX.W R1, R0, #4, #8

Result - R1 = 0xFFFFFFBC.

9) LDRD and STRD → are two instructions which are used to transfer two words of data from or into two registers.

Syntax -

```
LDRD.W <Rxf>, <Rxf2>, [Rn, # +/- offset] {!} ; Pre-indexed
LDRD.W <Rxf>, <Rxf2>, [Rn], # +/- offset ; Post-indexed
STRD.W <Rxf>, <Rxf2>, [Rn, #+/- offset] {!} ; Pre-indexed
STRD.W <Rxf>, <Rxf2>, [Rn], #+/- offset ; Post indexed
```

where <Rxf> - first destination/source register.
<Rxf2> - second destination/source register.

→ When using LDRD avoid using same register for <Rn> and <Rxf> because of an error in Cortex M3.

Example - (1) The code reads a 64-bit value located in memory address 0x1000 into R0 and R1:

```
LDR R2, =0x1000
LDRD.W R0, R1, [R2] ; This will gives R0= memory[0x1000],
                    ; R1 = memory [0x1004]
```

(2) STRD is used to store a 64-bit value in memory. Code is written in pre-indexed addressing mode.

```
LDR R2 = 0x1000 ; Base address
STRD.W R0, R1, [R2, #0x20]; This will gives memory
                    ; [0x1020] = R0, memory [0x1024] = R1
```

(10) Table Branch Byte and Table Branch Halfword.

→ Table Branch Byte (TBB) and Table Branch Halfword (TBH) are for implementing branch tables.

→ TBB instruction uses a branch table of byte size offset, and TBH uses a branch table of half word offset.

→ Since the bit 0 of a program counter is always zero, the value in the branch table is multiplied by two before it's added to PC.

→ because the PC value is the current instruction address plus four, the branch range for TBB is (2×255)+4 = 514, and

→ the branch range for TBH is (2×65535)+4 = 131074.

→ Both TBB and TBH support forward branch only.

→ TBB has this general syntax- TBB.W [Rn, Rm]

where Rn is the base memory offset and Rm is the branch table index.

→ The branch table item for TBB is located at Rn + Rm.

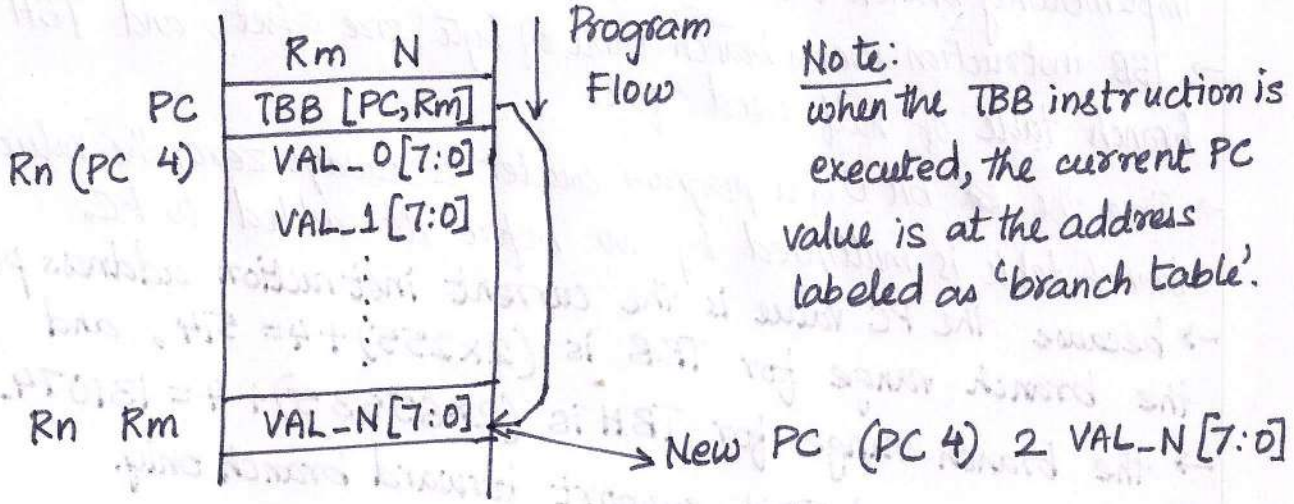
→ The TBB branch table can be created as follows-

```
TBB.W [PC,R0]; when executing this instruction, PC equal
              ; branch table

branch table
    DCB ((dest0 - branchtable)/2); Note that DCB is used
                                 ; because the value is 8bit
    DCB ((dest1 - branchtable)/2)
    DCB ((dest2 - branchtable)/2)
    DCB ((dest3 - branchtable)/2)

dest0
    ....; Execute if R0 = 0
dest1
    ....; Execute if R0 = 1
dest2
    ....; Execute if R0 = 2
dest3
    ....; Execute if R0 = 3.
```

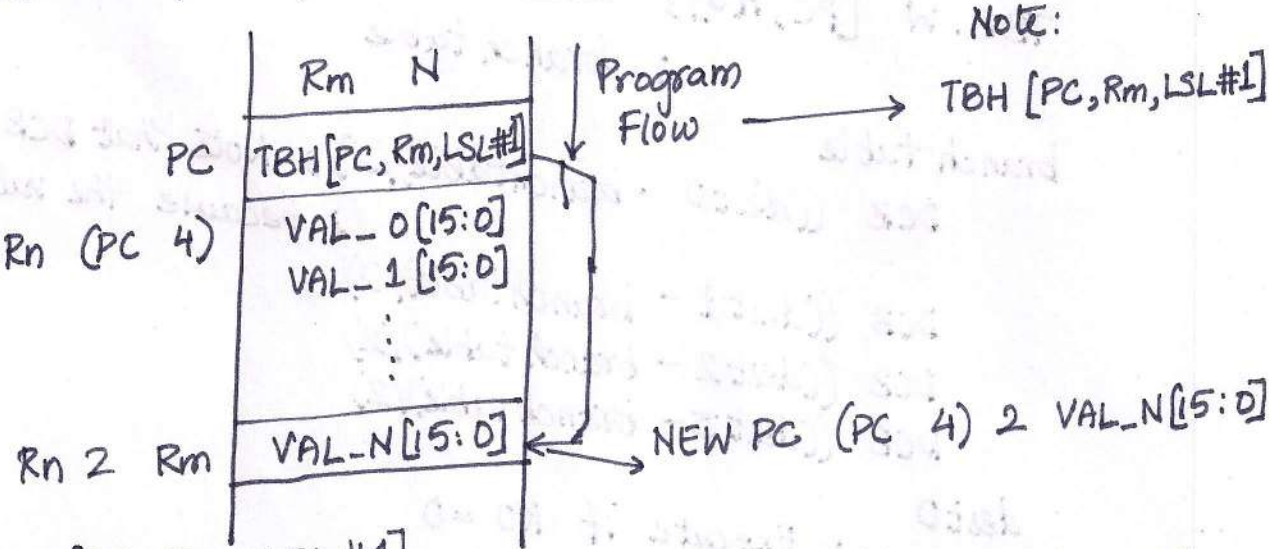
TBB Operation – Assume, we use PC for Rn, we can see the operation as shown in figure.



Note: when the TBB instruction is executed, the current PC value is at the address labeled as 'branch table'.

TBH Operation – For TBH instruction, the process is similar except the memory location of the branch table item is located at (Rn + 2 × Rm) and the maximum branch offset is higher.

→ We assume that Rn is set to PC as shown in figure. If Rn in the table branch instruction is set to R15, the value used for Rn will be PC + 4 because of the pipeline in the processor.



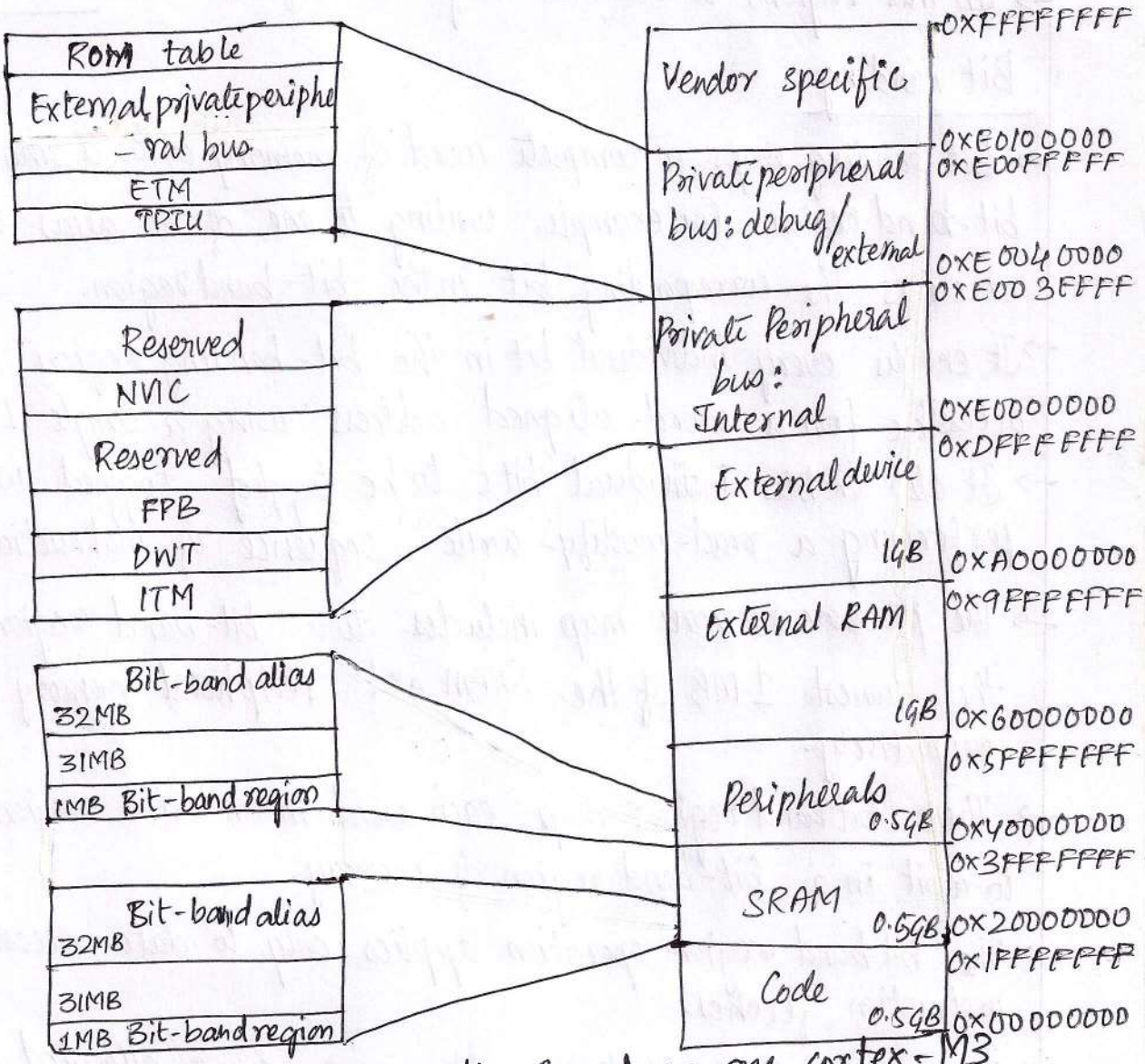

```
TBH.W [PC, R0, LSL#1]
branch table
        DCI ((dest0 - branchtable)/2) ; Note that DCI is used because
        DCI ((dest1 - branch table)/2);  ; the value is 16 bit
        DCI ((dest2 - branch table)/2);
        DCI ((dest3 - branch table)/2);
dest0 ....; Execute if R0 = 0
dest1 ....; Execute if R0 = 1
dest2 ....; Execute if R0 = 2
dest3 ....; Execute if R0 = 3
```

MODULE-2 ARM CORTEX M3 INSTRUCTION SETS AND PROGRAMMING-II

Memory Map (of Cortex-M3 processor)

B.S.Balaji,
Asst. Prof, BGSIT.

The Cortex-M3 processor has a fixed memory map as shown in figure.



→ It makes it easier to port software from one cortex-M3 product to another.

→ The Cortex M3 processor has a total of 4 GB of address space.

→ Program code can be located in the code region, the Static Random Access Memory (SRAM) region, or the external RAM region.

→ It is best to put the program code in the code region because it helps in instruction fetching and data accesses are carried out simultaneously on two separate bus interfaces.

→ Code Instruction fetches are performed over the Icode bus. Data accesses are performed over the DCode bus.

→ The 0.5 GB SRAM memory range is for connecting internal SRAM.

→ SRAM Instruction fetches and data accesses are performed over the system bus

→ In this region, a 32-MB range is defined as a bit-band alias.

Bit banding

→ Bit banding maps a complete word of memory onto a single bit in the bit-band region. For example, writing to one of the alias words sets or clears the corresponding bit in the bit-band region.

→ It enables every individual bit in the bit-banding region to be directly accessible from a word-aligned address using a single LDR instruction.

→ It also enables individual bits to be ~~toggled~~ toggled without performing a read-modify-write sequence of instructions.

→ The processor memory map includes two bit-band regions. These occupy the lowest 1MB of the SRAM and Peripheral memory regions respectively.

→ These bitband regions map each word in an alias region of memory to a bit in a bit-band region of memory.

→ The bit band region operation applies only to data accesses no instruction fetches.

→ The next 0.5GB block of address range is allocated to on-chip peripherals.

→ Similar to SRAM region, it supports bit-band alias and is accessed via the system bus interface. but instruction execution in this region is not allowed.

→ Two slots of 1-GB memory space are allocated for external RAM and external devices.

→ The difference b/w the two is that program execution in the external device region is not allowed and there are some differences with the caching behaviours

→ The last 0.5GB memory is for the system-level components, internal peripheral buses, external peripheral bus and vendor-specific system peripherals.

38

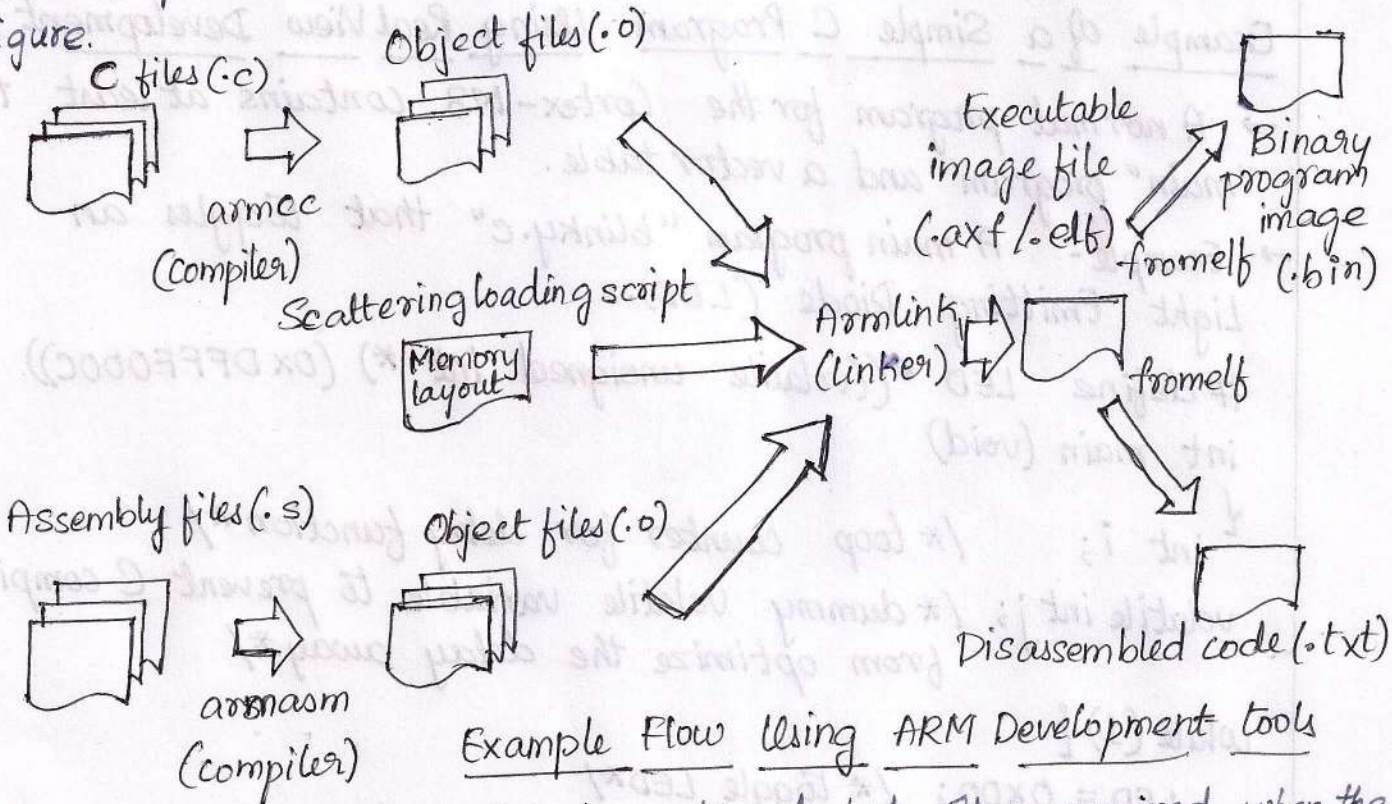
18.03.2018

CORTEX - M3 Programming

→ Cortex M3 can be programmed using either assembly language, C language or high-level languages like National Instruments Labview.
→ In most embedded applications, Cortex M3 processor can be used where the software can be written entirely in C language.
→ Developers frequent use assembly language or a combination of C and assembly language in their projects.
→ The procedure of building and downloading the resultant image files to the target device is largely dependent on the tool used.

A Typical Development flow

→ The concepts of code generation flow in terms of these various tools/ software programs are available for developing Cortex-M3 applications.
→ Programmer needs assembler, a C compiler, a linker, and binary files generation utilities.
→ For ARM solutions, the Real View Development Suite (RVDS) or Real View Compiler Tools (RVCT) provide a file generation flow, as shown in figure.



Example Flow Using ARM Development tools

→ The scatter-loading script is optional but often required when the memory map becomes more complex.
→ RVDS also contains a large no. of utilities, including an Integrated Development Environment (IDE) and debuggers.

Using C

→ For beginners in embedded programming using C language for software development on the Cortex-M3 processor is the best choice.

→ It is easier as the most of microcontroller vendors provide device driver libraries written in C to control peripherals.

→ Modern C compilers can generate very efficient code, hence it is better to program in C than spending a lot of time to try to develop complex routines in assembly language which is error prone and less portable.

→ C is a generic computer language and advantage of being portable and easier for implementing complex operations, compared with assembly language.

→ A no. of Cortex-M3 programs examples are already included in the installation of the ARM C compiler products like RVDS or Keil Real View Microcontroller Development Kit (MDK-ARM).

Example of a Simple C Program Using Real View Development Site

→ A normal program for the Cortex-M3 contains at least the "main" program and a vector table.

→ Example- A main program "blinky.c" that toggles an Light Emitting Diode (LED):

```
#define LED *((volatile unsigned int *) (0x DFFF000C))
int main (void)
{
  int i;        /* loop counter for delay function */
  volatile int j; /* dummy volatile variable to prevent C compiler
                     from optimize the delay away */
  while (1){
     LED = 0x00;  /* toggle LED*/
    for (i=0; i<10; i++) {j=0;} /* delay */
    LED = 0x01;  /* toggle LED*/
    for (i=0; i<10; i++) {j=0;} /* delay */
  }
  return 0;
}
```

The file "vectors.c" contains the vector table, as well as a number of dummy exception handlers.

ex- typedef void (*const ExecFuncPtr) (void)_irq;
extern int_main (void);

```
/*
 * Dummy handlers Exception handlers
 */
_irq void NMI_Handler (void)
{ while(1);}
_irq_void HardFault_Handler(void)
{ while(1);}
_irq void SVC_Handler (void)
{while(1);}
_irq void DebugMon_Handler(void)
{while(1);}
_irq void PendSV_Handler (void)
{while(1);}
_irq void SysTick_Handler(void)
{while(1);}
_irq void ExtInt0_IRQHandler(void)
{while(1);}
_irq void Ext Int1_IRQ Handler (void)
{while(1);}
_irq void Ext Int1_IRQ Handler (void)
{while(1);}
_irq void Ext Int2_IRQHandler (void)
{while(1);}
_irq void Ext Int3_IRQ Handler (void)
{while(1);}
```

```
(ExecFuncPtr) 0x20002000,
(ExecFuncPtr)_main,
NMI_Handler, /* NMI */
HardFault_Handler,
0, /* MemManage_Handler
  in Cortex-M3*/
0,/* ~~Mem~~Bus Fault_Handler
  in Cortex_M3*/
0,/* Usage Fault_Handler
  in Cortex -M3*/
0, /* Reserved */
0, /* Reserved */
0, /* Reserved */
0, /* Reserved */
SVC_Handler,
0, /* Debug Mon_Handler in
   Cortex-M3*/
0,/* Reserved*/
PendSV_Handler
SysTick_Handler,
/*External Interrupts*/
ExtInt0_IRQHandler,
ExtInt1_IRQ Handler,
ExtInt2_IRQ Handler,
ExtInt3_IRQ Handler
};
#pragma arm section
```

#pragma arm section rodata = "exceptions area"

ExecFuncPtr exception_table[] = { /* vector table */

Assume using RVDS through Command line

```
$> armcc -c -g -W blinky.c -o blinky.o
$> armcc -c -g -W vectors.c -o vectors.o
```

Compile the Same Example Using ~~Kiel~~ Keil MDK-ARM

→ In KEIL MDK-ARM, it is possible to compile the same program as in RVDS.

→ The command line options and a few symbols in the linker script (scattering loading file) have to be modified.

```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE (STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE (STACK_BASE-HEAP_BASE)/2)

LOAD_REGION 0x00000000 0x00200000
{ VECTORS 0x0 0xC0
  { ; Provided by the user in vectors.c
    * (exceptions_area)
  }
  CODE 0xC0 FIXED
  { * (+RO)
  }
  DATA 0x20000000 0x00010000
  { * (+RW, +ZI)
  }
  ;; Heap starts at 4kB and grows upwards
  Heap_Mem HEAP_BASE EMPTY HEAP_SIZE
  {
  }
  ;; Stack starts at the end of the 8kB of RAM
  ;; And grows downwards for 2kB
  Stack_Mem STACK_BASE EMPTY -STACK_SIZE
  {
  }
}
```

Accessing Memory-Mapped Registers in C

→ There are various ways to access memory-mapped peripheral registers in C language.

→ For example, System Tick (SYSTICK) Timer in the Cortex-M3 is used as an peripheral to demonstrate different access methods in C language.

→ The SYSTICK is a 24-bit timer which contains only four registers.

→ Method 1 - It is the easiest method - defining each register as a pointer.

Accessing Peripheral Registers as Pointers.

```
#define SYSTICK_CTRL (*((volatile unsigned long*)(0xE000E010)))
#define SYSTICK_LOAD (*((volatile unsigned long*)(0xE000E014)))
#define SYSTICK_VAL  (*((volatile unsigned long*)(0xE000E018)))
#define SYSTICK_CALIB (*((volatile unsigned long*)(0xE000E01C)))

/* Setup SYSTICK */
  SYSTICK_LOAD  0xFFFF; //set reload value
  SYSTICK_VAL   0x0;  // Clear current value
  SYSTICK_CTRL  0x5; //Enable SYSTICK and select core clock
```

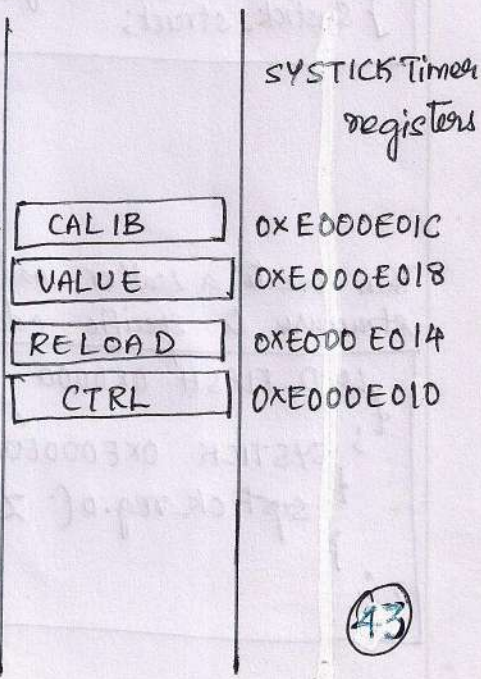
→ Based on the same method, A macro can be used to convert address values to C pointer.

→ Method 2 - is to define the registers as a data structure, and then define a pointer of the defined structure.
The C-code looks a bit different, but the generated code is the same as method-1.

Alternative way of Accessing Peripheral Registers as Pointers.

```
#define HW_REG(addr) (*((volatile unsigned long*)(addr)))
#define SYSTICK_CTRL   0xE000E010
#define SYSTICK_LOAD   0xE000E014
#define SYSTICK_VAL    0xE000E018
#define SYSTICK_CALIB  0xE000E01C

/*Setup SysTick */
 HW_REG (SYSTICK_LOAD) 0xFFFF; //Set reload value
 HW_REG (SYSTICK_VAL)  0x0; //Clear current value
 HW_REG (SYSTICK_CTRL) 0x5; //Enable SYSTICK
                            //and select core clock.
```



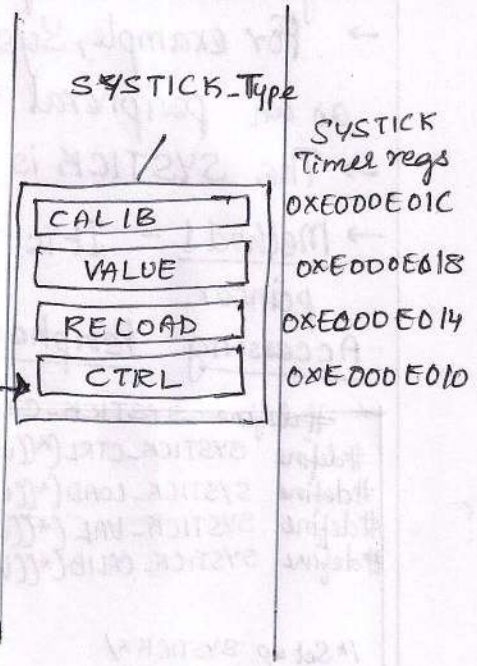
43

This is the method used in CMSIS compliant device driver libraries.

Accessing Peripheral Registers as Pointers to Elements in a Data Structure

```
typedef struct
{ volatile unsigned long CTRL; /*systick CONTROL and status reg*/
  volatile unsigned long LOAD; /*SYSTICK Reload Value reg*/
  volatile unsigned long VAL; /*SysTick Current Value register*/
  volatile unsigned long CALIB; /*SysTick Calibration register*/
} SysTick_Type;
#define SysTick ((SysTick_Type *)0xE000E010) /*SysTick struct*/

/*Setup Systick*/
  Systick -> LOAD  0xFFFF; // Set reload value
  Systick -> VAL   0x0   ; // Clear current value
  Systick -> CTRL  0x5; //Enable SYSTICK and select core clock
```

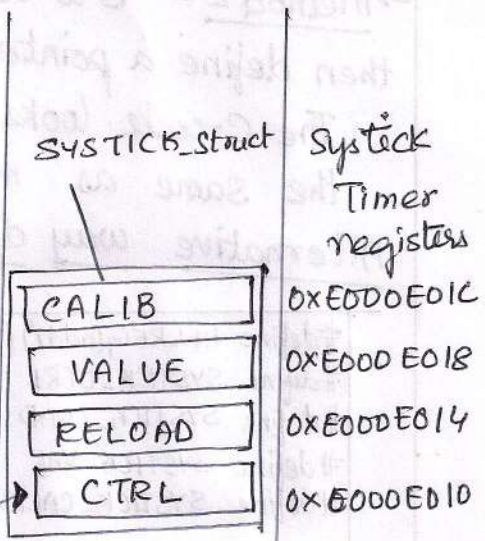


Method 3 also uses data structure, but the base address of the peripheral is defined using a scatter loading file (or linker script) during linking stage.

Defining Peripheral-Based Address Using Scatter Loading file.

In C File, define the data structure as

```
__attribute__ ((zero_init)) struct {
  volatile unsigned long CTRL; /*systick control*/
  volatile unsigned long RELOAD; /*systick reload*/
  volatile unsigned long VAL; /*systick value*/
  volatile unsigned long CALIB; /*systick calibration*/
} Systick_struct;
```



Then create a scatter loading file to place the data structure to specific address

```
LOAD_FLASH 0x0000
{
; SYSTICK 0xE000E010 UNINIT
  { systick_reg.o( ZI)
  }
};
```

→ In this case, the program code using the peripheral has to define the peripheral as a C pointer in an external object.
→ The code for accessing the register is the same as in the second method.

Method 1 → It is the simplest and results in less efficient code compared with the others as the address value for the registers are stored separately as constant.
→ As a result, the code size can be larger and might be slower as it requires more accesses to the program memory to set up the address values.
→ for peripheral control code that only accesses to one register, the efficiency of method 1 is identical to others.

Method-2 → It ~~is~~ allows the registers in a peripheral to share just one constant for base address value.
→ The immediate offset address mode can be used for access of each register. It is the method used in CMSIS.

Method-3 → It has the same efficiency as method 2, but it is less portable due to the use of a scatter loading file.
→ It is required when you are developing a device driver library for a peripheral that is used in multiple devices, (and, the base address of the peripheral is not known until in the linking stage).

Intrinsic Functions

→ It is used just like normal C functions. where C language can often speed up application development and used to generate some instructions that cannot be generated using normal C-code.
→ Some C compilers provide intrinsic functions for accessing these special instructions.
→ For example, ARM compilers (including RealView C Compilers and Keil MDK-ARM) provide the intrinsic functions. (listed in the Table.) for commonly used instructions.

45

Instrinsic Functions Provided in ARM Compilers

Assembly Instructions	ARM Compiler Intrinsic Functions
CLZ	unsigned char __clz (unsigned int val)
CLREX	void __clrex (void)
CPSID I	void __disable_irq (void)
CPSIE I	void __enable_irq (void)
CPSID F	void __disable_fiq (void)
CPSIE F	void __enable_fiq (void)
LDREX/LDREXB/LDREXH	unsigned int __ldrex (volatile void *ptr)
LDRT/LDRBT/LDRSBT/LDRHT/LDRSHT	unsigned int __ldrt (const volatile void *ptr)
NOP	void __nop (void)
RBIT	unsigned int __rbit (unsigned int val)
REV	unsigned int __rev (unsigned int val)
ROR	unsigned int __ror (unsigned int val, unsigned int shift)
SSAT	int __ssat (int val, unsigned int sat)
SEV	void __sev (void)
STREX/STREXB/STREXH	int __strex (unsigned int val, volatile void *ptr)
STRT/STRBT/STRHT	void int __strt (unsigned int val, const volatile void *ptr)
USAT	int __usat (unsigned int val, ~~const~~ unsigned int sat)
WFE	void __wfe (void)
WFI	void __wfi (void)
BKPT	~~void __breakpoint~~ void __breakpoint

Embedded Assembler and Inline Assembler

→ It is an alternative for intrinsic functions where it can access assembly instructions in C-code.

→ It is necessary in low-level system control (or) to implement a timing critical routine and decide to implement it in assembly for the best performance.

→ In most of ARM C Compilers allow programmer to include assembly code in form of inline assembler.

→ Traditionally inline assembler is used but in Real View C Compiler does not support instructions in Thumb-2 technology.

→ Embedded Assembler is included in new version of Real View C Compiler 3.0, and it supports the instruction set in Thumb-2.

For example:

```
__asm void SetFaultMask (unsigned int new_value)
{
// Assembly code here
    MSR FAULTMASK, new_value // write new value to FAULTMASK
    BX LR                    // Return to calling program.
}
```


46

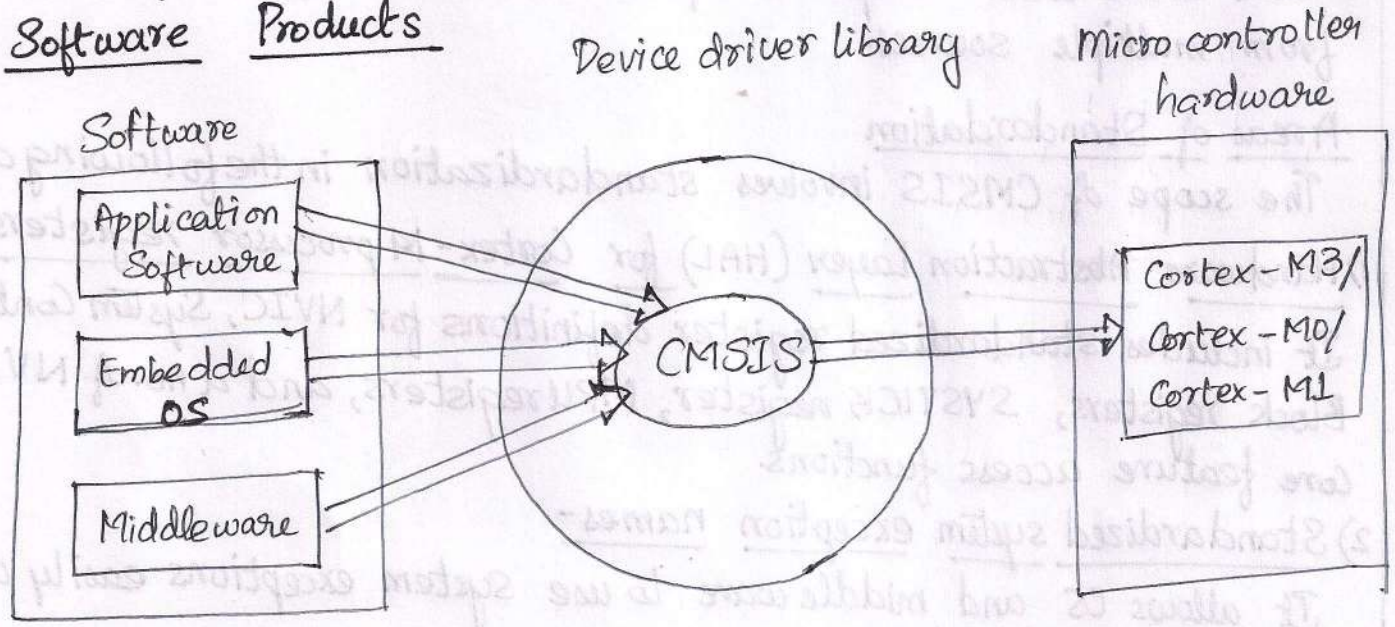
Cortex Microcontroller Unit (MCU) Software Interface Standard

CMSIS - A Background

→ The CMSIS was developed by ARM to allow users of the Cortex-M3 microcontrollers to gain the most benefit from all these software solutions, and

→ It allows them to develop their embedded application quickly and reliably.

CMSIS provides a Standardized Access Interface for Embedded Software Products



→ The Cortex-M3 microcontroller are gaining momentum in the embedded application market as more and more products based on Cortex M3 processor and softwares ~~that~~ are emerging.

→ A no. of companies providing embedded software solutions, including codecs, data processing libraries, and various software and debug solutions.

→ It was started in 2008 to improve software usability and inter-operability of ARM microcontroller software.

→ It is integrated into driver libraries provided by silicon vendors.

→ It provides a standardized software interface for the Cortex-M3 processor features as well as no. of common system and I/O functions.

→ The library is also supported by softwares ~~companies~~ including embedded OS ~~or~~ vendors and Compiler vendors.

The aims of CMSIS are -
→ improve software portability and reusability.
→ enable software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors.
→ allow embedded developers to develop software quicker with an easy-to-use and standardized software interface.
→ allow embedded software to be used on multiple compiler products.
→ avoid device driver compatability issues when using software solutions from multiple sources.

Areas of Standardation

The scope of CMSIS involves standardization in the following areas-

1) Hardware Abstraction Layer (HAL) for Cortex-M processor registers -
It includes standardized register definitions for NVIC, System Control Block registers, SYSTICK register, MPU registers, and a no. of NVIC & Core feature access functions.

2) Standardized system exception names -
It allows OS and middleware to use system exceptions easily without compatability issues.

3) Standardized method of header file organization -
It makes easier for users to learn new Cortex MCU and improves software portability.

4) Common method for system initialization -
Each Microcontroller Unit (MCU) vendor provides a SystemInit() function in their device driver library for essential setup and configuration, such as initialization of clocks.

5) Standardized intrinsic functions - Intrinsic functions are normally used to produce instructions that cannot generated by IEC/ISO C.

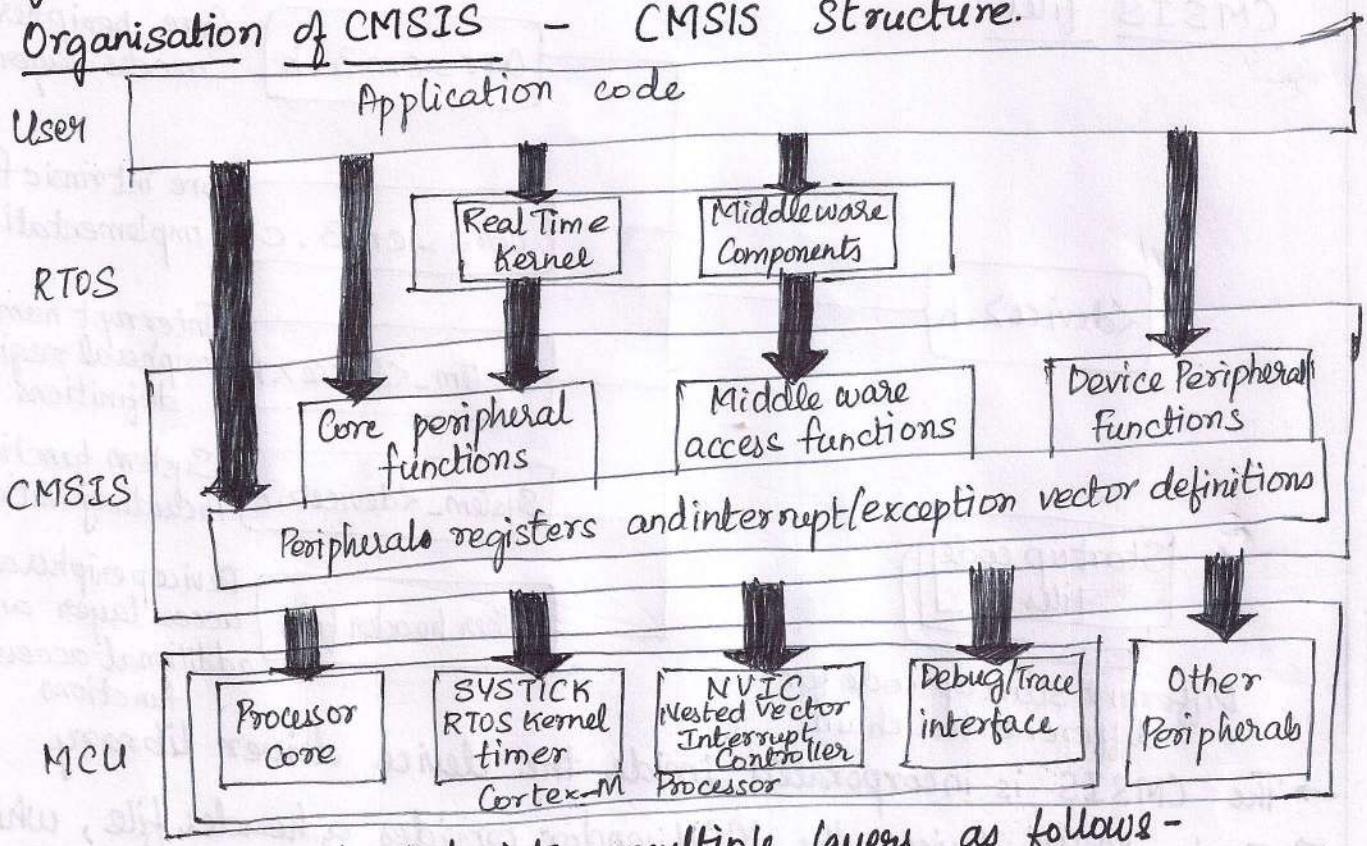
6) Common access functions for communication - It provides a set of software interface functions for common communication interfaces including universal asynchronous receiver/transmitter (UART), Ethernet, and Serial Peripheral Interface (SPI).

7) Standardized way for embedded software to determine system clock frequency - A software variable called System frequency is defined in device driver code.

48

It allows embedded OS to set up the SYSTICK unit based on the system clock frequency.

Organisation of CMSIS - CMSIS Structure.



The CMSIS is divided into multiple layers as follows -

1) Core Peripheral Access layer - Name definitions, and helper functions to access core registers and core peripherals.

2) Middleware Access layer -
- Common method to access peripherals for the software industry (work in progress).
- Targeted communication interfaces include Ethernet, UART, and SPI.
- Allows portable software to perform communication tasks on any Cortex microcontrollers that support the required communication interface.

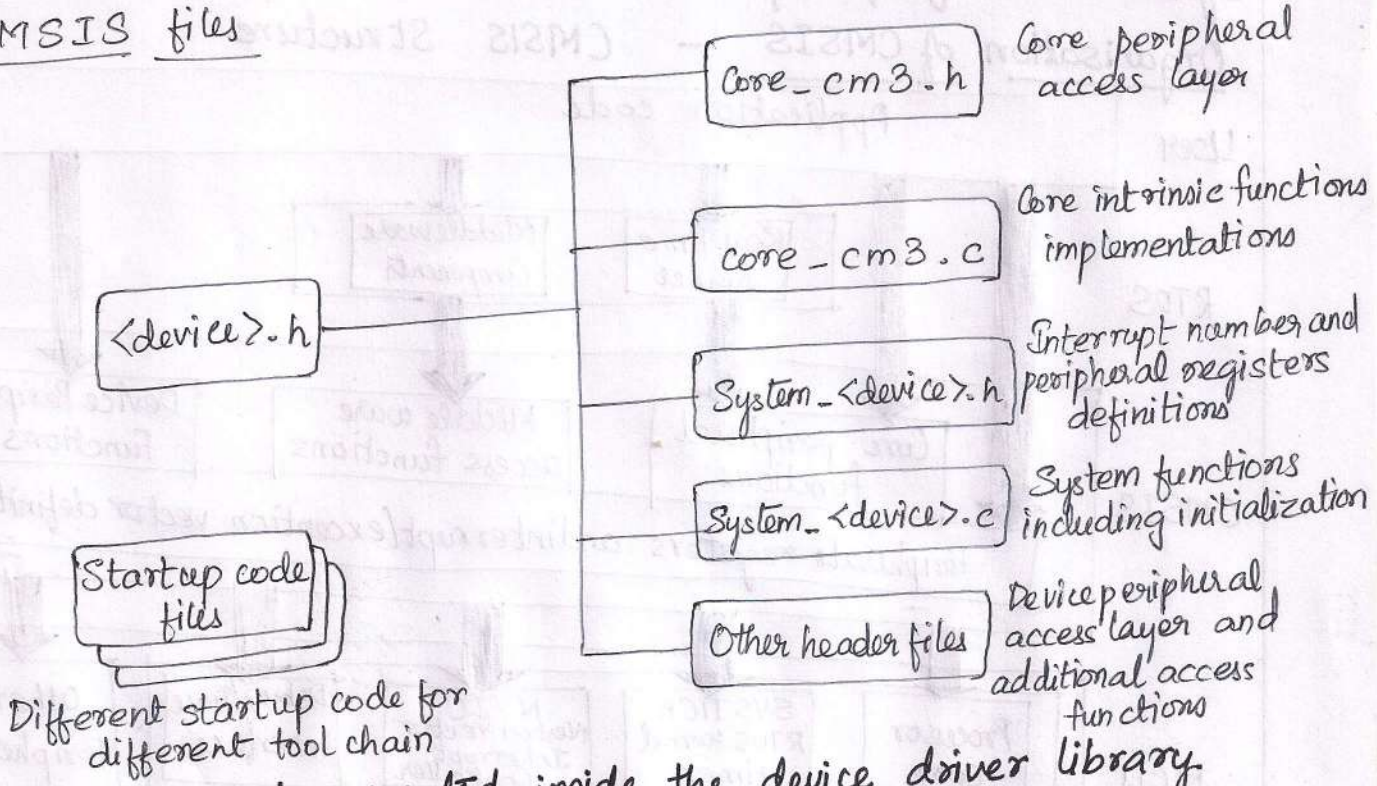
3) Device Peripheral Access layer (MCU Specific) - Name definitions, address definitions, and driver code to access peripherals.

4) Access Functions for Peripherals (MCU Specific) - Optional additional helper functions for peripherals.

49

Using CMSIS

CMSIS files



→ The CMSIS is incorporated inside the device driver library.
→ Each MCU device, the MCU vendor provides a header file, which pulls in additional header files required by the device driver library, including the Core Peripheral Access Layer (defined by ARM).
→ The file core_cm3.h contains the peripheral register definitions and access functions for the Cortex-M3 processor peripherals like NVIC, System Control Block registers, and SYSTICK registers.
→ It ~~also~~ contains declaration of CMSIS intrinsic functions to allow C applications to access instructions that cannot be generated using IEC/ISO C language.
→ It also contains a function for outputting a debug message via the Instrumentation Trace Module (ITM).
→ The file core_cm3.c contains implementation of CMSIS intrinsic functions that cannot be implemented in core_cm3.h using simple definitions.
→ The system_<device>.h file contains microcontroller specific interrupt number definitions, and peripheral register definitions.
→ The system_<device>.c file contains a microcontroller specific function called SystemInit for system initialization.

50

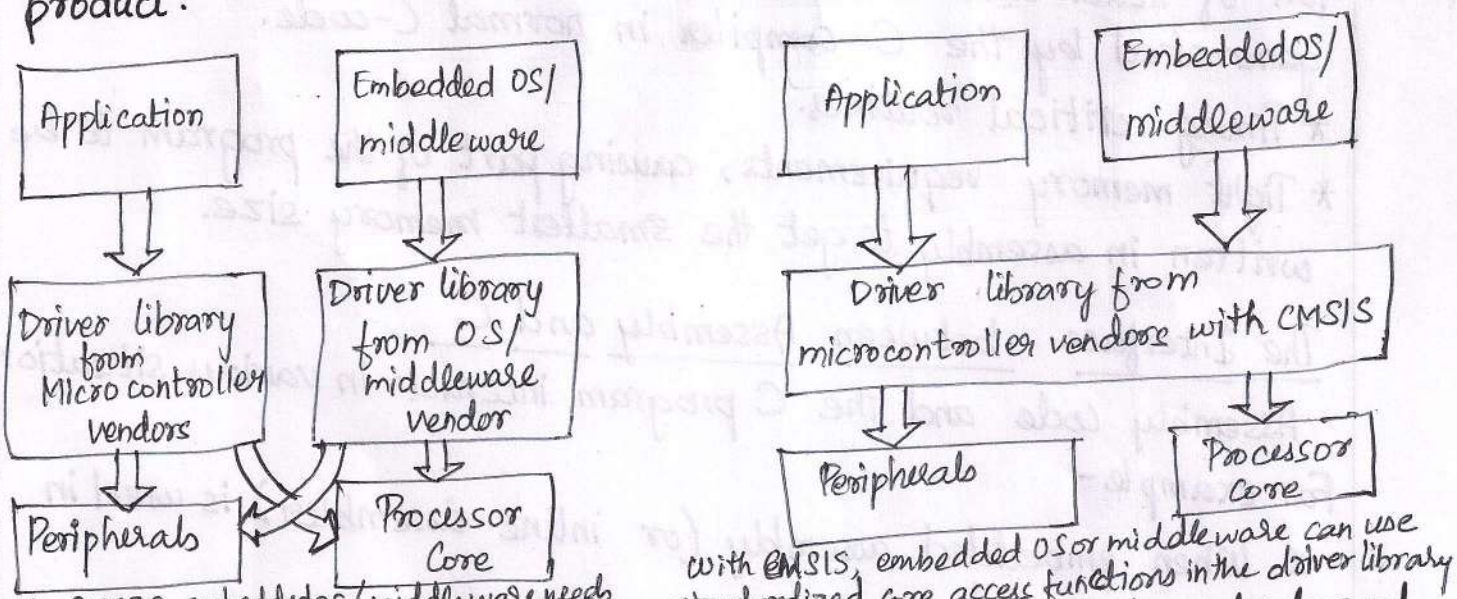
Benefits of CMSIS

→ The main advantage is much better software portability and reusability, easy migration between different Cortex-M3 microcontrollers.

→ It allows software to be quickly ported between Cortex-M3 and other Cortex-M3 processors, reducing time to market.

→ With the CMSIS, their software products can become compatible with device drivers from multiple microcontroller vendors, including future microcontroller products.

→ Without the CMSIS, software should include a small library for Cortex-M3 core functions or develop multiple configurations of their product.



without CMSIS, embedded OS/ middleware needs to include processor core access functions & might need to include a few peripheral drivers.

with CMSIS, embedded OS or middleware can use standardized core access functions in the driver library

→ Embedded OS and middleware can be MCU vendor independent and Compiler tool vendor independent.

→ The CMSIS has a small memory foot print (less than 1KB); and

→ It also avoids overlapping of core peripheral driver code when reusing software code from other projects.

→ With CMSIS, embedded OS or middleware can use standardized core access functions in the driver library.

→ Without CMSIS, embedded OS or middleware needs to include processor core access functions and might need to include a few peripherals drivers.

→ CMSIS is supported by multiple compiler vendors, embedded software can compile and run with different compilers.

Using Assembly

→ It is possible to develop the whole application in assembly language only for small projects but often much harder for beginners.
→ Using assembles, it is easy to get best optimization but increases development time
→ It is extremely difficult to handle complex data structures or function library management.
→ When the C language is used in a project. where the part of the program is implemented in assembly language -
* Functions that cannot be implemented in C, such as direct manipulation of stack data or special instructions that cannot be generated by the C compiler in normal C-code.
* Timing- critical routines.
* Tight memory requirements, causing part of the program to be written in assembly to get the smallest memory size.

The Interface between Assembly and C

Assembly code and the C program interact in various situations. For example-
* When embedded assembly (or inline assembler) is used in C program code.
* when C program code calls a function or subroutine implemented in assembler in a separate file.
* When an assembly program calls a C function or subroutine.

→ For simple cases, when a calling program needs to pass parameters to a subroutine or function, it will use registers R0-R3, where R0 is the first parameter, R1 is the second, and so on.
→ Similarly, R0 is used for returning a value at the end of the function.
→ R0-R3 and R12 can be changed by a function or routine whereas the contents of R4-R11 should be restored to the previous state before entering the function,
→ It is done by stacks PUSH and POP.

52

The first step in Assembly Programming -

The first simple program can be like this -

```
STACK_TOP EQU 0x20002000; constant for SP starting value
  AREA |Header Code|, CODE
  DCD STACK_TOP ; Stack top
  DCD Start ; Reset vector
  ENTER      ; Indicate program execution start here
        Start ; Start of main program
              ; initialize registers
  MOV r0, #10; Starting loop counter value
  MOV r1, #0; starting result
            ; calculated 10+9+8,....+1
   loop
      ADD r1, r0;    R1 = R1 + R0
      SUBS r0, #1; Decrement R0, update flag ("S" suffix)
      BNE loop     ; if result not zero jump to loop
      ; Result is now in R1
   dead loop
      B dead loop ; infinite loop
      END         ; END of file.
```

→ This simple program contains the initial stack pointer (SP) value, the initial program counter (PC) value, and setup registers and then does the required calculation in a loop.

Producing Outputs

→ The simplest way to connect microcontroller to the outside world is to turn on/off the LEDs.

→ One of the most common output methods is to send text messages to a console.

→ UART is the interface connecting the personal computer and embedded product development.

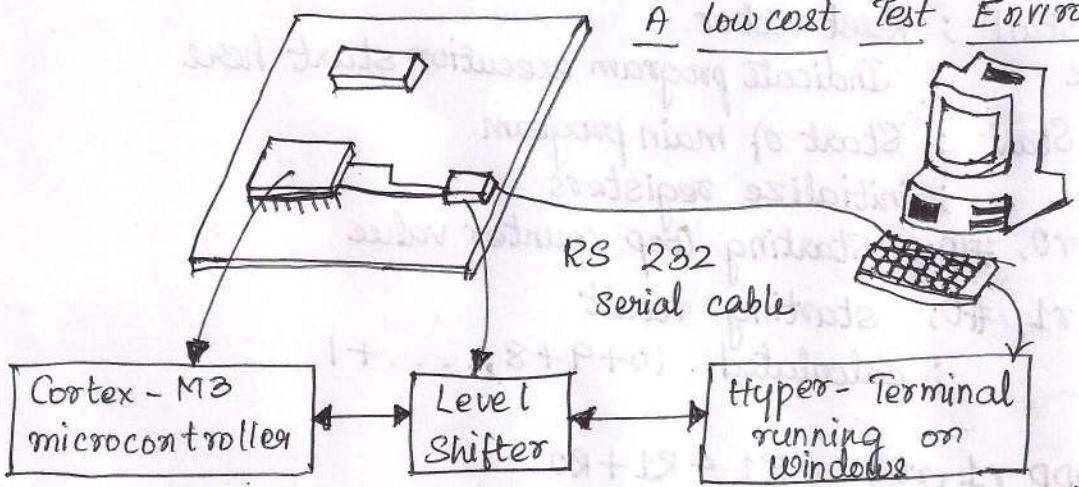
For example - a computer running a Windows system with the Hyper-Terminal program acting as a console to produce outputs.

53

→ The Cortex M3 processor does not contain a UART interface, but ~~most~~ it is provided by the Chip manufacturers which differs ~~from~~ in various devices.

→ Assume that a UART is available and has a status flag to indicate whether the transmit buffer is ready for sending out new data.

A low cost Test Environment for Outputting Text messages:



→ A level shifter is needed in the connection because RS 232 has a different voltage level than the microcontroller I/O pins.

Other features to help output debugging messages which are implemented on Cortex-M3 processor

1) Semihosting - Depending on the debugger and code library support, semihosting (~~output~~ displaying output printf messages via a debug probe device) can be be done via debug register in the NVIC.

2) Instrumentation Trace - It provides a trace port and an external Trace Port Analyzer (TPA) is available, instead of using UART to output messages.

→ The Trace port works much faster than UART and can offer more data channels.

3) Instrumentation trace via Serial-Wire Viewer (SWV) -

It provides an SWV operation mode on the Trace Port Interface Unit (TPIU). It allows outputs from ITM to be captured using low-cost hardware instead of a TPA. It provides limited bandwidth and does not ~~support~~ support large ~~amo~~ data transfer.

54

The "Hello World" Example -

Program to output a character via UART (where UART is available with status flag to indicate the transfer buffer is ready for sending out new data.)

```
UART0_BASE   EQU  0x4000C000
UART0_FLAG   EQU  UART0_BASE + 0x018
UART0_DATA   EQU  UART0_BASE + 0x000
  Putc                              ; Subroutine to send a character via UART
                                    ; Assume character to be sent is in R0
    PUSH {R1, R2, LR}               ; Save registers
    LDR  R1, =UART0_FLAG            ; Get the address of UART Flag
Wait LDR  R2, [R1]                  ; Get status flag
    TST  R2, #0x20                  ; Check transmit buffer full flag bit
    BNE  Wait                       ; If busy then wait until transmit buffer
                                    ; is ready
    LDR  R1,= UART0_DATA            ;otherwise load the address of transmit
                                    ;buffer in R1
    STRB R0, [R1]                   ; output data to transmit buffer
    POP  {R1, R2, PC}               ; Restore registers and Return.
```

Note - Register addresses and Bit definitions used here are just examples.

Subroutine to sent the text message to display device.

```
      Puts                          ;Subroutine to send text message to UART
                                    ;Assume that the starting address of the text
                                    ;string is in R1 and the string is terminated
                                    ; by NULL character
      PUSH {R0, R1, LR}             ; Save registers
LOOP  LDRB R0, [R1], #1             ; Read one character and increment address
      CBZ  R0, EXIT                 ; if character is NULL, stop
      BL   Putc                     ; Otherwise, output character to UART
      B    LOOP                     ; Repeat the process for next character
EXIT  POP  {R0, R1, PC}             ; Restore registers and Return.
```


55

Program to display "Hello World"

```
STACK_TOP EQU 0x20002000;
UART0_BASE EQU 0x4000C000;
UART0_FLAG EQU UART0_BASE+0x018
UART0_DATA EQU UART0_BASE+0x000
AREA| Header Code|, CODE
DCD STACK_TOP              ;Stack Pointer initial value
DCD Start                  ; Reset vector
ENTRY
Start                      ; Start of main program
    MOV R0, #0             ; Initialize registers
    MOV R1, #0
    MOV R2, #0
    BL Uart0Initialize ; Initialize the UART0
   LDR R1, =HELLO_TXT ; Set R1 to starting address of string
    BL Puts
Infiloop  B Infiloop       ; Infinite loop
;- - - - - - - - - - -
; Subroutines
;- - Puts - - - - - - - - - ; Subroutine to send string to UART
                           ; Assume that the starting address of the text string is in R1 and
                           ; the string is terminated by NULL character.
   PUSH {R0, R1, LR} ; Save registers
LOOP  LDRB R0, [R1], #1 ; Read one character and increment address
   CBZ R0, EXIT ; If character is NULL, stop
   BL Putc     ; Otherwise, output character to UART
   B LOOP      ; Repeat the process for next character
EXIT  POP {R0, R1, PC}; Restore registers and Return
;- - - - - - - - - - - - - -
   Putc                    ; Subroutine to send a character via UART
                           ; Assume character to be sent is in R0
   PUSH {R1, R2, LR}; Save registers
   LDR R1,=UART0_FLAG; Get the address of UART flag
WAIT  LDR R2, [R1]     ; Get status flag
   TST R2, #0x20 ; Check transmit buffer full flag bit
   BNE WAIT      ; If busy then wait until transmit buffer is ready
   LDR R1, =UART0_DATA ; otherwise load the address of transmit buffer in R1.
   STRB R0, [R1]       ; Output data to transmit buffer
   POP {R1, R2}, PC} ; Restore registers and Return
;- - - - - - - - - - - - - - -
       UART0 Initialize
                           ; Device specific, not shown here
       BX LR               ; Return
;- - - - - - - - - - - - - - - - -
   HELLO_TXT
   DCB "Hello world\n", 0  ; Null terminated Hello world string
   END                     ; End of file
```

Write a program to display register contents i.e., hexadecimal value.

```
Puthex      ; output register value in hexa decimal format
            ; Assume value to be displayed in R0
PUSH {R0-R3, LR}   ; Save registers
MOV  R1, R0        ; Save value to be displayed in R1
MOV  R0, #'0'      ; Starting the display with "0x"
BL   Putc
MOV  R0, #'x'
BL   Putc
MOV  R3, #8        ; Initialize iteration counter
MOV  R2, #28       ; Rotate offset
L1 ROR  R1, R2     ; Rotate data value left by 4 bits (right 28)
AND  R0, R1, #0xF  ; Extract the lowest 4 bit
CMP  R0, #0xA      ; Convert to ASCII
ITE  GE
ADDGE  R0, #55     ; If larger or equal 10, then convert to A-F
ADDLT  R0, #48     ; otherwise convert to 0-9
BL   Putc          ; Output 1 hex character
SUBS R3, #1        ; decrement iteration counter
BNE  L1            ; Repeat if iteration counter is not zero
POP  {R0-R3, PC}   ; otherwise restore registers and return.
```

Write a program to display register value in decimal.

```
PutDec      ; Subroutine to display register value in decimal
            ; Assume value to be displayed in R0. Since it is 32 bit, the maximum number of character
            ; in decimal format, including null termination is 11.
PUSH {R0-R5, LR}   ; save register values
MOV  R3, SP        ; Copy current Stack Pointer to R3
SUB  SP, SP, #12   ; Reserved 12 bytes as text buffer
MOV  R1, #0        ; Null character
STRB R1, [R3, #-1]! ; Put null character at end of text buffer, pre-indexed
MOV  R5, #10       ; set divide value to decimal 10
DL UDIV R4, R0, R5 ; R4 = R0/10
```

```
        MUL   R1, R4, R5      ;  R1 = R4 * 10
        SUB   R2, R0, R1      ;  R2 = R0 - (R4 * 10) = remainder
        ADD   R2, #48         ; convert to ASCII (R2 can only be 0-9)
        STRB  R2, [R3, #-1]!  ; Put ASCII character in text buffer, pre-indexed
        MOVS  R0, R4          ; Set R0 = Divide result and set Z flag if R4=0.
        BNE   DL              ; If R0(R4) is already 0, then there is no more digit
        MOV   R0, R3          ; Put R0 to starting location of text buffer
        BL    Puts            ; Display the result using Puts
        ADD   SP, SP, #12     ; Restore stack location
        POP   {R0-R5, PC}     ; Restore registers and Return.
```

Write an assembly language program to calculate the sum of 1 to 10 numbers using DATA memory.

```
STACK_TOP EQU 0x20002000 ; constant for SP starting value
        AREA |Header Code|, CODE ; Code Area
        DCD  STACK_TOP    ; Stack top
        DCD  START        ; Reset vector
        ENTRY             ; Indicate program
START                     ; Start of the main program
        MOV  r0, #10      ; Initialize iteration counter
        MOV  r1, #0       ; Sum = 0
BACK    ADD  r1, r0       ;  Sum = Sum + R0
        SUBS r0, #1       ; Decrement iteration counter, R0 and update flag
        BNE  BACK         ; If count not zero continue addition.
        LDR  r0, = Sum1   ; Put address of MyData1 into R0
        STR  r1, [r0]     ; Store the result in MyData1
INFILOOP B   INFILOOP     ; Infinite loop
        AREA| Header Data|, DATA ; Data Area
        ALIGN 4
        SUM1 DCD  0       ; Memory words reversed to store result of
                          summation
        SUM2 DCD  0
        END               ; End of File.
```

MODULE - 3 EMBEDDED SYSTEM COMPONENTS.

B.S. BALAJI
ASST. PROF,
BGSIT.

What is an embedded System?

→ An embedded system is an electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).

→ It is unique, and the hardware as well as the firmware is highly specialised to the application domain like household appliances, telecommunications, medical equipment, industrial control, consumer products, etc.

Embedded Systems vs. General Computing Systems

Embedded System	General Purpose Computing System.
1) A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications.	1) A system which is a combination of a generic hardware and a general purpose Operating system for executing a variety of applications.
2) It may or may not contain an operating system for functioning	2) It contains a General Purpose Operating System (GPOS).
3) The firmware of the embedded system is pre programmed and it is non-alterable	3) Applications are alterable or programmable by the user.
4) Applications - specific requirements like performance, power requirements, memory usage etc. are the key deciding factors.	4) Performance is the key deciding factor in the selection of the system. Always, 'Faster is Better'.
5) Highly tailored to take advantages of the power saving modes supported by the hardware and the operating system.	5) Less / not at all tailored towards reduced operating power requirements, options for different levels of power management.
6) Response time requirement is highly critical for ~~time~~ certain category of embedded systems like mission critical systems.	6) Response requirements are not time-critical.

Embedded System	General Purpose Computing System.
7) Execution behaviour is deterministic for certain types of embedded systems like 'Hard Real time' systems.	7) Need not be deterministic in execution behaviour.
8) Differentiating features: Power, Cost, Size and Speed	8) Differentiating features: Speed, cost and software compatability
9) Runs a few applications often known at design time	9) Intended to run a fully general set of applications.
10) Not end user programmable	10) End-user programmable.
11) Response time requirement ~~is not~~ may be major not critical.	11) Response requirements are not timer critical.
12) Execution behaviour is deterministic	12) Execution behaviour is not deterministic

Classification of Embedded Systems

Embedded systems can be classified into 4 types. They are

1. Based on Generation
2. Based on Complexity and performance requirements
3. Based on deterministic behaviour
4. Based on triggering

1. Based on Generation. — (i) First Generation

→ 8bit microprocessors like 8085 and Z80 and 4bit microcontrollers were used to built early embedded systems.
→ Simple hardware circuits with ~~firmware~~ firmware developed in assembly code
→ Example - Digital telephone keypads, Stepper motor control units etc.

2. (ii) Second Generation

→ 16 bit microprocessors and 8or 16bit microcontrollers were used to built embedded systems for 2nd generation
→ The processors / controller instruction set were much more complex and powerful and also contained embedded operating systems for their operation

Example - Data Acquisition Systems, SCADA systems

(iii) Third Generation

→ With advances in technology, more powerful 32 bit processors and 16 bit microcontrollers are used in these embedded systems with a new concept of application and domain specific processors / controllers like DSPs and ASICs. It has dedicated ~~RTOS~~ real time and general purpose operating systems.

→ The instruction set of processors became more complex and powerful and the concept of instruction pipelining ~~also~~ also evolved.

→ Processors like Intel Pentium, Motorola 68K gained attention in high performance embedded requirements.

Example - Robotics, Media, Industrial process control, networking etc.

(iv) Fourth Generation -

→ The fourth generation embedded systems are making use of high performance real time embedded operating systems for their functioning.

→ The advent of Systems on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturization into the embedded device market.

→ SoC technique implements a 'total system on a chip' by integrating different functionalities with a processor core on an integrated circuit.

Examples - Smart phone devices, mobile internet devices (MIDs). etc

2. Classification based on Complexity and Performance

(i) Small-scale embedded systems -

→ Embedded systems which are simple in application needs and where the performance requirements are not time critical.

→ Small scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors / microcontrollers and may or may not contain an operating system for its functioning.

→ Example - An electronic toy.

(ii) Medium-scale Embedded Systems –

→ Embedded systems which are slightly complex in hardware and firmware (software) requirements fall under this category.

→ Medium-scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/microcontrollers or digital signal processors.

→ They contain an embedded operating system (like general purpose or real time operating system) for functioning.

(iii) Large-scale Embedded Systems / Complex systems –

→ It involves highly complex hardware and firmware requirements fall under this category. and used in mission critical applications demanding high performance.

→ These systems are commonly built around high performance 32 or 64 bit RISC processors/controllers or Reconfigurable System-on-Chip (RSoC) or multi-core processors and programmable logic devices.

→ It contains a high performance Real Time Operating System (RTOS) for task scheduling, prioritization and management.

→ It also contains multiple processors/ controllers and co-units / hardware accelerators for decoding / encoding of media, eg cryptographic function implementation ~~and~~ are examples for offloading the processing requirements from the main processor of the system.

3. Based on deterministic behaviour

→ Embedded systems based on deterministic behaviour is applicable for 'Real-Time' systems. Based on application or task execution behaviour, it can be either deterministic or non deterministic.

→ Based on execution behaviour, Real time embedded systems are classified into 'Hard real time systems' and 'Soft real time systems'.

4. Based on triggering

→ Embedded systems which are 'Reactive' in nature can be classified based on the trigger. Reactive systems can be either event triggered or time triggered.

example – Industrial control applications.

Major applications areas of Embedded Systems

→ Embedded technology has acquired a new dimension from its first generation model, the Apollo guidance computer to the latest radio navigation system combined with in-car entertainment (infotainment) technology and the microprocessor based "Smart" running shoes launched by Adidas in April 2005.

→ The application areas and the products in the embedded domain are listed below -

1. Consumer electronics - Camcorders, cameras, etc.
2. Household appliances - Television, DVD players, Washing machine, Refrigerator, microwave oven, etc.
3. Home automation and security systems - Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc
4. Automotive industry - Antilock breaking systems (ABS), ~~engin~~ engine control, ignition ~~control~~, systems, automatic navigation systems, etc
5. Telecom - Cellular telephones, telephone switches, handset multimedia applications, etc.
6. Computer peripherals - Printers, Scanners, Fax machines, etc.
7. Computer networking systems - Network routers, switches, hubs, firewalls, etc
8. Health care - Different kinds of scanners, EEG, ECG machines, etc.
9. Measurement & Instrumentation - Digital multi-meters, digital CROs, Logic analyzers PLC systems, etc.
10. Banking & Retail - Automatic teller machines (ATM) and currency counters, point of sales (POS).
11. Card Readers - Barcode, smart card readers, hand held devices, etc.

Purpose of Embedded Systems

- Each embedded system is designed to serve the purpose of any one or a combination of the following tasks -

1. Data collection/Storage/Representation
2. Data communication
3. Data (signal) processing
4. Monitor
5. Control
6. Application Specific User Interface

1) Data Collection / Storage / Requirements

→ Embedded systems designed for the purpose of data collection performs acquisition of data from the external world.
→ Data collection is usually done for storage, analysis, manipulation and transmission.
→ Data can be either analog (continuous) or digital (discrete). It refers all kinds of information, like text, voice, image, video, electrical signals and any other measurable quantities.
→ Embedded systems with analog data capturing techniques collect data directly in the form of analog signals.
→ Embedded systems with digital data collection mechanism converts the analog signal to corresponding digital signal using analog to digital (A/D) converters and then collects the binary equivalent of the analog data.
→ The collected data may be stored directly in the system (or) may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly after giving a meaningful representation.
→ It is designed for pure measurement applications without storage, used in control and instrumentation domain, collects data and gives a meaningful representation of the collected data by means of graphical representation (or quantity value)
→ The collected data is deleted when new data arrives at the data collection terminal.
→ Analog and Digital CROs without storage memory are typical examples cosed as measuring equipment used in the medical domain for monitoring without storage functionality.
→ Embedded systems store the collected data for processing and analysis. It incorporates a built-in / plug-in storage memory for storing the capture data and a meaningful representation of the collected data by visual or audible means using display units (LCD, LED etc), instruments with storage memory and monitoring

Examples - 1) Measuring instruments with storage memory used in medical applications.

2) A digital camera is a typical example of an embedded system with data collection/storage/representation of data.

Images are captured and the captured image may be stored within the memory of the camera. Images are presented to the user through a graphic LCD unit.

2) Data Communication –

→ Embedded data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems.

→ The data collected by an embedded terminal may require transferring of the same to some other system located remotely.

→ The transmission is achieved either by a wire-line medium or by a wire-less medium.

→ Wire-line medium was the most common choice in all olden days and as technology is changing, wireless medium is becoming the de-facto standard for data communication in embedded systems.

→ A wireless medium offers cheaper connectivity solutions and make the communication link free from the hassle of wire bundles.

→ Data can either be transmitted by analog means or by digital means.

Example – Wireless modules – Bluetooth, ZigBee, Wi-Fi, EDGE, GPRS etc and Wired-line module (RS232-C, USB, TCP/IP, PS2, etc).

→ Certain embedded system acts as a dedicated transmission unit between the sending and receiving terminals, offering special functions like data packetizing, encrypting and decrypting.

example- Network hubs, routers, switches, etc.

→ They acts as mediators in data communication and provide various features like data security, monitoring etc.

3) Data (Signal) Processing –

→ The data (voice, image, video, electrical signals and other measurable quantities) collected by embedded systems may be used for various data processing.

→ Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, synthesis, audio video codec, transmission applications etc.

Example- Digital hearing aid is an embedded system employing data processing. It improves the hearing capacity of hearing impaired persons.

4) Monitoring -

→ Embedded systems are specifically designed for monitoring purpose like embedded products coming under the medical domain are with monitoring functions only.

→ It is used to determine the state of some variables using input sensors. It cannot impose control over variables.

Example - 1) Electro cardiogram (ECG) machine for monitoring the heartbeat of a patient. It cannot impose control over the heart beat.

The sensors used in EGG are the different electrodes connected to the patient's body.

2) Embedded systems with monitoring functions are measuring instruments like Digital CRO, digital multimeters, logic analyzers used in Control & Instrumentation applications. ~~It is~~ These are used to monitor variables like current, voltage etc. They cannot control the variables.

5) Control -

→ Embedded systems with control functionalities impose control over some variables acc to changes in input variables.

→ A system with control functionality contains both sensors and actuators.

→ Sensors are connected to the input port for capturing the changes in environmental variables or measuring variables.

→ The Actuators connected to the output port are controlled acc to the changes in input variables to put an impact on the controlling variable to bring the controlled variable to the specified range.

→ Example- Air conditioner system is used to control the room temperature to a specified limit for control purpose.

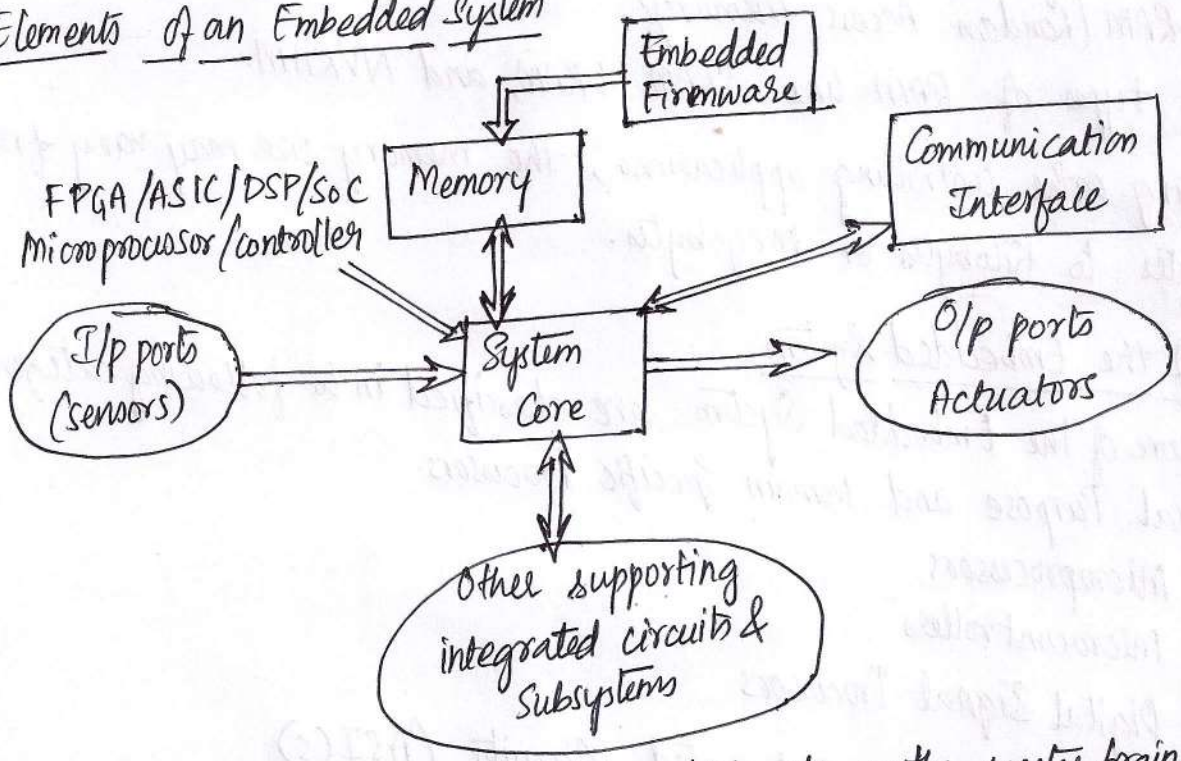
* It contains a room temperature sensing element (sensor) like thermistor.
* ~~and~~ hand held unit is used for setting up the desired temperature. and it is connected to central embedded unit residing inside the air conditioner through a wireless link or through a wired link.
* Air compressor acts as a actuator, it controls the current room temperature and the desired temperature set by the end user.

6) Application Specific User Interface

Embedded systems comes with application specific user interfaces like buttons, switches, keypad, lights, bells, display units etc.

Example - Mobile phone were user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert etc

Elements of an Embedded System



→ It contains a single chip controller which acts as the master brain of the system. like Microprocessor, Microcontroller, FPGA, ASIC, DSP, SoC etc

→ Embedded hardware/software systems are basically designed to regulate a physical variable or to manipulate the state of some devices by sending some control signals to the Actuators ~~or dt~~ connected to the o/p ports of the system, in response to the i/p signals provided by the end users or Sensors which are connected to the input ports.

→ Keyboards, Push button switches, etc are example for common user interface i/p devices where as LEDs, LCD displays, piezo electric buzzers, etc are examples for common user interface output devices for a typical embedded system.

→ The memory of the system is responsible for holding the control algorithm and other important configuration details.

Program memory

→ The memory, storing the algorithm or configuration data is of fixed type is Read Only Memory (ROM) and is not available for the end user for modifications and is protected from unwanted user interaction. like OTP, PROM, UVEPROM, EEPROM and FLASH. also called as Program Memory.

→ The system requires temporary memory for performing arithmetic operations or control algorithm execution and it is called as "Working memory" or RAM (Random Access Memory).

→ Various types of RAM like SRAM, DRAM, and NVRAM.

→ Depending on the controlling applications, the memory size may vary from a few bytes to Kilobytes or Megabytes.

Core of the Embedded System

The core of the Embedded System are classified in to following categories-

1. General Purpose and Domain Specific Processors.
   - 1.1. Microprocessors
   - 1.2 Microcontrollers
   - 1.3 Digital Signal Processors
2. Application Specific Integrated Circuits (ASICs)
3. Programmable logic Devices (PLDs)
4. Commercial off-the-shelf Components (COTS).

1) General Purpose and Domain Specific Processors.

1.1. Microprocessors

→ It is a silicon chip representing a Central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to predefined set of instructions.

→ The CPU contains the Arithmetic and Logic Unit (ALU), control unit and working registers.

→ It is a dependent unit which requires the combination of other hardware like memory, timer unit, and interrupt controller etc for proper functioning.

→ Intel developed the first microprocessor unit Intel 4004 a 4bit microprocessor in November 1971.
→ It featured 1K data memory, a 12 bit program counter and 4K program memory, sixteen 4bit general purpose registers and 46 instructions.
→ It works with a clock speed of 740KHz. It is used in Calculators.
→ In April 1974, Intel launched first 8bit, the Intel 8080 with 16bit address bus and program counter and seven 8bit registers (A-E, H,L; BC, DE and HL pairs formed the 16 bit register) ~~for this~~
→ Intel 8080 was the most commonly used processors for industrial control and other embedded applications in the 1975s.
→ ~~Intel~~ Motorola also ~~entered~~ introduced a processor-Motorola 6800 with a different architecture and instruction set ~~in the market~~ compared to Intel 8080.
→ In 1976, Intel ~~up~~ came up with the upgraded vision of 8080, Intel 8085 with two newly added instructions, three interrupt pins and serial I/o.
→ Clock generator and bus controller circuits were built-in and the power supply part was modified to a single +5V. supply.
→ In July 1976, Zilog introduced a Z80 processor with the original 8080 architecture and instruction set with an 8bit data bus and a 16 bit address bus.
→ It has two sets of index registers for flexible design and also includes 80 more new instructions and ~~it~~ new concept of register banking by doubling ~~the~~ the register set.
→ It is capable of executing of all 8 instructions of 8080 processor.
→ Processors based on Harvard architecture contains separate buses for program memory and data memory, where as processors based on Von-neumann architecture shares a single ~~bus~~ system bus for program and data memory.
→ ~~Hard~~ Harvard architecture and Von-Neumann architecture are the two common system architectures for processor design.

2) General Purpose Processor (GPP) vs Application-Specific Instruction Set Processor

→ A GPP is a processor designed for general computational tasks. example- Pentium4 / AMD Athlon etc. They are produced in large volumes and targeting the market where as the per unit cost for a chip is low compared to ASICs.

→ It contains Arithmetic Logic Unit (ALU) and Control Unit (CU).

→ ASIPs are processors with architecture and instruction set optimised to specific-domain/application requirements like network processing, automotive, telecom, media applications, digital signal processing, control applications

→ It fills the spectrum between GPPs and ASICs. The need for ASIPs arises when the traditional GPP are unable to meet the increasing application need.

→ It incorporates a processor and on-chip peripherals, demanded by the application requirement, program and data memory.

3) Microcontrollers

→ It is a highly integrated chip that contains a CPU, scratchpad RAM, special and general purpose registers arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports.

→ It can be considered as a superset of Microprocessors as it contains all necessary functional blocks for independent working in the embedded domain.

→ It is cheap, cost effective and are readily available in the market.

→ Texas Instrument's TMS1000 is considered as the world's first microcontroller.

→ TMS1000 had 4040, 4 bit processor design and added some amount of RAM, program storage memory (ROM) and I/O support on a single chip, ~~thereby~~ which eliminated the requirement of multiple hardware chips for self-functioning.

→ In 1980, Intel introduced 8 bit Microcontroller domain, the 8051 family which is the most popular and powerful microcontroller under the family MCS-51.

→ Another important family of microcontrollers used in Industrial control and embedded applications in the PIC ~~family~~ micro controllers from micro chip Technologies.

→ It

→ It is a high performance RISC microcontroller complementing the CISC features of 8051.

→ The instruction set architecture of a microcontroller can be either RISC or CISC.

→ Microcontrollers are designed for either general purpose application requirement (general purpose controller) or domain specific application requirement (application specific instruction set processor).

Microprocessor vs Microcontroller

Microprocessor	Microcontroller
1. A Silicon chip representing a central processing unit (CPU) which is capable of performing arithmetic as well as logical operations acc to a pre-defined set of instructions.	1. A Microcontroller is a highly integrated chip that contains a CPU, scratchpad RAM, special and general purpose registers arrays, on chip ROM/ FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports
2. It is a dependent unit.	2. It is a self-contained unit.
3. It requires the combination of other chips like timers, program and data memory chips, interrupt controllers, etc for functioning.	3. It doesn't require external interrupt controller, timer, UART etc, for its functioning
4. Most of the time general purpose in design and operation.	4. Mostly application-oriented or domain specific.
5. It doesn't contain a built-in I/O port	5. It contains multiple built-in I/O ports.
6. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255.	6. I/O ports can be operated as a single 8 or 16 or 32 bit port or as individual port pins.
7. Targeted for high end market where Performance is important.	7. Targeted for embedded market where performance is not so critical.
8. Limited power saving options to microcontrollers.	8. Includes a lot of power saving features.

5) Digital Signal Processors (DSPs)

→ DSPs are powerful special purpose 8/16/32 bit microprocessors designed specifically to meet the computational demands and power constraints like embedded audio, video, and communications applications

↔ It is 2 to 3 times faster than the general purpose microprocessors in signal processing applications.

→ It is a microchip designed for performing high computional operations for 'addition', 'subtraction', 'multiplication' and 'division'

→ DSPs implement algorithms in hardware which speeds up the execution where GPPs implement the algorithm in firmware and the speed of execution depends primarily on the clock for the ~~purpose~~ processors.

→ It consists of the following key units -

Program Memory - Memory for storing the program required by DSP to process the data.

Data Memory - Working memory for storing temporary variables and data / ~~sing~~ signal to be processed.

Computational Engine - Performs the signal processing in accordance with the stored program memory.

It consists of specialised ~~ari~~ arithmetic units and each ~~A thum~~ operates simultaneously to ~~th~~ increase the execution speed. and also multiple hardware shifters for shifting operands and saves execution time.

I/O Unit - Act as an interface b/w the DSP and outside world. It is used for capturing signals to be processed and delivering the processed signals.

Examples - Audio, video signal processing, telecommunication and multimedia applications

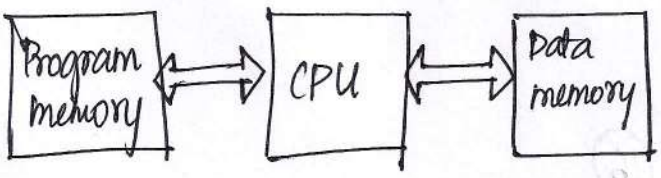
→ It employs a large ~~amount~~ amount of real time calculations like Sum of Products (SOP) calculation, Convolution, Fast fourier transform (FFT), Discrete Fourier transform etc.

6) RISC vs CISC Processors / Controllers

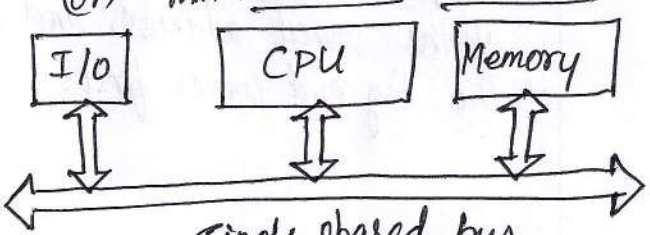
RISC (Reduced Instruction set Computing)	CISC (Complex Instruction Set Computing).
1. Lesser no. of instructions	1. Greater no. of instructions.
2. Instruction pipelinging and increased execution speed.	2. Generally no instruction pipelinging feature
3. It has orthogonal instruction set	3. It has non-orthogonal instruction set
4. It allows each instruction to operate on any register and use any addressing mode.	4. It allows operations to be performed on registers or memory depending on the instruction.
5. A large number of registers are available	5. Limited no. of general purpose registers.
6. Programmer needs to write more code to execute a task since the instructions are simpler ones.	6. Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC.
7. Single, fixed length instructions	7. Variable length instructions
8. Less silicon usage and pin count	8. More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
9. With Harvard architecture	9. Can be Harvard or Von-Neumann Architecture.

7) Harvard vs Von-Neumann Processor / Controller Architecture

Harvard architecture



Von-Neumann Architecture
(or) Princeton Architecture



Harvard architecture	Von-Neumann architecture
2. Separate buses for instruction and data fetching	2. Single shared bus for instruction and data fetching
3. Easier to pipeline, so high performance can be achieved	3. Low performance compared to Harvard architecture
4. It is comparatively high cost	4. It is low cost.
5. It has no memory alignment problems	5. It allows self modifying codes
6. data memory and program memory are stored physically in different locations	6. Data memory and Program memory are stored physically in the same chip.
7. There is no chances for accidental corruption of program memory.	7. There is chances for accidental corruption of program memory.

8) Big-Endian vs Little-Endian Processors / Controllers

→ Endianess specifies the order in which the data is stored in the memory by processor operations in a multibyte system (processors whose word size is greater than one byte).

1) Little-endian means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address.

→ The little end comes first.

For example- a 4 byte long integer Byte 3 Byte 2 Byte 1 Byte 0 will be stored in the memory as shown below.

Base Address + 0	Byte 0	Byte 0	0x20000 (Base address)
Base Address + 1	Byte 1	Byte 1	0x20001 (Base address + 1)
Base Address + 2	Byte 2	Byte 2	0x20002 (Base address + 2)
Base Address + 3	Byte 3	Byte 3	0x20003 (Base address + 3)

2) Big-endian means the higher order byte the data is stored in memory at the lowest address, and the lower-order byte at the highest address.

→ The big end comes first.

For example, a 4 byte long integer Byte3 Byte2 Byte1 Byte0 will be stored in the memory as follows-

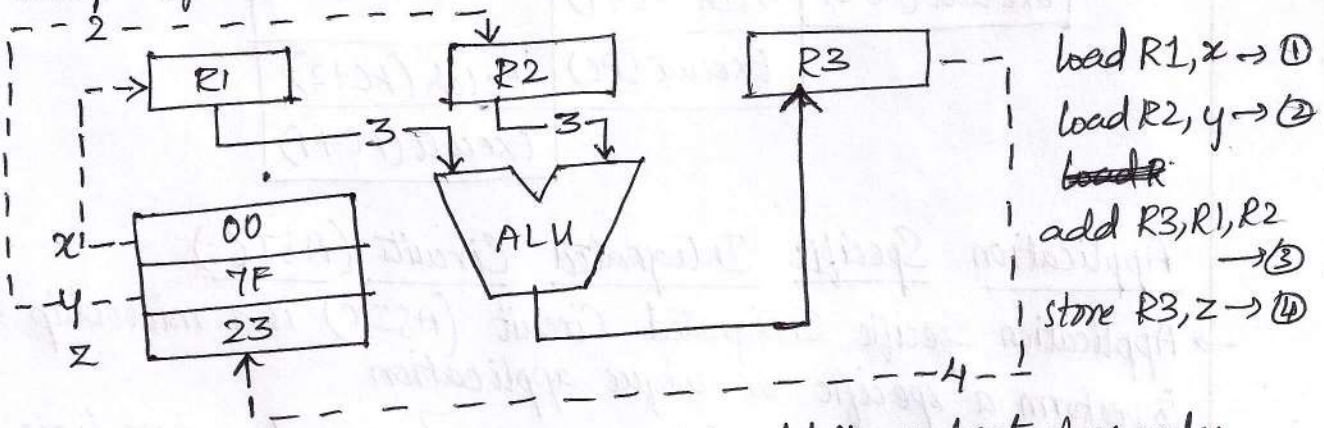
Base Address +0	Byte 3	Byte 3	0x20000 (Base address)
Base Address +1	Byte 2	Byte 2	0x20001 (Base address+1)
Base Address +2	Byte 1	Byte 1	0x20002 (Base address+2)
Base Address +3	Byte 0	Byte 0	0x20003 (Base address+3)

9) Load Store Operation and Instruction Pipelining

In RISC processor, the memory access related operations are performed by special instructions load and store.

→ If the operand is specified as memory location, the content of it is loaded to a register using the load instruction. The instruction store stores data from a specified register to a specified memory location.

→ The concept of Load Store Architecture as shown in figure.



→ Suppose x, y, and z are memory locations and to add the contents of x and y store the result in location z. Using the load store architecture is achieved with 4 instructions.

→ The first instruction load R1,x loads the register R1 with the content of memory location x, the second instruction load R2,y loads R2 with the content of memory location y.

→ The add R3, R1, R2 adds the content of registers R1 and R2 and stores the result in register R3.

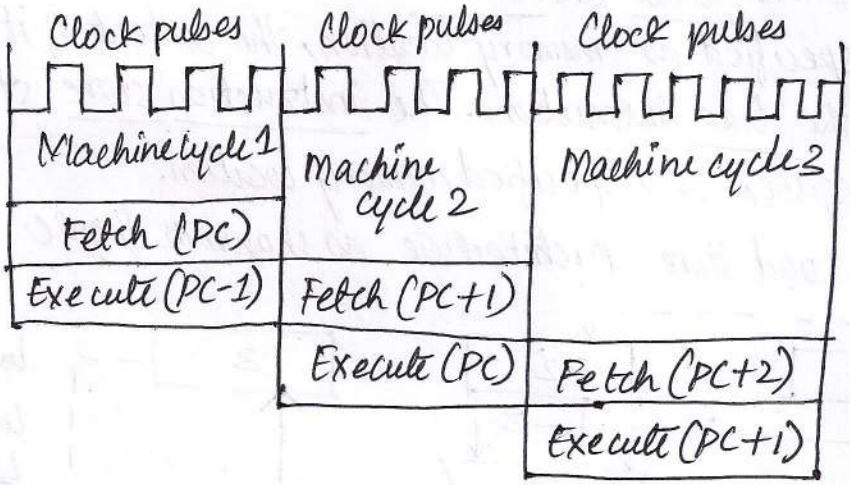
→ The next instruction store R3,z stores the content of register R3 in memory location.

Instruction Pipelining

→ It refers to the overlapped execution of instructions. where the normal program execution flow indicates to fetch the next instruction to execute, while decoding and ~~executing~~ execution of the current instruction is in progress.

→ Depending on the stages involved in an instruction (fetch, read register and decode, execution instruction, access an operand in data memory, write back the result to register, etc.) there can be multiple levels of instruction pipelining

→ The concept of Instruction pipelining for single stage pipelining.



Application Specific Integrated Circuits (ASICs)

→ Application Specific Integrated Circuit (ASIC) is a microchip designed to perform a specific or unique application

→ It is used as replacement to conventional general purpose logic chips

→ It integrates all functions into a single chip and reduces the system development cost.

→ It consumes a very small area in the total system and helps in the design of smaller systems with high ~~capabi~~ capabilities / functionalities.

→ It can be pre fabricated for a special application (or) custom fabricated by using the components from a re-usable 'building block' library of components for a particular customer application.

→ ASIC based systems are profitable only for large volume commercial productions where its fabrication requires a non refundable initial investment for the process technology and configuration expenses.
→ It is a one-time investment also called as Non Recurring Engineering Charge (NRE)
→ If it is borne by a third party, ASIC is referred as Application Specific Standard Product (ASSP) is made openly available in the market.
→ ASSP is marketed to multiple customers ~~not~~ as general-purpose product but to a smaller no. of customers since it is for a specific application.
Example - ADE7760 Energy metre ASIC developed by Analog Devices for energy metering applications.

Programmable Logic Devices

→ Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and other functions a system must perform.
→ It can be classified into two categories - fixed and programmable logic devices.
→ Fixed logic devices are circuits with permanent and fixed logic which cannot be changed. They perform one function or set of functions - once manufactured, ~~th~~ they cannot be changed.
→ Programmable logic devices (PLDs) offers a wide range of logic capacity, features speed, and voltage characteristics and can be re-configured to perform any no. of functions at any time.
→ A design can be quickly programmed into a device where it uses inexpensive software tools to quickly develop, simulate and test their designs as a ~~live~~ live circuit.
→ It will be used for prototyping the same PLD as the final production of a piece of end equipment such as a network router, a DSL modem, a DVD player, or an automotive navigation system.

→ No NRE costs and final design is completed much faster than that of a custom, fixed or logic device

→ The benefit ~~is~~ of using PLDs, during the design phase, customers can change the circuitry as often as until the design operates to their satisfaction.

→ PLDs are based on re-writable memory technology - to change the design, the device is simply reprogrammed.

CPLDs and FPGAs

The 2 major types of Programmable logic devices are Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs).

→ FPGAs offers the highest amount of logic density, the most features, and the highest performance.

→ It also offer features such as built-in hardwired processors (like IBM Power PC), substantial amounts of memory, clock management system and support for many of the latest, very fast device-to-device signaling technologies.

→ FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing.

→ FPGAs are especially popular for prototyping ASIC designs where the designer can test his design by downloading the design file into an FPGA device

→ After the design is set, Hardwired chips are produced for faster performance. Example - Xilinx Virtex™ provides eight million "system gates" (the relative density of logic)

→ CPLDs offer much smaller amounts of logic - up to about 10,000 gates.

→ It offer very predictable timing characteristics and it is ideal for critical control applications.

Example - Xilinx CoolRunner™ require extremely low amounts of power and very inexpensive; and.

→ It is ideal for cost-sensitive, battery operated, portable applications such as mobile phones and digital handheld assistants.

Advantages of PLDs

1) It offers customers much more flexibility during the design cycle.
2) It do not require long lead times for prototypes or production parts.
3) It do not require customers to pay for large NRE costs and purchase expensive mask sets.
4) It allow customers to order just the no. of parts they need, when they need them allowing them to control inventory to avoid short of parts & production delays.
5) It can be reprogrammed even after a piece of equipment is shipped to a customer.

Commercial Off-the Shelf Components (COTS)

→ It is a product which is used 'as-is' & are designed in such a way to provide easy integration and interoperability with existing system components.
→ It may be developed around a general purpose or domain specific processor or an Application Specific Integrated circuit or a programmable logic device.
→ The major advantage is that they are readily available in the market, are low cost and a developer can cut down his/her development time to a great extent.
→ It reduces the time to market your embedded systems.

Examples - TCP/IP plug-in module available from various manufactures like 'WIZnet', 'Freescale', 'Dynalog', etc
→ COTS hardware unit are remote controlled toy car control units including the RF circuitry part, high performance, high frequency micro wave electronics (2-200GHz), high bandwidth ADC, devices and components for operation at very high temperatures, electro-optic IR imaging arrays, UV/IR Detectors.
→ Network plug-in module gives TCP/IP connectivity to the system which is developed and no need to design and write the firmware for the TCP/IP protocol and data transfer.

→ Multiple vendors supply COTs for the same application, the major problem faced by the end user ~~is that~~ there are no operational and manufacturing standards

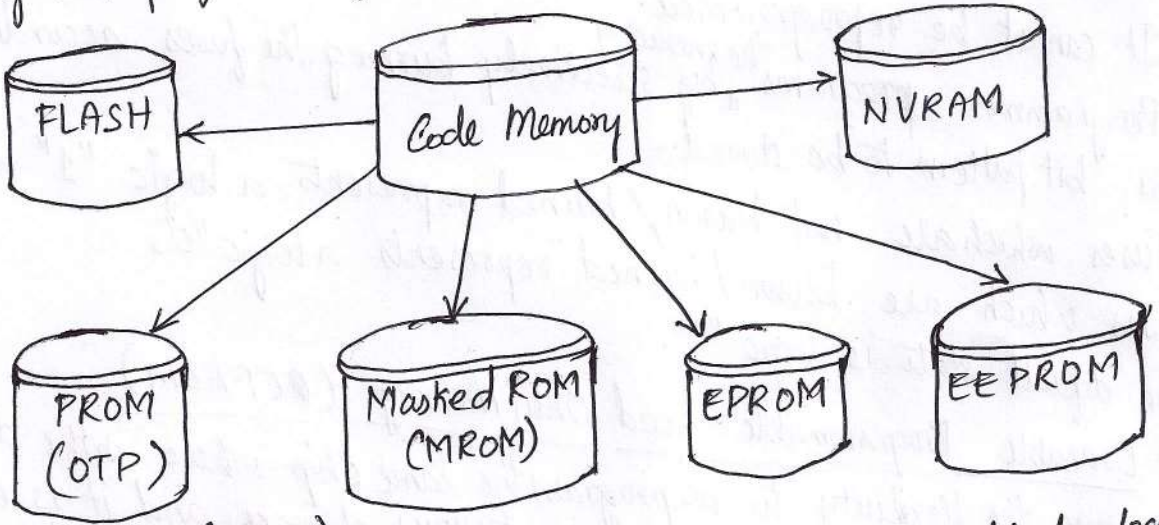
→ The major drawback ~~of~~ COTs used in embedded system design is that Manufacturer of the COTs component may withdraw the product or discontinue the production when there is a rapid change in technology occurs and it will adversely affects a commercial manufacturer of the embedded system.

Memory

→ It is an important part of processor / controller based embedded systems.
→ Processor / controllers contain built-in and this memory is referred as on-chip memory.
→ It does not contain any memory inside the chip and requires external memory to be connected with the controller / processor to store the control algorithm called as Program Storage Memory. (or) Off-chip memory. (ROM)
→ It also needs some working memory is required for holding data temporarily during certain operations. It is called as RAM. (Random Access Memory).

1) Read Only Memory (ROM) (or) Program Storage Memory

→ The program memory or code storage memory of an embedded system stores the program instructions.
→ It retains its contents even after the power to it is turned off. It is known as non-volatile memory.
→ They are classified into the following types depending on the fabrication, erasing and programming



(i) Masked ROM (MROM) –

→ It is a one-time programmable device. It uses hardwired technology for storing data. It is permanent in bit storage, it is not possible to alter the bit information.
→ It is factory programmed by masking and metallisation process at the time of production as per the data provided by end user.
→ The primary advantage is low cost for high volume production. They are least expensive type of solid state memory.
→ The limitation is the inability to modify the device firmware against firmware upgrades.

Different mechanisms are used for the masking process of the ROM, like
1) Creation of an enhancement or depletion mode transistor through channel implant.
2) By creating the memory cell either using a standard transistor or a high threshold transistor.
→ In the high threshold mode, the supply voltage required to turn ON the transistor is above the normal ROM IC operating voltage.
→ It ensures that the transistor is always OFF and the memory cell stores always logic 0.

(ii) Programmable Read Only Memory (PROM) / One Time Programmable Memory (OTP)
→ One Time Programmable Memory (OTP) or PROM is not preprogrammed by the manufacturer. The end user is responsible for programming these devices.
→ It has nichrome or polysilicon wires arranged in a matrix. It can be functionally viewed as fuses.
→ OTP is widely used for commercial production of embedded systems as it is low cost solution and the code is finalised.
→ It cannot be reprogrammed.
→ Programmer programs the memory by selectively burning the fuses according to the bit pattern to be stored.
→ Fuses which are not blown / burned represents a logic "1" whereas fuses which are blown / burned represents a logic "0".
→ The default state is logic "1".

(iii) Erasable Programmable Read Only Memory (EEPROM)
→ It gives the flexibility to re-program the same chip where in the development phase of the code is subject to continuous changes and it is economical.
→ It stores the bit information by charging the floating gate of an FET. through applying high voltage to charge the floating gate.
→ It contains a quartz ~~window~~ crystal window for erasing the stored information; ~~when~~ when exposed to UV rays eraser device for a fixed duration of 20 to 30 minutes, entire memory will be erased.
→ It is a tedious and time-consuming process.

(iv) Electrically Erasable Programmable Read only Memory (EEPROM).

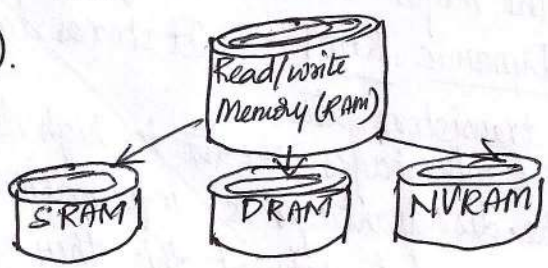
→ It can be altered by using electrical signals at the register / Byte level.
→ It can be erased and reprogrammed in-circuit where the chip includes a chip erase mode and erases the memory in a few milliseconds.
→ It provides greater flexibility for system design.
→ It has limited capacity when compared with the standard ROM (a few kilobytes)

(v) FLASH

→ It is the latest and most popular ROM technology.
→ It combines the re-programmability of EEPROM and the high capacity of standard ROMs.
→ It is organised as sectors (blocks) or pages.
→ It stores information in an array of floating gate of MOSFET Transistors.
→ The erasing of memory can be done at sector level or page level without affecting the other sectors or pages.
→ Each sector/page should be erased before re-programming. The typical erasable capacity of FLASH is 1000 cycles.

2) Read-Write Memory / Random Access Memory (RAM)

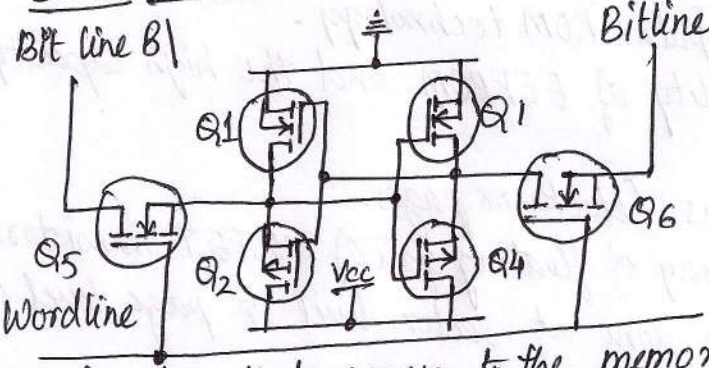
→ It is the data memory or working memory of the controller/processor where Controllers/Processors can read or write from it.
→ It is volatile means when the power is turned off all the contents are destroyed.
→ RAM is a direct access memory which is in contrast to the Sequential Access Memory (SAM), where the desired memory location is accessed by either traversing through the entire memory or through a 'seek' method.
→ It is classified into three types - Static RAM (SRAM), Dynamic RAM (DRAM) and non-volatile RAM (NVRAM).



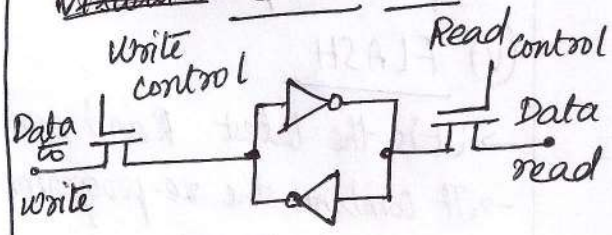
(i) Static RAM (SRAM)

→ It stores the data in the form of voltage. They are made up of flip-flops.
→ It is the fastest form of RAM available.
→ SRAM cell is realised using six transistors (or 6 MOSFETs).
Four of the transistors are used for building the latch (flip-flop) part of the memory cell and two for controlling the access.
→ SRAM is fast in operation due to its resistive networking and switching capabilities.

SRAM cell implementation



SRAM Cell

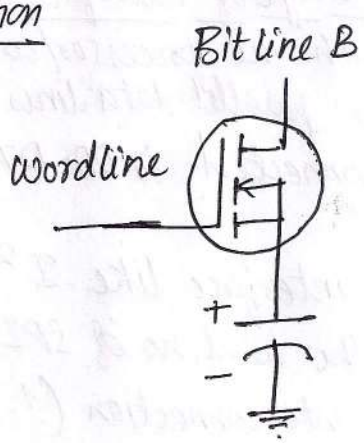


→ It is clear that access to the memory cell is controlled by the Wood line, which controls the access transistors Q5 and Q6.
→ The access transistors control the connection to bit lines B & B\.
→ In simpler form can be visualised as two-cross coupled inverters with read/write control through transistors.
→ The four transistors in the middle form the cross-coupled inverters.
→ To write a value to the memory cell, apply the desired value to the bit control lines (For writing = 1, make B=1 and B\=0; for writing =0, make B=0 and B\=1). and assert the Word line (make the word line high).
→ For reading the content of the memory cell, assert both B and B\ bit lines to 1 and set the Word line to 1.

→ The major limitations of SRAM are low capacity and high cost.

(ii) Dynamic RAM – It stores data in the form of charge and are made up of MOS transistors gates.
→ The advantages is high density and low cost compared to SRAM.
→ The disadvantages is information stored as charge it gets leaked off with time and to prevent this they need to be refreshed periodically.

DRAM cell implementation



→ Special circuits called DRAM controllers are used for the refreshing operation.
→ The refresh operation is done periodically in milliseconds interval.
→ The MOSFET acts as the gate for the incoming and outgoing data where as the capacitor acts as the bit storage unit.

Comparison SRAM cell V/s DRAM cell

SRAM Cell	DRAM cell
1. Made up of 6 CMOS transistors (MOSFET)	1. Made up of a MOSFET and a capacitor
2. Doesn't require refreshing	2. Requires refreshing
3. Low capacity (less dense)	3. High capacity (highly dense)
4. More expensive	4. Less expensive
5. Fast in operation. Typical access time is 10ns.	5. Slow in operation due to refresh requirements. Typical access time is 60ns.
	6. Write operation is faster than read operation.

NVRAM - Non-volatile RAM is a random access memory with battery backup.
→ It contains static RAM based memory and a minute battery for providing supply to the memory in the absence of the external power supply.
→ The memory and battery are packed together in a single package
→ It is used for the non-volatile storage of results of operations or for setting up of flags etc.
→ The life span of NVRAM is expected to around 10 years

Memory according to the type of Interface -

→ The interface of memory with the processor/controller can be of various types like a parallel interface (where parallel data lines (D0-D7) for an 8bit processor/controller will be connected to D0-D7 of the memory).

(or)

the interface may be a serial interface like I^2C, SPI (2+n line interface) (2 line serial interface).
where n stands for the total no. of SPI device in the system).

→ It can also be a single wire interconnection (1-wire interface)

→ Serial interface is commonly used for data storage memory like EEPROM.

→ The memory density of a serial memory is usually expressed in terms of Kilobits whereas a parallel interface memory is expressed interms of kilobytes.

Example- Atmel AT24C512 has serial memory with capacity 512 kilobits and 2-wire interface.

Memory Shadowing

→ Generally the execution of a program or a configuration from a Read only Memory (ROM) is very slow (120 to 200ns) compared to the execution from random access memory (40 to 70ns)

→ RAM access is about three times as fast as ROM access.

→ Shadowing of memory is a technique adopted to solve the execution speed problem in processor-based systems

→ In computer systems, there will be a configuration holding ROM called Basic Input Output Configuration ROM (or) Simply BIOS.

→ BIOS stores the hardware configuration information like the address ~~signal~~ assigned for various serial ports and non-plug 'n' play devices etc.

→ It is read operation and the system is configured acc to it during system boot up. It is time consuming.

→ The manufacturers included a RAM behind the logical layer of BIOS at its same address as a shadow to the BIOS and the first step is to copy the BIOS to the shadowed RAM and write protecting the RAM then disabling the BIOS reading, during the boot up.

→ For high system performance, it should be accessed from a RAM instead of accessing from a ROM. RAM is volatile and it cannot hold the configuration data which is copied from BIOS when the power supply is switched off. Only a ROM can hold it permanently.

Memory Selection for Embedded Systems

→ Embedded Systems require a program memory for holding control algorithm or embedded OS, data memory for holding variables and temporary data during task execution, and memory for holding non-volatile data which are modifiable by the application (or) program memory which is non-volatile as well unalterable by the user.

→ The memory requirement for an embedded systems in terms of RAM and ROM (FLASH/EEPROM/NVRAM) is dependent on the type of the embedded system design.

→ Embedded system is designed using SoC or a microcontroller with on-chip RAM and ROM (EEPROM/FLASH), depends on the application needs the on-chip memory may be sufficient for designing the total system.

→ As a rule of thumb, identify your system requirement and based on the type of processor used for the design, take a decision on whether the on-chip memory is sufficient or external memory is required.

Example - a simple electronic toy design where as the complexity of requirements are less and data memory requirement are minimal where a Microcontroller with a few bytes of internal RAM, a few Kilobytes of FLASH and a few bytes of EEPROM is used for designing the system.

→ A PIC microcontroller device which satisfies the I/O and memory requirements can be used.

→ In RTOS based embedded system design, it requires certain amount of RAM for its execution and ROM for storing the RTOS image where the image of RTOS is a binary code for RTOS kernel containing all its services is stored in a non-volatile memory (FLASH memory) as either compressed or non-compressed data.

→ During the boot up of the device, the RTOS files are copied from the program storage memory (ROM), decompressed if required and then loaded to the RAM for execution.

→ The supplier of RTOS gives a rough estimate on the run time RAM requirements and program memory requirements for the RTOS.

→ Memory chips comes in standard sizes like 512 bytes, 1024 bytes (1 kilobyte), 2048 bytes (2 kilo byte), 4Kb, 8Kb, 16Kb, 32Kb, 64Kb, 128 Kb, 256Kb, 512Kb, 1024 Kb (1 mega bytes) etc.

→ In case of an embedded system requires only 750 bytes of RAM, there is no choice of getting a memory chip with a size of 750 bytes, the only option to get is to select the memory chip with a size closer to the size needed.

→ Example 1024 bytes ~~is the least possible and~~ is the best option while 512 bytes is not possible as 750 bytes is the minimum requirements.

→ FLASH memory is the popular choice of ROM (program memory) and is a powerful and cost-effective solid-state storage technology for mobile electronics devices and other consumer applications.

→ It has two types - NAND FLASH and NOR FLASH.

→ NAND FLASH is a high density low cost non-volatile storage memory.

→ NOR FLASH is less dense and slightly expensive and it supports the Execute in Place (XIP) technique for program execution.

→ The XIP technology allows the execution of code memory from ROM itself without the need for copying it to the RAM.

→ NAND Flash doesn't support XIP and if it is used for storing program code a DRAM can be used as the ~~memory for bootloader or for even storing~~ copying and executing the program code.

→ NOR Flash supports XIP and it can be used as the memory for bootloader or storing the complete program code.

→ EEPROM data storage memory is available as serial or parallel interface chip. The processor/controller of the device supports serial interface and the amount of data to write and read to and from the device is less, it is better to have a serial EEPROM chip.

→ It saves the address space of the total system.

→ The ~~memory~~ serial EEPROM memory capacity is usually expressed in bits or kilobits like 512 bits, 1K bits, 2K bits, 4K bits etc.

SENSORS AND ACTUATORS

Sensors

A Sensor is a transducer device that converts energy from one form to another for any measurement or control purpose.

Actuators

Actuator is a form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). It acts as an output device.

The I/O Subsystem

→ The I/O Subsystem of the embedded system facilities the interaction of the embedded system with the external world.

→ It is the interaction happens through the sensors and actuators connected to the input and output ports respectively of the embedded system.

→ The sensors may not be directly interfaced to the input ports, instead they may be interfaced through signal conditioning and translating systems like ADC, optocouplers, etc.

In Embedded Systems, Sensors and Actuators used and the I/O Subsystems to facilitate the interaction of embedded systems with external ~~world~~ world.

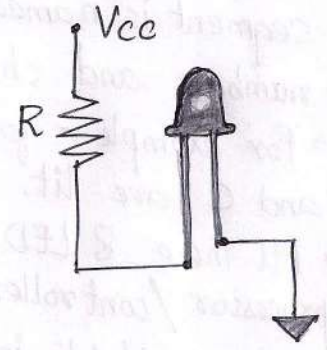
LIGHT EMITTING DIODE (LED)

→ LED is an important ~~device~~ output device for visual indication in an embedded system.

→ It can be used as an indicator for the status of various signals or signals. examples - 'Device ON', 'Battery low' or 'Charging of battery' in embedded devices.

→ It is a p-n junction diode where it contains an anode and a cathode.

→ The LED interfacing circuit



* The anode terminal should be connected to +ve terminal of the supply voltage and
* cathode terminal should be connected to -ve terminal of the supply voltage.
* The current flowing through the LED must be limited to a value below the maximum current that it can conduct.
* A resistor is used in series between the power supply and the LED to limit the current through the LED.

LEDs can be interfaced to the port pin of a processor/controller in 2 ways –

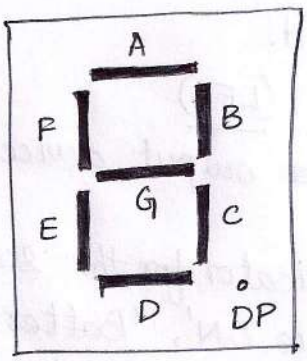
→ In the first method, the anode is directly connected to the port pin and the port pin drives the LED.
* In this approach, the port pin 'sources' current to the LED, when the port pin is at logic High (Logic '1').

→ In the second method, the cathode of the LED is connected to the port pin ~~drives~~ of the ~~cathode~~ processor/controller and the anode to the supply voltage through a current limiting resistor.
* The LED is turned ON when the port pin is at logic 'Low' (Logic '0'). Here the port pin 'sinks' current.

7-segment LED Display –

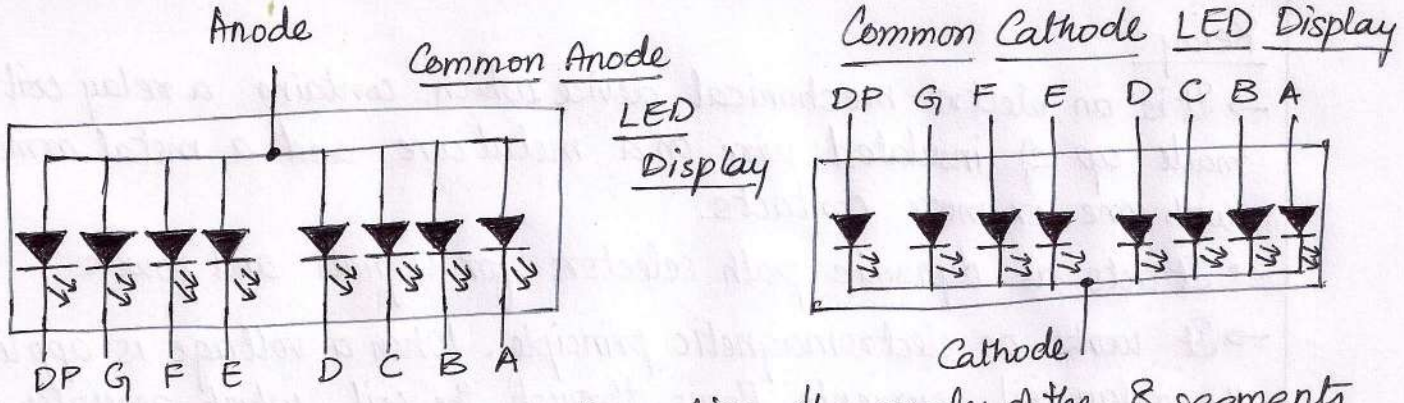
→ It is an output device for displaying ~~dpi~~ alpha numeric characters.
→ It contains 8 light-emitting diode (LED) segments arranged in a special form. 7 are used for displaying alpha numeric characters and 1 is used for ~~decimal~~ representing 'decimal point' in decimal number display.

7-Segment LED Display



→ It is a popular choice for low cost embedded applications like, Public telephone call monitoring devices, point of sale terminals, etc.

→ The LED segments are named A to G and the decimal point LED segment is named as DP. It should be lit accordingly to display numbers and characters.
→ For example – for displaying the number 4, the segments F, G, B and C are lit.
→ All these 8 LED segments need to be connected to one port of the processor/controller for displaying alpha numeric digits.
→ It is available in 2 different configurations – Common anode and Common Cathode.

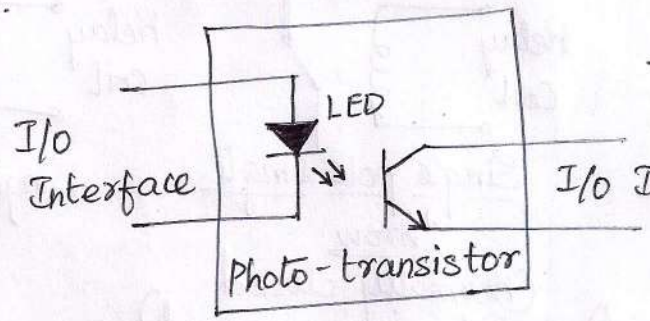


→ In the common anode configuration, the anodes of the 8 segments are ~~commonly~~ connected commonly where as in the common cathode configuration, the 8 LED segments share a common cathode line.

Optocoupler

→ It is a solid state device to isolate two parts of a circuit.

→ It combines an LED and a photo-transistor in a single housing (package).

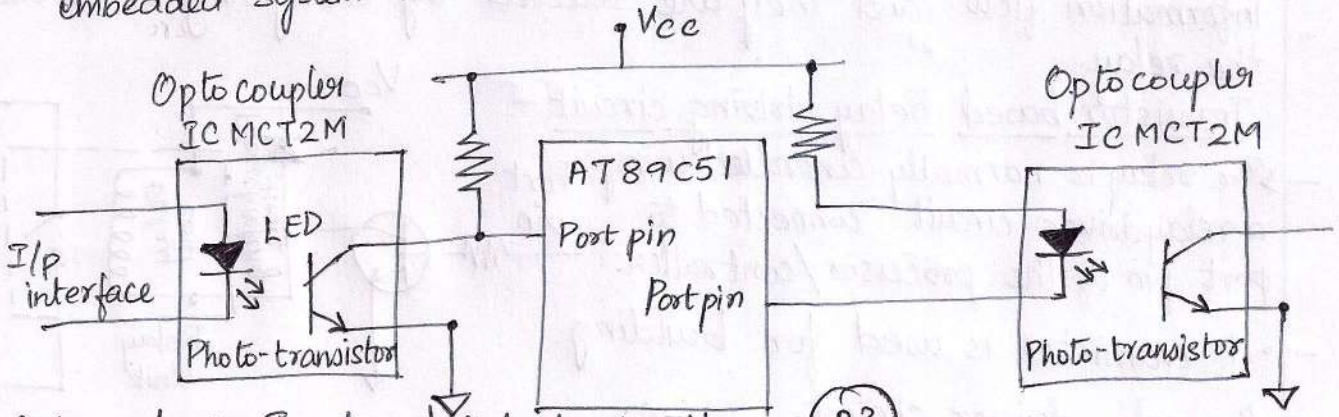


An optocoupler device.

→ It is used for suppressing interference in data communication, circuit isolation, high voltage separation, simultaneous separation and signal intensification, etc.

→ Optocouplers can be used in either input circuits or in output circuits.

→ It is used to isolate the input circuit and output circuit of an embedded system with a microcontroller as the system core.



Optocoupler in Input and Output circuit.

Relay

→ It is an electro-mechanical device which contains a relay coil made up of insulated wire on a metal core and a metal armature with one or more contacts.

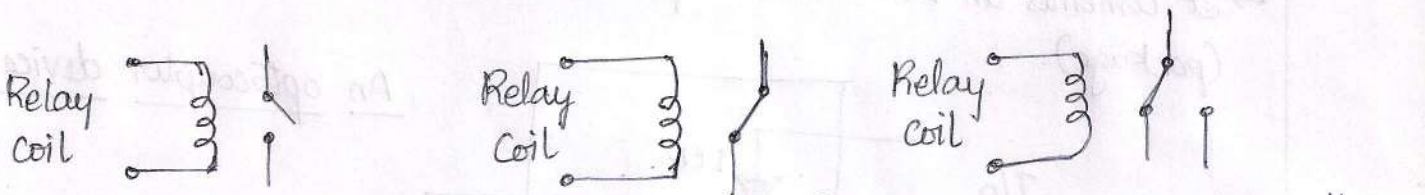
→ It acts as dynamic path selectors for signals and power.

→ It works on electromagnetic principle. When a voltage is applied to the relay coil, current flows through the coil, which generates a magnetic field.

→ The magnetic field attracts the armature core and moves the contact point.

→ The movement of the contact point changes the power/signal flow path.

Relay configurations in terms of Single Pole Single Throw.



Single Pole Single throw (normally open Relay is de-energised)

Single pole single throw (normally closed Relay is energised)

Single pole double throw

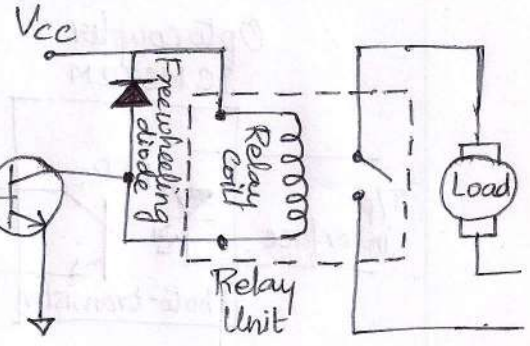
→ The Single Pole Single Throw configuration has only one path for information flow. The path is either open or closed in normal condition.

→ For Single Pole Double Throw relay, there are two paths for information flow and they are selected by energising or de-energising the relay.

Transistor based Relay driving circuit –

→ The relay is normally controlled using a relay driver circuit connected to port pin of the processor/controller.

→ A transistor is used for building the relay driver circuit.



→ A free-wheeling diode is used for free-wheeling the voltage produced in the opposite direction when the relay coil is de-energised.
→ The free-wheeling diode is essential for protecting the relay and the transistor.
→ Industrial relays are bulky and requires high voltage to operate. Reed relays are used in embedded applications requiring switching of low voltage DC signals.

PIEZO BUZZER

→ It is a piezoelectric ~~buzzer~~ device for generating audio indications in embedded application. It contains a piezoelectric diaphragm which produces audible sound in response to the voltage applied to it.
→ Self Driving and External Driving are two types of Piezo electric buzzers.
→ 'Self Driving' circuit contains all the necessary components to generate sound at a predefined tone.
It will generate a tone on applying the voltage.
→ 'External Driving piezo buzzers' supports the generation of different tones. The tone can be varied by applying a variable pulse train to the piezo electric buzzer.
→ A Piezo buzzer can be directly interfaced to the port pin of the processor / controller using a ~~transistor~~ based driver circuit (like in case of Relay) depending up on the driving current requirements.

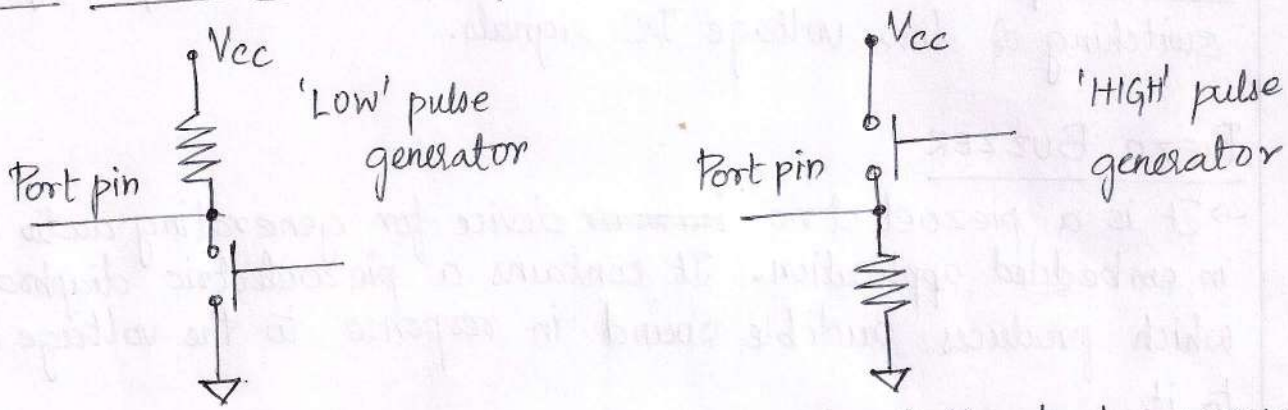
Push Button Switch –

→ It is an input device. Push button switch ~~com~~ has two configurations - 'Push to Make' and 'Push to Break'.
→ In 'Push to Make' configuration, the switch is normally ~~open~~ in the open state and it makes a circuit contact when it is pushed or pressed.
→ In 'Push to Break' configuration, the switch is normally in the closed state and it breaks the circuit contact when it is pushed or pressed.

→ The push button stays in the 'closed' or 'open' state as long as it is kept in the pushed state and it breaks / makes the circuit connection when it is released.

→ It is used as reset and start switch and pulse generator. It is also used for generating a momentary pulse.

Push button switch configurations.



→ It is normally connected to the port pin of the host processor/ controller.

→ Based on the interface b/w the push button switch and port pin of the controller, it can generate either a 'HIGH' pulse or a 'LOW' pulse.

COMMUNICATION INTERFACE

→ It is essential for communicating with various subsystems of the embedded system and with the external world.

→ Communication Interface can be classified in two types as

i) Device / board level communication interface and
(Onboard Communication Interface)

ii) Product level Communication Interface
(External Communication Interface)

→ Embedded product is a combination of different types of components (chips / devices) arranged on a printed circuit board (PCB).

→ The communication channel which interconnects the various inter connects within an embedded product referred to as On board Communication Interface like Serial interfaces like I^2C, SPI, UART, 1-Wire and Parallel bus interface.

→ External Communication Interface (Product level Communication Interface) is responsible for data transfer between the embedded system and other devices or modules.

→ It can be either a wired media or a wireless media and it can be a serial or a parallel interface.

→ Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio frequency waves (RF), GPRS, etc are examples for wireless communication interface.

→ RS232/RS-422/RS-485, USB, Ethernet IEEE 1394 port, Parallel Port, CF-II interface, SDIO, PCMCIA, etc, are examples for wired interfaces.

Onboard Communication Interfaces

→ It refers to the different communication channels/buses for interconnecting the various integrated circuits and other peripherals within the embedded system.

1) Inter Integrated Circuit (I2C) Bus

→ It is a synchronous bidirectional half duplex (one-directional communication at a given point of time) two wire serial interface bus.

→ It was developed by 'Philips semiconductors' in 1980s; to provide an easy way of connection between microprocessor/microcontroller system and the peripheral chips in television sets.

→ It comprise of two bus lines - Serial Clock - SCL and Serial Data - SDA.

→ SCL line is responsible for generating synchronisation clock pulses and SDA is responsible for transmitting the serial data across devices.

→ I2C bus is a shared bus system to which many number of I2C devices can be connected where it acts as either 'Master' device or 'Slave' device.

→ The 'Master' device is responsible for controlling the communication by initiating/terminating data transfer, sending data and generating necessary ~~sig~~ synchronisation clock pulses.

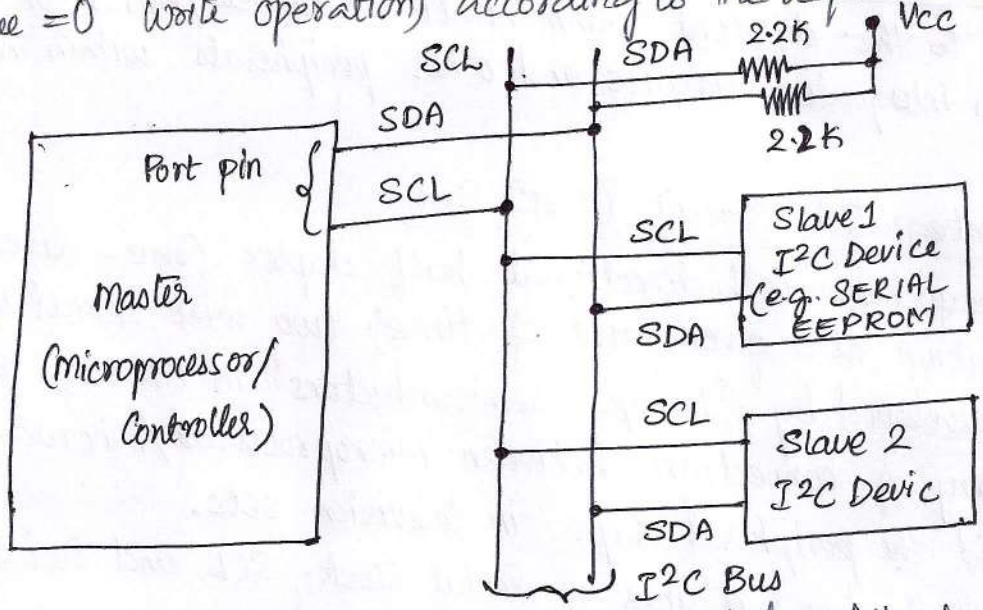
→ 'Slave' devices wait for the ~~channels~~ commands from the master and respond upon receiving the commands.

→ 'Master' and 'Slave' devices can acts as either Transmitter or Receiver.

→ It supports 3 different rates - Standard mode (Data rate up to 100Kbits/sec (100 Kbps)), Fast mode (Data rate up to 400Kbits/sec (400 Kbps)) and High speed mode (Data rate upto 3.4Mbits/sec (3.4 Mbps)).

→ The sequence of operations for communicating with an I^2C slave device-

1) The Master device pulls the clock line (SCL) of the bus to 'HIGH'.
2) The Master device pulls the data line (SDA) 'LOW', when the SCL line is at logic 'HIGH' (Start condition)
3) The master device sends the address (7bit or 10bit wide) of the 'slave' device to which it wants to communicate, over the SDA line.
4) The Master device sends the Read or Write bit (Bit value = 1 Read operation; Bit value = 0 Write operation) according to the requirement.



5) The master device waits for the acknowledge bit from the slave device whose address is sent on the bus along with the Read/write operation command.
6) The Slave device with the address requested by the master device responds by sending an acknowledge bit (Bit value = 1) over the SDA line.
7) After receiving the acknowledge bit, the master device sends the 8bit data to the slave device over SDA line, if the requested operation is 'Write to device'. (or) the slave device sends data to master over the SDA line where the requested operation is 'Read from Device'.
8) The master device waits for the acknowledgement bit from the device upon byte transfer complete for a write operation and sends a acknowledgement bit to the Slave device for a read operation. (9) The master device terminates the transfer by pulling the SDA line 'HIGH' when the clock line SCL is at logic 'HIGH'. (Stop condition).

2) Serial Peripheral Interface (SPI) Bus -

→ It is synchronous bidirectional full duplex four wire serial interface bus and was introduced by Motorola.

→ It is a single master-multi slave system, where device can be master provided the condition only one master device is active at any given point of time

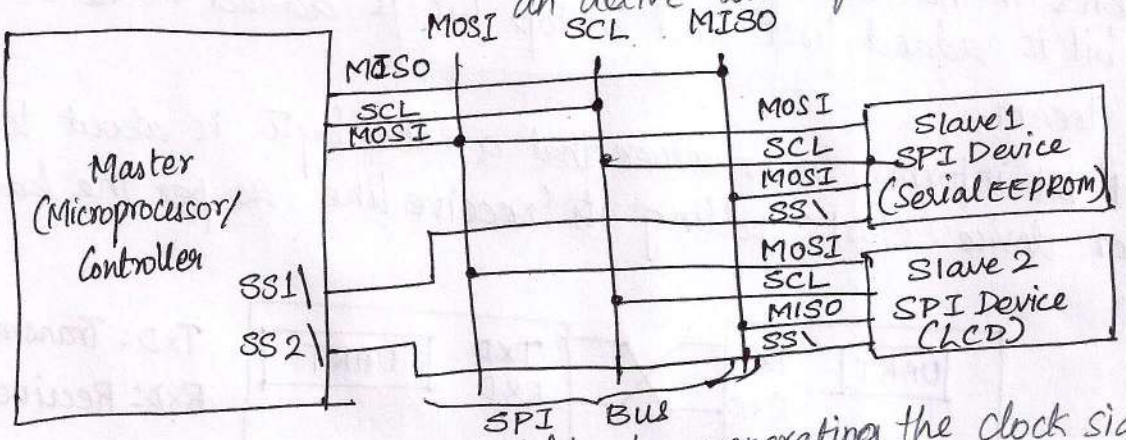
→ It requires four signal lines for communication. They are

* Master Out Slave In (MOSI): Signal line carrying the data from master to slave device. It is also known as "Slave Input / Slave Data In (SI/SDI)

* Master In Slave Out (MISO): Signal line carrying the data from slave to master device. It is also known as Slave Output / Slave Data Out (SO/SDO).

* Serial Clock (SCLK): Signal line carrying the clock signals.

* Slave Select (SS): Signal line for slave device select. It is an active low signal.



→ The master device is responsible for generating the clock signal. It selects the required slave device by asserting the corresponding slave device's slave select signal 'LOW'.

→ It works on the principle of "Shift register". The master and slave devices contain special shift register for the data to transmit or receive.

→ The size of the shift register is device dependent and it is a multiple of 8.

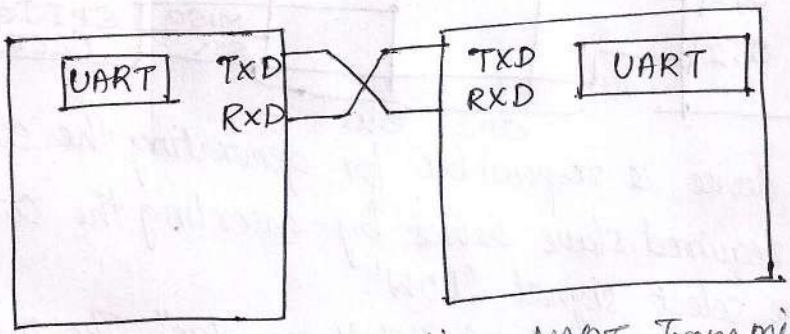
→ The serial transmission of data is fully configurable and contains a certain set of registers for holding these configurations.

→ The control registers of Serial ~~ph~~ peripheral holds configuration parameters like master/slave device selection, baudrate selection for communication, and clock signal control etc.

→ The status register holds the status of various conditions for transmission and reception.
→ During Transmission from the master to slave, the data in the master's shift register is shifted out to the MOSI pin and it enters the shift register of the slave device through the MOSI pin of the slave device.
→ At the same time the shifted out data bit from the slave device's shift register enters the shift register of the master device through MISO pin.

3) Universal Asynchronous Receiver Transmitter (UART)

→ It is an asynchronous form of serial data transmission. and does not require a clock signal to synchronize the transmitting end and receiving end for transmission.
→ The serial communication settings (Baudrate, number of bits per byte, parity, no. of start bits and stop bit and flow control) for both transmitter and receiver should be set as identical.
→ The start and stop of communication is indicated through inserting special bits in the data stream. While sending a byte of data, a start bit is added first and a stop bit is added at the end of the bit stream.
→ The 'start' bit informs the receiver that a data byte is about to arrive. The receiver device starts polling its 'receive line' as per the baudrate settings.



TxD: Transmitter line
RxD: Receiver line

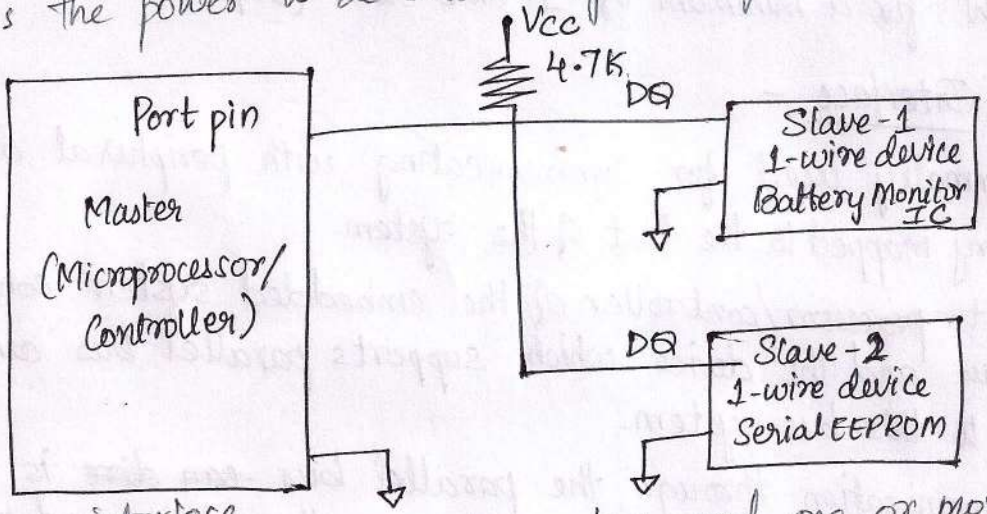
→ If parity is enabled for communication, UART Transmitting device adds a parity bit (In the transmitted bit stream, bit=1, odd no. of 1's; bit=0, even no. of 1's.
→ The UART of the receiving device calculates the parity of the bits received and compares it with the received parity bit for error checking.
→ The Receiver discards the 'Start', 'Stop' and 'Parity' bit from the received bit stream and the received serial bit data to a word.
→ It provides hardware handshaking signal support for controlling the serial data flow.

4) 1- Wire Interface -

→ It is an asynchronous half duplex communication protocol developed by Maxim Dallas Semiconductor and also known as Dallas 1-wire protocol.

→ 1 wire bus consists of a single signal line (wire) called DQ for communication and follows the master-slave communication model.

→ It allows the power to be sent along the signal wire as well.



interface

→ The 1-wire supports a single master and one or more slave devices on the bus.

→ Each device contains a globally ~~inputs~~ unique 64 bit identification number stored within it.

→ The identifier has three parts - an 8 bit family code, a 48 bit serial number and an 8 bit CRC computed from the first 56 bits.

→ The sequence of operation for communicating with a 1-wire slave device -

1) The master device sends a 'Reset' pulse on the 1-wire bus.
2) The slave device(s) present on the bus respond with a 'Presence' pulse.
3) The master device sends a ROM command (64 bit address of the device). It addresses the slave device(s) to which it wants to initiate a communication.
4) The master device sends a read/write function command to read/write the internal memory or register of the slave device.
5) The master initiates a Read data/write data from the device or to the device.

→ The communication over the 1-wire bus is initiated by master and it divides a timeslots of 60 μs. The 'Reset' pulse occupies 8 time slots.

→ For starting a communication, the master asserts the reset pulse by pulling the 1-wire bus 'LOW' for atleast 8 time slots (480μs).

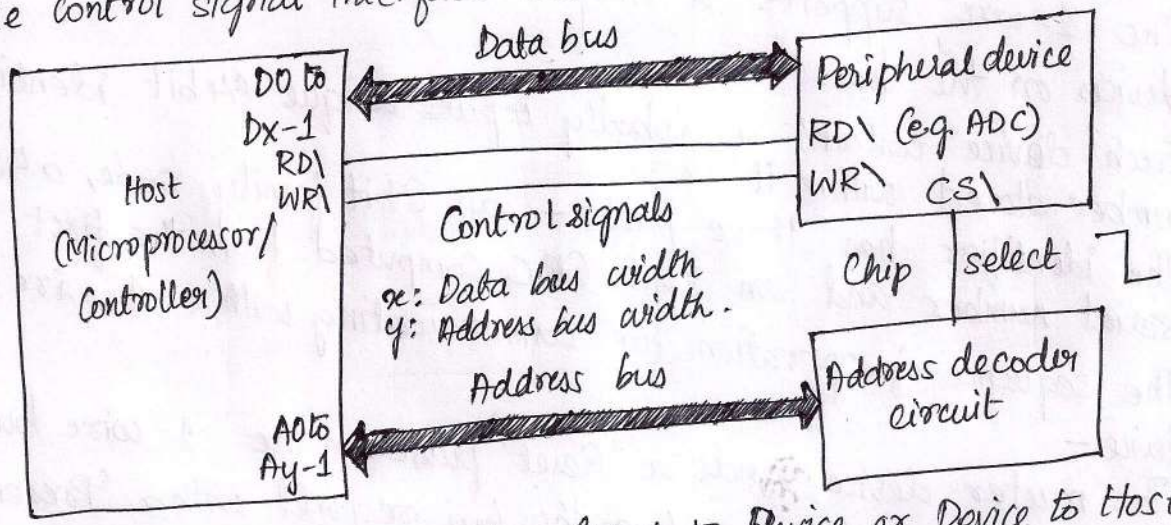
→ The slave devices responds with a 'Presence' pulse by pulling the 1-wire bus 'LOW' for a minimum of 1-time slot (60μs).

5) Parallel Interface -

→ It is normally used for communicating with peripheral devices which are memory mapped to the host of the system.

→ The host processor/controller of the embedded system contains a parallel bus and the device which supports parallel bus can directly connect to this bus system.

→ The communication through the parallel bus ~~can dire~~ is controlled by the control signal interface between the device and the host.



→ The direction of data transfer (Host to Device or Device to Host) can be controlled through the control signals lines for 'Read' and 'write'.

→ The Host processor has control over the 'Read' and 'write' control signals.

→ The device is memory mapped to the host processor and a range of address is assigned to it.

→ An Address decoder circuit is used for generating the chip select signal for the device.

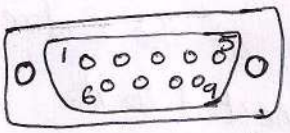
→ When the address selected by the processor is within the range assigned for the device, the decoder circuit activates the chip select line and there by the device becomes active.

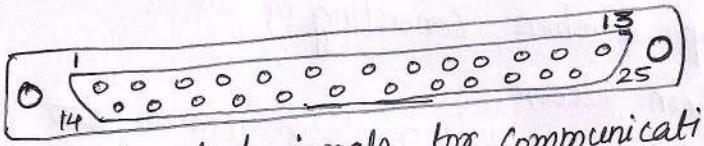
→ The processor then can read or write from or to the device by asserting the corresponding control line (RD\ and WR\ respectively).
→ The Interrupt line of the device is connected to the interrupt line of the processor and the corresponding interrupt is enabled in the host processor.
→ The width of the parallel interface is determined by the data bus width of the host processor. It can be 4bit, 8bit, 16bit, 32bit or 64 bit.
→ The parallel interface bus width supported by the device should be same as that of the host processor.

External Communication Interface

It refers to the different communication channels/buses used by the embedded system to communicate with the external world.

1) RS232C and RS485 - is developed by EIA in early 1960's.
→ RS232C (Recommended Standard number 232, revision C from the Electronic Industry Association) is a legacy, full duplex, wired, asynchronous serial communication interface.
→ It extends the UART communication signals for external data communication.
→ It follows the EIA standard for bit transmission where logic '0' is represented with voltage between +3 and +25V and logic '1' is represented with voltage between -3 and -25V.
→ In EIA standard, logic '0' is known as 'Space' and logic '1' as 'mark'.
→ It support two different types of connectors namely - DB-9: 9-Pin Connector and DB-25: 25 pin connector.

DB-9 



DB-25

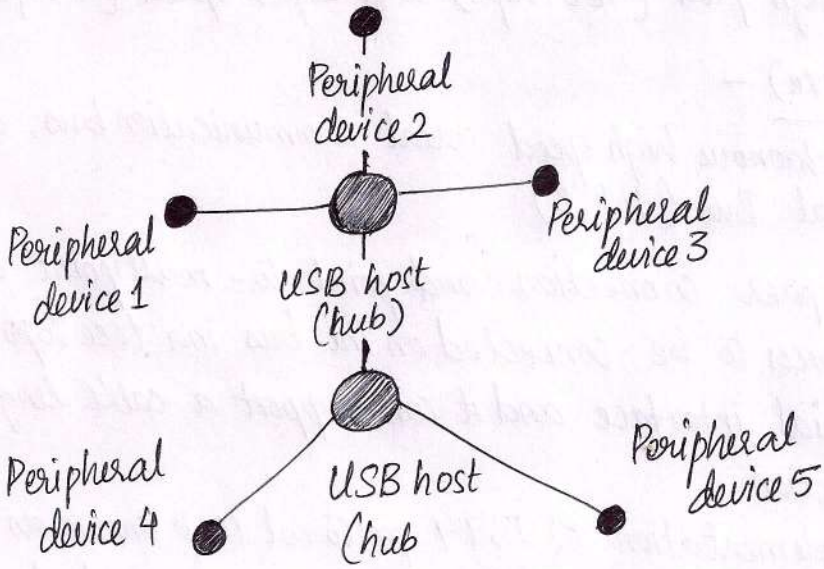
→ It defines various handshaking and control signals for communication apart from the 'Transmit' and 'Receive' signals lines for data communication.
→ RS232 - DB-9 - 9pin connector pin details
1 - DCD (Data Carrier Detect), 2 - RXD - Receive Pin, 3 - TXD - Transmit Pin, 4 - DTR - Data Terminal Ready, 5 - GND - Signal Ground, 6 - DSR - Data Set Ready, 7 - RTS - Request to Send, 8 - CTS - Clear to Send, 9 - RI - Ring Indicator.

→ It is a point-to-point communication interface and the devices involved in RS-232 communication are called 'Data Terminal Equipment' (DTE) and 'Data Communication Equipment' (DCE).
→ The RXD pin of DCE should be connected to TXD pin of DTE and vice versa for proper data transmission.
→ The control signals are implemented mainly for modem communication. (and some of them may not be relevant for other types of devices.)
→ The Request To Send (RTS) and Clear To Send (CTS) signals co-ordinate the communication between DTE and DCE.
→ Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line.
→ The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data.
→ The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link.
DTR should be in the activated state before the activation of DSR.
→ The Data Carrier Detect (DCD) control signal is used by the DCE to indicate the DTE that a good signal is being received.
→ Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line.

2) Universal Serial Bus (USB)
→ It is a wired high speed serial bus for data communication.
→ USB 1.0 was released in 1995 and was created by the USB core group members consisting of Intel, Microsoft, IBM, Compaq, Digital and Northern Telecom.
→ It follows a star topology with a ~~star~~ USB host at the centre and one or more USB peripheral devices / USB hosts connected to it.
→ A USB host can support connections upto 127, including slave peripheral devices and other USB hosts.
→ The Pin details for the connectors are as shown below -

Pin No.	Pin Name	Description
1	V_{BUS}	Carries power (5V)
2	D-	Differential data carrier line
3	D+	Differential data carrier line
4	GND	Ground signal line.

The star topology for USB device connection.



→ The physical connection between a USB peripheral device and master device is established with a USB cable.

→ The USB cable supports communication distance of up to 5 metres.

→ The USB standard uses two different types of connector at the ends of the USB cable - 'Type A' connector is used for upstream connection (connection with host), and 'Type B' connector is used for downstream connection (connection with slave device)

→ It transmits data in packet format. Each data packet has a standard format.

→ The USB host contains a host controller which is responsible for controlling the data communication, including establishing connectivity with USB slave devices, packetizing and formatting the data

→ It supports four different types of data transfers namely - Control, Bulk, Isochronous and Interrupt.

Control Transfer is used by USB system software to query, configure and issue commands to the USB device.

Bulk Transfer is used for sending a block of data to a device. It supports error checking and correction. example - Printer, for bulk transfer.

Isochronous data transfer is used for real-time data communication where data is transmitted as streams in real-time and doesn't support error checking and retransmission of data in case of any transmission loss.

Interrupt transfer is used for transferring small amount of data and makes use of polling technique to see whether the USB device has any data to send. examples - Devices like Mouse & Keyboard.

→ USB supports four different data rates namely - Low speed (1.5Mbps), Full speed (12Mbps), High Speed (480Mbps) and Super speed (4.8Gbps).

3) IEEE 1394 (Firewire) -

→ It is a wired, isochronous high speed serial communication bus, called as High Performance Serial Bus (HPSB).

→ It supports peer-to-peer connection and point-to-multipoint communication allowing 63 devices to be connected on the bus in a tree topology.

→ 1394 is a wired serial interface and it can support a cable length of upto 15 feet for interconnection.

→ Apple Inc's ~~imp~~ implementation of 1394 protocol is known as Firewire. i.LINK is the 1394 implementation from Sony corporation and Lynx is the implementation from Texas instruments.

→ ~~It can support a cable length of upto~~ It supports a data rate of 400 to 3200 Mbits/sec.

→ It uses differential data transfer and the interface cable supports 3 types of connectors - 4 pin connector, 6-pin connector (alpha connector), 9 Pin Connector (beta connector).

→ There are two differential data transfer lines A and B per connector. In a 1394 cable, normally the differential lines of A are connected to B (TPA+ to TPB+ and TPA- to TPB-) and vice versa. (in case 4-pin connector).

→ 1394 is a popular communication interface for connecting embedded devices like Digital Camera, Camcorder, Scanners to desktop computer for data transfer and storage.

→ It can directly connect a scanner with a printer and doesn't require a host for communicating between the devices.

→ The data rate supported by 1394 is far higher than the one supported by USB 2.0 interface.

→ The 1394 hardware implementation is much costlier than USB implementation.

4) Infrared (IrDA)

→ It is a serial, half duplex, cline of sight based wireless technology for data communication between devices.
→ The remote control of TV, VCD player, DVD player, etc. works on Infrared data communication principle.
→ It uses infrared waves of the electromagnetic spectrum for transmitting the data.
→ It supports point-point and point-to-multipoint communication, provided all devices involved in the communication are within line of sight.
→ It supports a typical communication range of 10cm to 1m. and data rates ranging from 9600 bits/second to 16 Mbps.
→ The speed of data transmission IR can be classified into -
* Serial IR (SIR) - supports transmission rates ranging from 9600bps to 115.2 Kbps.
* Medium IR (MIR) - supports data rates of 0.576 Mbps and 1.152 Mbps.
* Fast IR (FIR) - Supports data rates of upto 4Mbps.
* Very Fast IR (VFIR) - supports high data rates upto 16Mbps.
* Ultra Fast IR (UFIR) - supports very high data rates upto 100Mbps.
→ It involves a transmitter unit for transmitting the data over IR and a receiver for receiving the data
→ Infra LED is the IR source for transmitter and at the receiving end a photodiode acts as the receiver. Each device supporting IrDA communication for bidirectional data transfer are known as 'Transceiver'.
→ 'Infra-red Data Association (IrDA) is the regulatory body responsible for defining and licensing the specifications for IR data communication.
→ It has two essential parts; a physical link and a protocol part. The physical link is responsible for the physical transmission of data between devices supporting IR communication and protocol part is responsible for defining the rules of communication.
→ IrDA control protocol contains implementations for Physical Layer (PHY), Media Access Control (MAC), and Logical Link Control (LLC). The physical layer defines the physical characteristics of communication like range, data rates, power etc.

→ IrDA is a popular interface for file exchange and data transfer in low cost devices. It was the prominent communication channel in mobile phones before Blue tooth's existence.

5) Blue tooth (BT)

→ Blue tooth is a low cost, low power, short range wireless technology for data and voice communication. It was first proposed by 'Ericsson' in 1994.

→ It operates at 2.4GHz of the Radio frequency spectrum and uses the Frequency Hopping Spread Spectrum (FHSS) technique for communication.

→ It supports a data rate of upto 1Mbps and a range of approximately 30 feet for data communication.

→ It also has two essential parts; a physical link part and a protocol part. The physical link is responsible for the physical transmission of data between devices supporting Blue tooth communication (using RF waves) and protocol part is responsible for defining the rules of communication. implemented as 'Blue tooth protocol stack'.

→ It supports point-to-point (device-to-device) or (master-slave) and point-to-multipoint (device to multiple device broadcasting) wireless communication.

→ It is the favourite choice for short range data communication in handheld embedded devices. It is the easiest communication channel for transferring ringtones, music files, pictures, media files, etc between neighbouring Bluetooth enabled phones.

→ The Bluetooth standard specifies the minimum requirements - Generic Access Profile (GAP) defines the requirements for detecting a blue tooth device and establishing a connection with it.

* Serial Port Profile for serial data communication, File Transfer Profile (FTP) for file transfer between devices, Human Interface Device (HID) for supporting human interface devices like Keyboard and Mouse etc.

→ The specifications for Bluetooth communication is defined and licensed by 'Blue tooth Special Interest Group (SIG)'.

6) Wi-Fi or Wireless Fidelity –

→ It is the popular wireless communication technique for networked communication of devices. It follows the IEEE 802.11 standard.

→ It is used for network communication and it supports Internet Protocol (IP) based communication.

→ It is important to device identities in a multipoint communication to address specific devices for data communication.

→ It is an IP based communication each device is identified by an IP address, which is unique to each device on the network.

→ It require an intermediate agent called Wi-Fi router / Wireless access point.

→ Wireless router is responsible for restricting the access to a network, assigning IP address to devices on the network, routing the data packets to the intended devices on the network.

→ WiFi enabled devices contain a wireless adaptor / WiFi Radio for transmitting and receiving data in the form of radio signals through an antenna.

→ It operates at 2.4GHz or 5GHz of radio spectrum and they co-exist with other ISM band devices like Bluetooth.

→ It supports data rates ranging from 1Mbps to 150Mbps depending on the standards (802.11 a/b/g/n) and access/modulation method.

→ WiFi offers a range of 100 to 300 feet, depending on the type of antenna and usage location (indoor/outdoor).

→ In a Wi-Fi network, when its WiFi radio is turned ON, searches the available Wi-Fi network in its vicinity and lists out the Service Set Identifier (SSID) of the available networks.

→ A password is required to connect to a particular SSID where if the network is security enabled.

→ Wi-Fi employs different security mechanisms like Wired Equivalency Privacy (WEP), Wireless Protected Access (WPA) etc. for securing the data communication.

7) ZigBee -

→ It is a low power, low cost, wireless network communication protocol based on the IEEE 802.15.4-2006 standard.

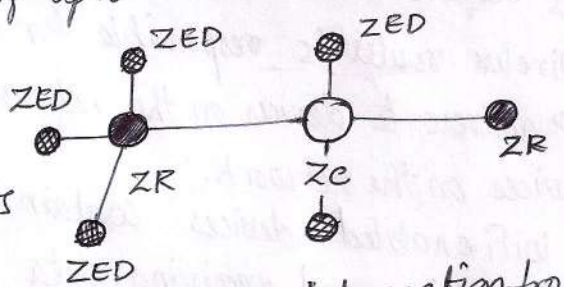
→ It is targeted for low power, low data rate and secure applications for Wireless Personal Area Network (WPAN).

→ It support a robust mesh network containing multiples nodes. (Its networking strategy makes the network ~~mobile~~ reliable by permitting messages to travel through a no. of different paths to get from one node to another)

→ It operates at the unlicensed bands of Radio spectrum mainly at 2.400 to 2.484 GHz, 902 to 928 MHz and 868.0 to 868.6 MHz.

→ It supports an operating distance of upto 100 metres and a datarate of 20 to 250 kbps.

A Zigbee network model



* Zigbee Coordinator (ZC) - It acts as the root of the Zigbee network.

* Zigbee Router (ZR) - It is responsible for passing information from device to another device or to another ZR. & also called as full function device (FFD).

* Zigbee End device (ZED) or Reduced Function Device (RFD) - End device containing Zigbee functionality for data communication. It can talk only with a ZR or ZC and doesn't have the capability to act as a mediator for transferring data from one device to another.

→ Zigbee is primarily targeting applications areas like home & industrial automation, energy management, home control/security, medical/patient tracking, logistics & asset tracking and sensor networks & active RFID.

→ Automatic Meter Reading (AMR), Smoke detectors, wireless telemetry, HVAC control, heating control, lighting controls, environmental controls etc. are examples for applications which can make use of the Zigbee technology.

8) General Packet Radio Services (GPRS) –

→ It is a communication technique for transferring data over a mobile communi-cation network like GSM.
→ Data is sent as packets in GPRS communication. The transmitting device splits the data into several related packets. At the receiving end, the data is reconstructed by combining the received data packets.
→ It supports a theoretical maximum transfer rate of 171.2 kbps.
→ In GPRS communication, the radio channel is concurrently shared between several users instead of dedicating a radio channel to a cellphone user.
→ It divides the channel into 8 timeslots and transmits data over the available channel.
→ It supports Internet Protocol (IP), Point to Point Protocol (PPP) and X.25 protocols for communications.
→ It is mainly used ~~for~~ by mobile enabled embedded devices for data communi-cation and supports the necessary GPRS hardware like GPRS modem and GPRS radio.
→ It is an old technology and is replaced by new generation data configuration techniques like EDGE, High Speed Downlink Packet Access (HSDPA), etc which offers higher bandwidths for communication..

Embedded Firmware

→ It refers to the control algorithm (program instructions) and or the configuration settings that an embedded system developer dumps into the code (program) memory of the embedded system.
→ It is an unavoidable part of the embedded system.
→ There are various methods available for developing the embedded firmware.
1. Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment. (The IDEs contains an editor, compiler, linker, debugger, simulator, etc).
2. Write the program in Assembly language using the instructions supported by application's target processor / controller.

→ The instruction set for each family of processor/controller is different and the program written in assembly language or high level languages like embedded C/C++.

→ It should be converted into a processor understandable machine code before loading it into the program memory.

→ The process of converting the program written in either a high level language or processor/controller specific Assembly code to machine readable binary code is called 'HEX File Creation'.

→ The methods used for 'HEX file creation' is different depending on the programming techniques used.

→ If the program is written in Embedded C/C++ using an IDE, the cross compiler included in the IDE converts it into corresponding processor/controller understandable 'HEX file'.

→ The embedded software development process in assembly language is tedious and time consuming.

→ Programs written in high level languages are not developer dependent. Any skilled programmer can trace out the functionalities of the program easily as it contains necessary comments and documentation lines.

→ It is very easy to debug and the overall system development time will be reduced to a greater extent.

Other System Components

→ It refers to the components/circuits/ICs which are necessary for the proper functioning of the embedded system as it is essential for the working of the processor/controller and firmware execution.

→ Watchdog timer, Reset IC (or Passive circuit), Brown-out protection IC (or passive circuit), etc are examples of circuits/ICs which are essential for the proper functioning of the processor/controller.

1) Reset Circuit

→ It is essential to ensure that the device is not operating at a voltage level where the device is not guaranteed to operate, during system power ON.

→ The reset signal brings the internal registers and the different hardware systems of the processor/controller to a known state and starts the firmware execution from the reset vector.

→ The reset vector can be relocated to an address for processors/controllers supporting boot loader.

→ The reset signal can be either active high or active low.

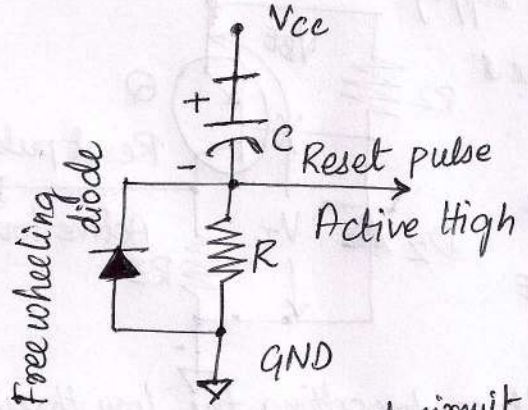
→ Since the processor operation is synchronized to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilise before the internal reset state starts.

→ The reset signal to the processor can be applied at Power ON through an external passive reset circuit comprising a Capacitor and Resistor or through a standard Reset IC. (like MAX810 from Maxim Dallas).

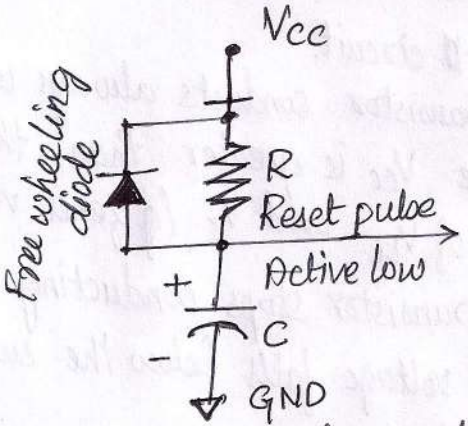
→ Microprocessors / controllers contain built-in reset circuitry and they don't require external reset circuitry

RC based reset circuit. - It is a resistor capacitor based passive reset circuit for active high and low configurations.

The reset pulse width can be adjusted by changing the resistance value R and Capacitance value C.



RC-based passive reset circuit for active high configurations



RC-based passive reset circuit for active low configurations

2) Brown-out Protection Circuit

→ It prevents the processor/controller from unexpected program execution behaviour when the supply voltage to the processor/~~controller~~ controller falls below a specified voltage.

→ It is essential for battery powered devices since there are greater ~~devices~~ chances for the battery voltage to droop below the required threshold.

→ The processor behaviour may not be ~~like~~ predictable if the supply voltage falls below the recommended operating voltage. It may lead to situations like data corruption.

→ It holds the processor/controller in reset state, when the operating voltage falls below the ~~the~~ threshold, until it rises above the threshold voltage

→ Few processors/controllers support built-in brown-out protection circuit which monitors the supply voltage internally and others who doesn't integrate a built-in ~~brown~~ ~~is~~ brown-out protection circuit can be implemented using external passive circuits (or) supervisor ICs.

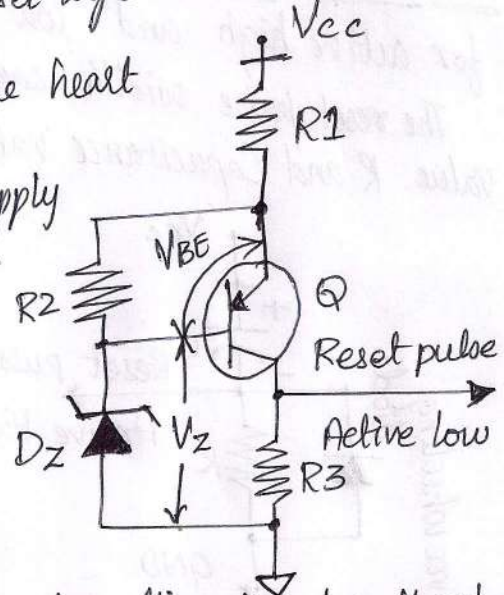
Brown-out protection circuit with Active low output

A Brown-out protection circuit ~~imp~~ consists of Zener diode and Transistor for processor/controller with active low Reset logic.

→ The Zener diode D_Z and Transistor Q forms the heart of this ~~the~~ circuit.

→ The transistor conducts always when the supply voltage V_{cc} is greater than that of the sum of V_{BE} and V_Z (~~of~~ zener voltage).

→ The transistor stops conducting when the supply voltage falls below the sum of V_{BE} and V_Z.



→ Select the Zener diode with required voltage for setting the low threshold value for V_{cc}.

→ The values of R1, R2, and R3 can be selected based on the electrical characteristics (absolute maximum current and voltage ratings) of the transistor in use.

3) Oscillator Unit -

→ A Microprocessor / microcontroller is a digital device made up of digital combinational and sequential circuits.

→ The instruction execution of a microcontroller / microprocessor occurs in sync with a clock signal.

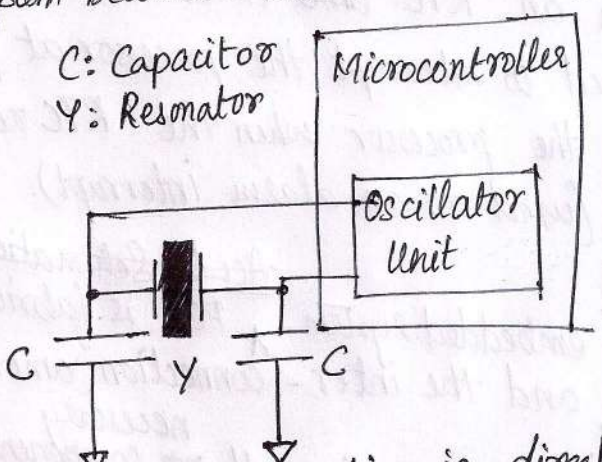
→ It is analogous to the heartbeat of a living being which synchronises the execution of life. For a living being, the heart is responsible for the generation of the beat where as the oscillator unit of the embedded system is responsible for generating the precise clock for the processor.

→ Certain processors/controllers integrate a built-in oscillator unit and simply require an external ceramic resonator/quartz crystal for producing the necessary clock signals.

→ Certain devices may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally.

→ Quartz crystal Oscillators are available in the form of chips and they can be used for generating the clock pulses ~~in such a cases~~. The speed of operation of a processor is primarily dependent on the clock frequency.

→ The logical circuits lying inside the processor always have an upper threshold value for the maximum clock at ~~the~~ which the system can run beyond which the system becomes unstable and non functional.



Oscillator circuitry using quartz crystal and quartz crystal oscillator

→ The total system power consumption is directly ~~proper~~ proportional to the clock frequency. The power consumption increases with increase in clock frequency.

→ The accuracy of program execution depends on the accuracy of the clock signal. The accuracy of the crystal oscillator or ceramic resonator is normally expressed in terms of +/- ppm (Parts per million).

4) Real-Time Clock (RTC)

→ It is a system component responsible for keeping track of time. It holds information like current time (in hours, minutes and seconds) in 12 hour/24 hour format, date, month, year, day of the week, etc.
→ It is intended to function even in the absence of power.
→ It is available in the form of ICs from different semiconductor manufacturers like Maxim/Dallas, ST Microelectronics etc.
→ RTC chip contains a microchip for holding the time and date related information and back up battery cell for functioning in the absence of power, in a single IC package.
→ RTC chip is interfaced to the processor or controller of the embedded system.
→ For Operating system based embedded devices, a timing reference is essential for synchronising the operations of the OS kernel.
→ It can interrupt the OS kernel by asserting the interrupt line of the processor/controller to which the RTC interrupt line is connected.
→ The OS kernel identifies the interrupt in terms of the Interrupt Request (IRQ) number generated by an interrupt controller.
→ One IRQ can be assigned to the RTC interrupt and the kernel can perform necessary operations like system date time updation, managing software timers etc. when an RTC ~~time~~ timer tick interrupt occurs.
→ The RTC can be configured to interrupt the processor at predefined intervals (or) to interrupt the processor when the RTC register reaches a specified value (used as an alarm interrupt).

5) PCB and Passive Components

→ It is the backbone of every embedded system. A PCB is fabricated only after finalising the components and the inter-connection among them as per schematic design is created.
→ It acts as a ~~necessary~~ platform for mounting all necessary components and for testing embedded firmware as per design requirement.
→ Resistor, Capacitor, Diodes etc are few passive components on board along subsystems in embedded systems.
→ Additional chips are like co-workers of embedded hardware for its proper functioning e.g. a regulator IC and Spike suppressor filter capacitors.

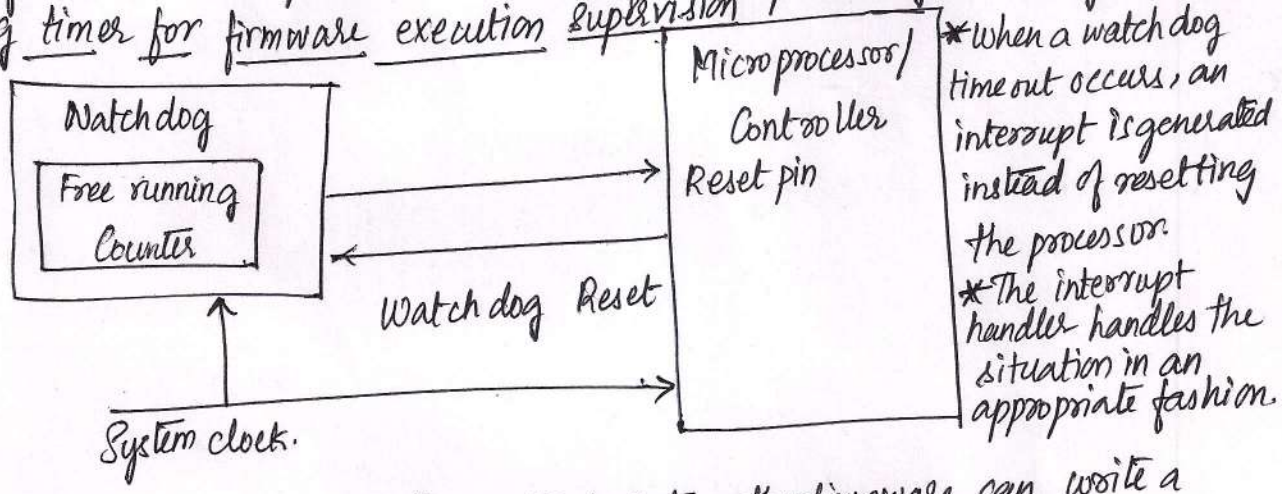
6) Watch Dog Timer –

→ A Watch Dog timer or simply a watch dog is a hardware timer for monitoring the firmware execution.

→ A Watch Dog to monitor the firmware execution and reset the system processor / microcontroller when the program execution hangs up.

→ It is implemented internally where it increments or decrements a free running counter with each clock pulse and generates a reset signal to reset the processor — if the count reaches zero for a down counting watch dog, or the highest count value for an up counting watch dog.

→ Watchdog timer for firmware execution supervision



→ If the watch dog counter is in the enabled state, the firmware can write a zero (up counting watchdog) to it before starting the execution of a piece of code (subroutine) which is susceptible to execution hang up) and the watch dog will start counting.

→ If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate a reset pulse and this will reset the processor.

→ If the firmware execution completes before the expiration of the watch dog timer which can reset the count by writing a 0 (up counter) to the watchdog timer register.

→ Watchdog is implemented as built-in components in many processors and provides status register to control the watch dog timer (like enabling and disabling) and Watchdog timer register for writing the count value.

→ The external Watch dog timer IC can be implemented using hardware logic for enabling / disabling, resetting the watch dog count, etc instead of the firmware based 'writing' to the status and watch dog timer register.

10.05.2018

Module-4 Embedded System Design Concepts

B.S.Balaji,
Asst. Prof,
B.G.S.I.T.

Characteristics of an Embedded System

The important characteristics of an embedded system are -

1) Application and Domain specific

→ Each embedded system is having certain functions to perform and they are developed to do the intended functions only.

→ It is the major criterion which distinguishes an embedded system from a general purpose system. They cannot be used for any other purpose.

Example- The embedded control unit of microwave oven cannot be replaced by air conditioner's embedded control unit, because they are specifically designed to perform certain specific tasks.

2) Reactive and Real Time

Embedded systems are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system. (Event is the changes happening in the real world.)

→ Embedded systems produce changes in output in response to the changes in the input. They are generally referred as Reactive Systems.

→ Real Time System operation means the timing behaviour of the system should be deterministic; the system should respond to requests or tasks in a known amount of time.

It should not miss any deadlines for tasks or operations.

→ Embedded applications or systems are mission critical like flight control systems, Antilock Brake Systems (ABS), etc. are examples of Real Time systems.

3) Operates in Harsh environment

→ All embedded systems deployed are not always in controlled environments. but it may be a dusty one or a high temperature zone or an area subject to vibrations and shock.

→ Systems placed in such areas should be capable to withstand all these adverse operating conditions.

supply
→ Power fluctuations, corrosion, and component aging etc are the other factors that need to be taken into consideration for embedded systems to work in harsh environments.

4) Distributed

→ The term distributed means that embedded systems may be a larger systems. Many no. of such distributed embedded systems form a single large embedded control unit.

→ Example - Automatic Teller Machine (ATM) contains a card reader embedded unit, responsible for reading and validating the user's ATM card, transaction unit for performing transactions, a currency counter for dispatching / vending currency to the authorised person and a printer unit for printing the transaction details.

5) Small Size and Weight

→ Product aesthetics is an important factor in choosing a product. like size, weight, shape, style etc. ~~Many application demands small sized~~

→ For example, People plan to buy a new mobile phone based on the comparative study of pros and cons of the products available in the market. People believe in the phrase "Small is beautiful".

→ Many application demands compact, small sized and low weight products.

6) Power concerns

→ Power management is another important factor that needs to be considered in designing embedded systems in order to minimise the heat dissipation by the system.

→ The production of high amount of heat demands cooling requirements like cooling fans which occupies additional space and make the system bulky.

→ It is critical constraint in battery operated applications where more the power consumption the less is the battery life.

→ Select the design using low power components like low dropout regulators and controllers / processors with power saving modes.

Quality attributes of Embedded Systems

Quality attributes are the non-functional requirements that need to be documented properly in any system design.

Quality attributes can be broadly classified in to two, namely -

1) Operational Quality attributes, and
2) Non-Operational Quality attributes.

1) Operational Quality attributes.

It represents the relevant quality attributes related to the embedded system design when it is in the operational mode or 'online' mode.

The important Operational quality attributes are -

(i) Response
(ii) Throughput
(iii) Reliability
(iv) Maintainability
(v) Security
(vi) Safety.

(i) Response - It is a measure of quickness of the system. where it informs about how fast your system is tracking the changes in input variables.

Example - In Real time, An embedded system deployed in flight control application where any response delay in the system will create potential damages to the safety of the flight as well as the passengers.

(ii) Throughput - It deals with the efficiency of a system. It can be defined as the rate of production or operation of a defined process over a stated period of time.

Throughput is measured in terms of 'Benchmark' where it is a performance criteria that a product is expected to meet or a standard product that (Benchmark) can be used for comparing other products of the same product line.

(iii) Reliability - It is a measure of how much % you can rely upon the proper functioning of the system (or) what is the % susceptibility of the system to failures. Reliability is defined as (i) Mean Time Between Failures (MTBF) - gives the frequency of failures in hours/ weeks/ months. and (ii) Mean Time To Repair (MTTR) specifies how long the system is allowed to be out of order following a failure.

(iv) Maintainability - It deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup.

It can be classified into two categories namely - 'Scheduled or Periodic Maintenance' (preventive maintenance) and 'Maintenance to unexpected failures' (corrective maintenance).

(1) Scheduled or Periodic maintenance - example - an inkjet printer uses ink cartridges, which are consumable components and as per the printer manufacturer the end user should replace the cartridge after each 'n' no. of printouts to get quality prints.

(2) Maintenance to unexpected failures - example - If the paper feeding part of the printer fails then the printer fails to print and it requires immediate repairs to rectify this problem.

(v) Security - Confidentiality, Integrity and Availability are three major measures of information security.

→ Confidentiality deals with the protection of data and application from ~~that~~ unauthorised disclosure.

→ Integrity deals with the protection of data and application from unauthorized modification.

→ Availability deals with the protection of data and application from unauthorized users. Example - Personal Digital Assistant (PDA)

(vi) Safety - It deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system ~~or due~~ to the emission of radioactive or hazardous materials from the embedded products.

(The breakdown of an embedded system may occur due to a hardware failure or a firmware failure).

Safety analysis is used to evaluate the anticipated damages & determine the best course of action to bring down the consequences of the damages to an acceptable level.

2) Non operational Quality attributes -

→ The quality attributes that needs to be addressed for the product 'not' on the basis of operational aspects.

The important non operational quality attributes are -

1. Testability and Debugability
2. Evolvability
3. Portability
4. Time to prototype and market
5. Per unit and Total cost.

1. Testability and Debugability

→ Testability deals with how easily one can test his/her design, application and it is applicable to the embedded Hardware and firmware.

→ Embedded hardware testing ensures that the peripherals and the total hardware functions in the desired manner.

→ Firmware testing ensures that the firmware is functioning in the expected way.

→ Debugability is a means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system. It has two aspects - (i) Hardware debugging which is used for figuring out the issues created by hardware problems where as (ii) firmware debugging is employed to figure out the probable errors that appear as a result of flaws in the firmware.

2. Evolvability - It is referred as non-heritable variation where it eases the design of embedded product can be modified to take advantage of new firmware or hardware technologies.

3. Portability - It is a measure of 'system independence' where an embedded product is said to be portable if the product is capable of functioning 'as such' in various environments, target processors/controllers and embedded operating systems.

4. Time-to-Prototype and Market -

→ Time to market is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial product).

→ It is a critical factor in the success of a commercial embedded product.

→ Time-to-Prototype is also another critical factor where the rapid product development in which the important features of the product under consideration are developed and it helps a lot in reducing time-to-market.

5. Per Unit Cost and Revenue (or) Per Unit and Total cost.

→ Cost is a factor which is closely ~~monit~~ monitored by both end user and product manufacturer. It is a highly sensitive factor for commercial products. where any failure to position the cost of a ~~comme~~ commercial product at a nominal rate, may lead to the failure of the product in the market.

→ The ultimate aim of a product is to generate marginal profit to designer / product development company. where the budget and total system ~~cost~~ cost should be properly balanced to provide a marginal profit.

Product life cycle (PLC) curve

→ Every embedded product has a product life cycle which starts with the design and development phase. (like product idea generation, prototyping, roadmap definition, actual product design and development).

* In this phase, there is only investment and no returns.

→ Product Introduction stage - If the product is ready to sell, it is introduced to the market where the sales and revenue will low and less competition but the product sales and revenue increases with time.

→ Growth phase - the product grabs high market share.

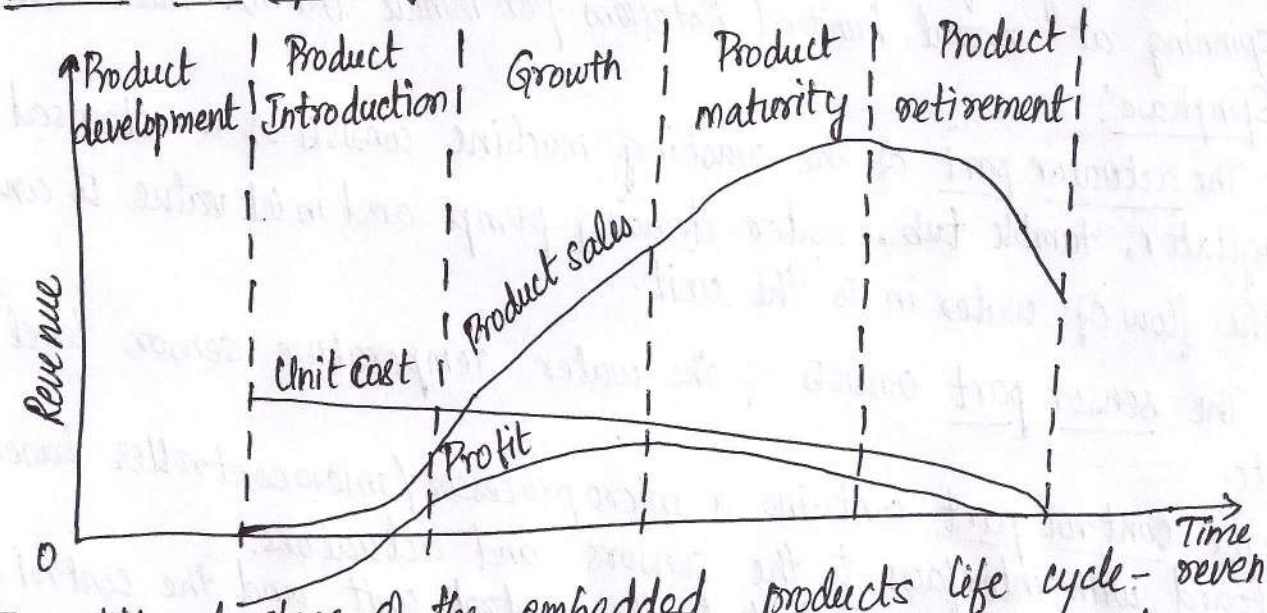
→ Maturity phase - the growth and sales will be steady and the revenue reaches at its peak.

→ Product Retirement / Decline phase - It starts with the drop in sales volume, market share and revenue.

* The decline happens due to various reasons like competitions from similar product with enhanced features or technology changes. etc.

→ In Decline phase, the manufacturer declares discontinuing of the product.

Product Life-cycle graph



* The different stage of the embedded products life cycle - revenue, unit-cost and profit in each stage are represented in the graph.

* It is clear that the total revenue increases from the product introduction stage to the product maturity stage.

Embedded Systems - Application - and Domain-Specific.

1) Washing Machine - Application-Specific Embedded system.

→ It is a typical example of an embedded system providing extensive support in home automation applications.

→ An embedded system contains sensors, actuators, control unit and application-specific user interfaces like keyboards, display units etc.

→ Washing machine comes in two models, namely top loading and front loading machines.

→ In top loading models the agitator of the ~~model~~ machine twists back and forth and pulls the cloth down to the bottom of the tub. On reaching the bottom of the tub the clothes work their way back to the top of the tub where the agitator grabs them again and repeats the mechanism.

→ In the front loading machines, the clothes are tumbled and plunged into water over and over again. This is the first phase of washing.

* In the second phase of washing, the water is out from the tub and inner tub uses centrifugal force to wring out more water from the clothes by spinning at several hundred Rotations per minute (RPM). This is called a 'Spin phase'.

→ The actuator part of the washing machine consists of a motorised agitator, tumble tub, water drawing pump and in let value to control the flow of water in to the unit.

→ The sensor part consists of the water temperature sensor, level sensor, etc

→ The control part contains a microprocessor / microcontroller based board with interfaces to the sensors and actuators.

→ The sensor data is fed back to the control unit and the control unit generates the necessary actuator outputs.

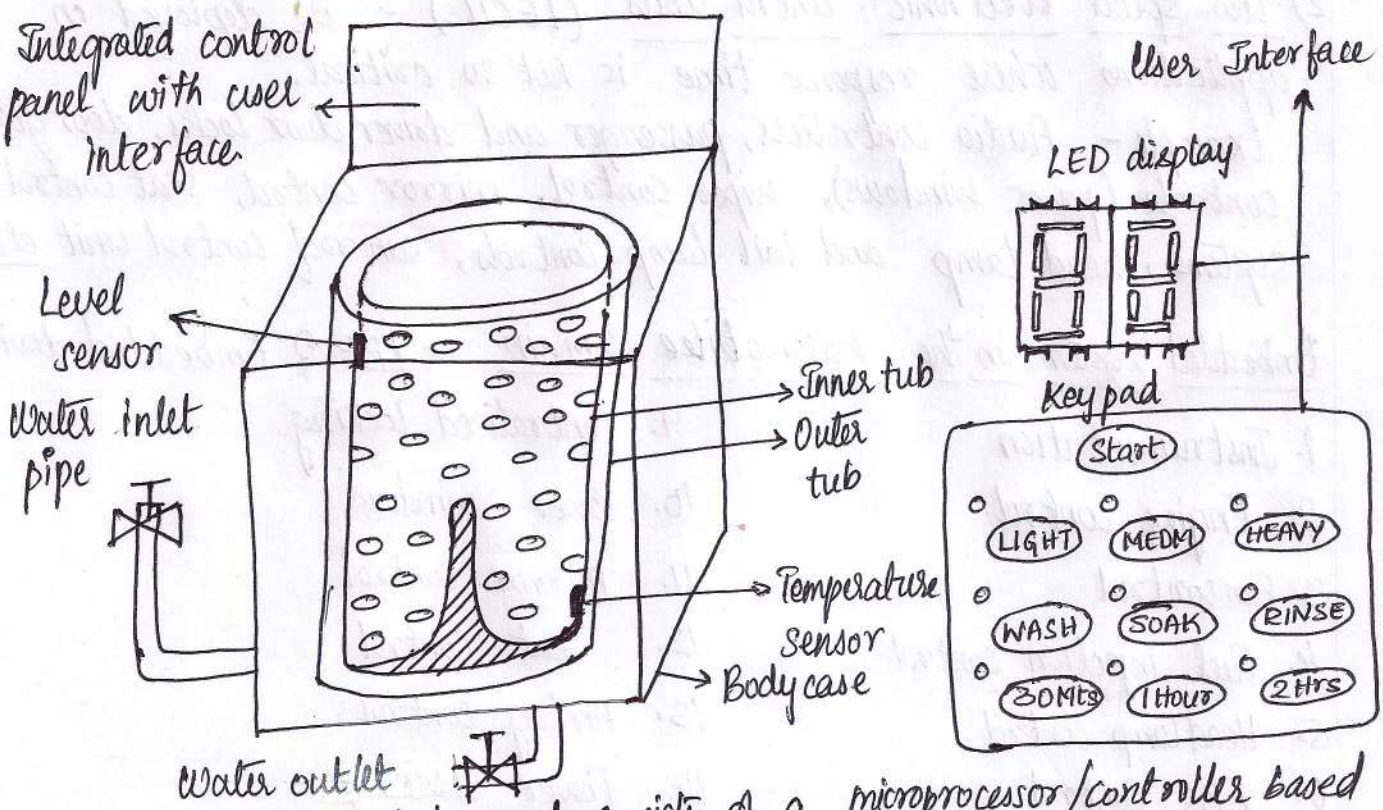
→ The control unit also provides connectivity to user interfaces like keypad for setting the washing time, selecting the type of material to be washed like light, medium, heavy duty etc.

The functional diagram of Washing machine

→ The basic controls consists of a timer, cycle selector, mechanism, water temperature selector, load size selector and start button.

→ The mechanism includes the motor, transmission, clutch, pump, agitator, inner tub, outer tub and water inlet valve.

→ Water inlet valve connects to the water supply line using at home and regulates the flow of water in to the tub.



→ The integrated control panel consists of a microprocessor/controller based board with I/O interfaces and a control algorithm running in it.

→ Input interface includes the keyboard which consists of wash type selector - Wash, Spin and Rinse, cloth type selector - Light, Medium, Heavy duty and wash time setting etc.

→ Output interface consists of LED/LCD displays, ~~etc~~ status indication LEDs etc. connected to the I/O bus of the controller.

Automotive - Domain - Specific Examples of Embedded System

→ Automotive embedded systems are the one where electronics take control over the mechanical systems. like simple mirror and wiper controls to complex air bag controllers and antilock brake systems (ABS).

→ These are normally built around microcontrollers or DSPs or a hybrid of the two and are generally known as Electronic Control Units. (ECUs) and its 2 types —

1) High-Speed Electronic Control Units (HECUs) - are deployed in critical control units requiring fast response.

They include fuel injection systems, antilock brake systems, engine control, electronic throttle, steering controls, transmission control unit and central control unit.

2) Low-Speed Electronic Control Units (LECUs) - are deployed in applications where response time is not so critical.
Example - Audio controllers, passenger and driver door locks, door glass controls (power windows), wiper control, mirror control, seat control systems, head lamp and tail lamp controls, sun roof control unit etc.

Embedded system in the Automobile domain - List of Embedded devices

1. Instrumentation
2. Engine control
3. Fan control
4. Fuel injection control
5. Head lamp control
6. ABS Control
7. Wiper control
8. Suspension control
9. Centralized locking
10. Power windows
11. Mirror control
12. Seat Control
13. Airbag control
14. Power steering
15. Air conditioner

Automotive Communication Buses - Types of serial interface buses deployed in automotive embedded applications are-

1) Controller Area Network (CAN) Bus - It was originally proposed by Robert Bosch. It supports medium speed (data rates upto 125 Kbps), and high speed (data rates up to 1 Mbps) data transfer.
→ It is an event-driven protocol interface with support for error handling in data transmission.
→ It is generally employed in safety system like airbag control; power train systems like engine control and Antilock Brake System (ABS); and navigation systems like GPS.

2) Local Interconnect Network (LIN) bus is a single master multiple slave (upto 16 independent slave nodes) communication interface.
→ It is a low speed, single wire communication interface with support for data rates up to 20 kpbs and is used for sensor/actuator interfacing.
→ It is employed in applications like mirror controls, fan controls, seat positioning controls, window controls, and position controls where response time is not a critical issue.

3) Media-Oriented System Transport (MOST) Bus - is mostly used for automotive audio/video ~~inte~~ equipment interfacing.
→ It is a multimedia fibre-optic point-to-point network implemented in a star, ring or daisy chained topology over optical fibre cables.
→ It consists of physical (electrical and optical parameters) layer, application layer, network layer, and media access control.
→ It is an optical fibre cable connected between the Electrical Optical Converter (EOC) and Optical Electrical Converter (OEC).

Key players of the Automotive Embedded Market

1) Silicon Providers are responsible for providing the necessary chips ~~which~~ which are used in the control application development.
The chip may be a standard product like microcontroller or DSP or ADC/DAC chips.
* Analog Devices - provider of Digital signal processing chips, precision analog microcontrollers, programmable inclinometer, LED drivers, audio systems, GPS/navigation systems etc.
* Xilinx - Supplier of high performance FPGAs, CPLDs and automotive specific IP cores for GPS navigation systems, driver information systems, distance control, collision avoidance, voice recognition etc.
* Atmel - Supplier of cost-effective high density Flash controllers and memories. It provides a series of high performance microcontrollers like ARM, AVR and 80C51.
* Maxim/Dallas - supplier of analog, digital and mixed signal products like Microcontrollers, ADC/DAC, amplifiers, comparators, regulators etc.
* NXP Semiconductor - supplier of 8/16/32 Flash microcontrollers.
* Renesas - provides of high speed microcontrollers and Large scale Integration (LSI) technology for car navigation systems with three transfer speeds: high, medium and low.
* Texas Instruments - supplier of microcontroller, DSPs and automotive communication control chips for Local Interconnect Network (LIN) bus products.

* Infineon - Supplier of high performance microcontrollers and customised application specific chips.
* NEC - provider of high performance microcontrollers.

2) Tools and Platform providers -

They are manufacturers and suppliers of various kinds of development tools and Real Time Embedded Operating Systems for developing and debugging different control unit related applications.

* ENEA - is the developer of the OSE RTOS which supports both CPU & DSP and also support multi-core and fault-tolerant system development.
* The Mathworks - is the leading developer and supplier of technical software. It offers a wide range of tools, consultancy and training for numeric computation, visualisation, modelling and simulation.

MATLAB is a high level programming language and environment for technical computation and numerical analysis.

* Keil Software - is a powerful embedded software design tool for 8051 and C166 family for microcontrollers.
* Lauterbach - supplier of debug tools, providing support for processors in the automotive market.
* ARTiSAN - is the supplier of collaborative modelling tools for requirement analysis, specification, design and development of complex applications.
* Microsoft - It is a RTOS platform provider for automotive embedded applications. ex- Windows CE provides support for automotive application developers.

3) Solution Providers - supply OEM (Original Equipment Manufacturer) and complete solutions for automotive applications

* BOSCH Automotive - provides complete automotive solution ranges from body electronics, diesel engine control, gasoline engine control, powertrain systems, safety systems, in car navigation systems and infotainment systems.
* DENSO automotive - is an OEM & solution provider for engine management, climate control, body electronics, driving control & safety, hybrid vehicles, embedded infotainment and communications.
* Infosys Technologies - is a solution provider for automotive embedded hardware with competitive edge in integrating technology change through cost-effective solutions.
* Delphi is the solution provider for engine control, safety, infotainment etc. and OEM for spark plugs, bearings etc.

Hardware Software Co-Design and Program Modelling

→ In traditional embedded system development approach, the hardware software partitioning is done at an early stage.
→ Software group engineers take care of the software architecture where Hardware group engineers are responsible for building the hardware (required for the product).
→ During co-design process, the product requirements captured from the customer are converted into system level needs or processing requirements.
→ System level processing requirements are then transferred into functions as functional requirements which can be simulated and verified against performance and functionality.
→ Architecture design follows the system design. The partition of system level processing requirements into hardware and software takes place during the architecture design phase.
→ Each system level processing requirement is mapped as either hardware and/or software requirements.
→ The partitioning is performed based on the hardware-software trade offs.

Fundamental issues in Hardware Software Co-design

i) Selecting the model -
→ In HW SW Codesign, models are used for capturing and describing the system characteristics.
→ A model is a formal system consisting of objects and composition rules. (It is hard to make a decision on which model should be followed in a particular system design)
→ Objective varies with each phase (where at specification stage, ~~only~~ the functionality of the system is in focus and not the implementation information.)

ii) Selecting the Architecture -
→ The architecture specifies how a system is going to implement in terms of the number of and types of different components and the interconnection among them.

→ Controller architecture, Datapath architecture, Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), Very Long Instruction Word Computing (VLIW), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD) etc. are the commonly used architectures in system design.

Note:

* Controller architecture implements the finite state model using a state register (for present state) and two combinational circuits (for next state and output).
* Datapath architecture implements the data flow graph model where the output is generated as a result of a set of predefined computations on the input data.
* Finite State Machine Datapath architecture combines the controller architecture with datapath architecture. It implements a controller with datapath. The controller generates the control input where as the datapath processes the data.
* Complex Instruction Set Computing (CISC) architecture - uses an instruction set representing complex operations.
* Very Long Instruction Word (VLIW) architecture - implements multiple functional units (ALUs, multipliers, etc) in the datapath, like one standard instruction per functional unit of the datapath.
* Parallel processing architecture implements multiple concurrent Processing Elements (PEs) and each processing elements may associate a datapath containing register and local memory.

Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) architectures are examples for parallel processing architectures.

(iii) Selecting the Language -

→ A programming language captures a 'Computational model' and maps it into architecture.

→ A model can be captured using multiple programming languages like C, C++, C#, Java, etc for software implementations and languages like VHDL, System C, Verilog, etc for hardware implementations.

(iv) Partitioning system requirements into hardware and software

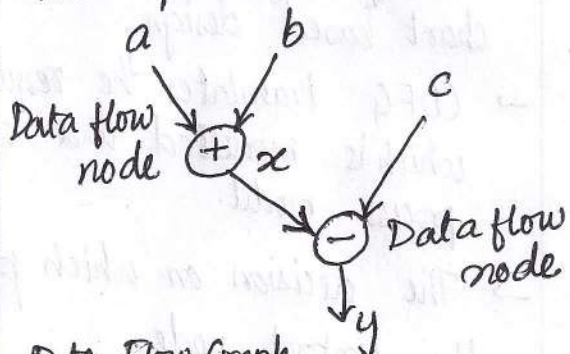
→ To implement the system requirements in either hardware (or) software (firmware).
→ Various hardware software trade-offs are used for making a decision on the hardware-software partitioning.
→ It is a tough decision making task to figure out which one to opt.

Computational Models in Embedded Design

Data flow Graph (DFG) model, State Machine model, Concurrent Process model, Sequential Program model, Object Oriented model, etc are the commonly used computational models in embedded system design.

(1) Data Flow Graph (DFG) model -

→ It translates the data processing requirements into a data flow graph
→ It is a data driven model in which the program execution is determined by data.. It translates the program as a single sequential process ~~eqv~~ execution.
→ DFG model emphasises on the data and operations on the data which transforms the input data to output data.
→ It is a visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows.
→ An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data.
→ Suppose one of the functions in our application contains the computational requirement $x = a + b$; and $y = x - c$.
→ DFG model in figure ~~illus~~ illustrates the implementation of and its requirements
→ DSP applications are typical examples where an embedded applications needs computational intensive and data driven model using DFG model.



Data Flow Graph (DFG) model

→ In a DFG model, a data path is the data flow path from input to output.

→ A DFG model is ~~said~~ of two types - Acyclic DFG (ADFG) and Non-acyclic DFG.

(i) Acyclic DFG - it does not contain multiple values for the input variable and multiple output values for a given set of inputs.

(ii) Non-acyclic DFG - it contains multiple values for the input and multiple output values for ~~example~~ example - like feedback inputs (Output is fed back to Input), events.

(2) Control Data Flow Graph / Diagram (CDFG)

→ It is used for modelling applications involving conditional program execution. It contains both data operations and control operations.

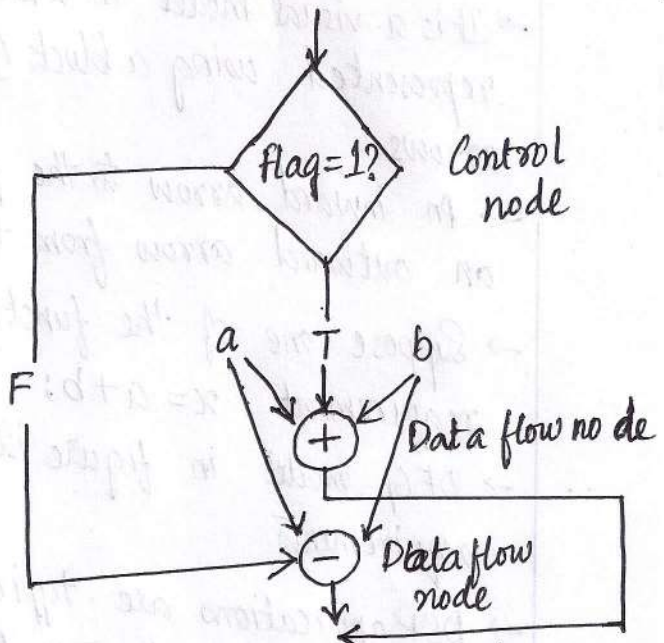
→ It uses Data flow graph (DFG) as element and conditional (constructs) as decision makers.

→ It contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes.

→ CDFG model implementation illustrates the requirement as ~~for~~ follows -

* If flag = 1, x = a + b; else y = a - b;

* It contains a decision making process.



→ The control node is represented by a 'Diamond' block which is the decision making element in a normal flow chart based design.

→ CDFG translates the requirement, which is modelled to a concurrent process model.

→ The decision on which process is to be executed is determined by the control node.

(3) State Machine Model

→ It is used for modelling reactive or event-driven embedded systems whose processing behaviour are dependent on state transitions.

Embedded systems used in the control and industrial applications are typical examples for event driven systems.

→ The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'.

* State - It is a representation of a current situation. An event is an i/p to state.
* Event - acts as stimuli for state transition.
* Transition - is the movement from one state to another.
* Action - is an activity to be performed by the state machine.

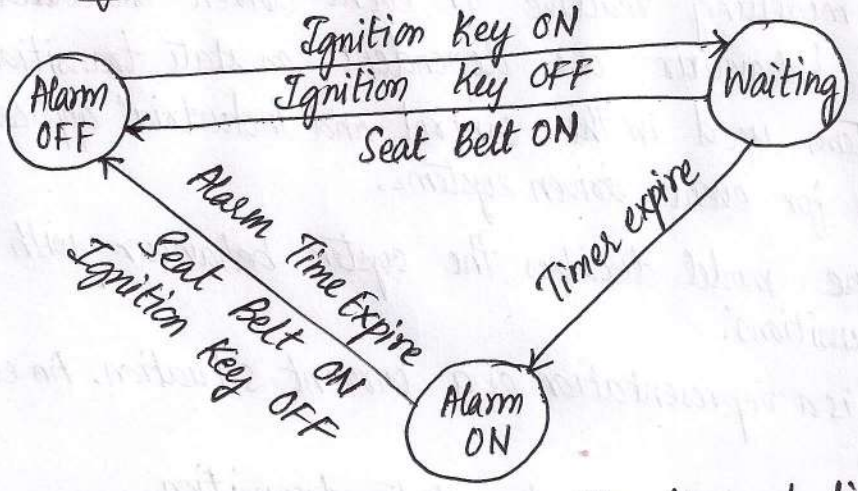
→ A Finite State Machine (FSM) model is one in which the number of states are finite where the system is described using a finite no. of possible states.

Example - Design of an embedded systems for driver/passenger 'Seat Belt Warning' in an automotive using the FSM model.

→ The system requirements are captured as
(i) When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
(ii) The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Here the states are - 'Alarm OFF', 'Waiting' and 'Alarm ON' and the events are 'Ignition Key ON', 'Ignition Key OFF', 'Timer Expire', 'Alarm Time Expire' and 'Seat Belt ON'.

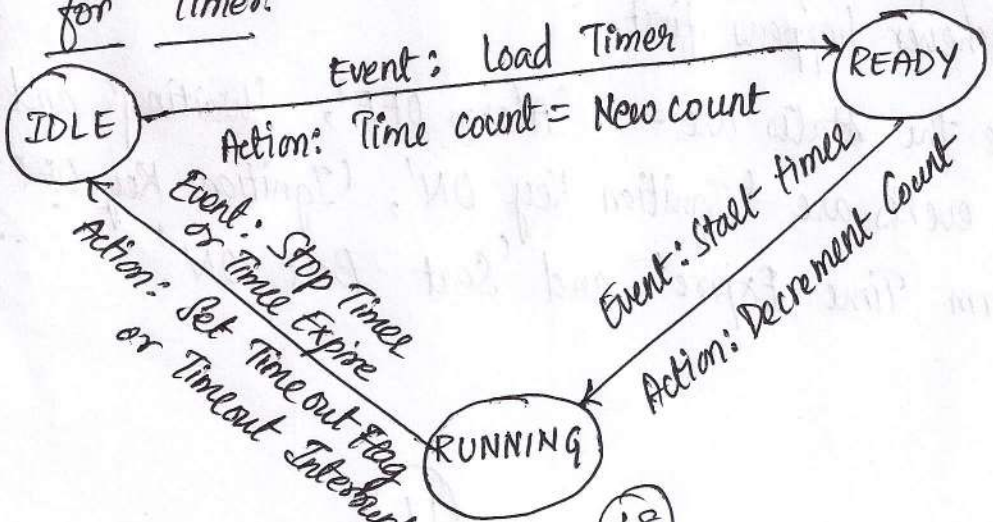
① FSM Model for Automatic seat belt warning system



1) The 'Ignition Key ON' event triggers the 10 second timer and transitions the state to 'Waiting'.
2) If a 'Seat Belt ON' or 'Ignition Key OFF' event occurs during the wait state, the state transitions in to 'ALARM OFF'.
3) When the wait timer expires in the waiting state, the event 'Timer expire' is generated and it transitions the state to 'Alarm ON' from the 'Waiting' state.
4) The 'Alarm ON' state continues until a 'Seat Belt ON' or 'Ignition Key OFF' event or 'Alarm Time Expire' event, whichever occurs first.
5) The occurence of any of these ~~state~~ events transitions the state to 'ALARM OFF'.

Note: The wait state is implemented using a timer. The Timer also has certain set of states and events for state transitions.

② FSM model for Timer.



→ The timer consists IDLE, READY, or RUNNING states.
→ During the normal condition when the timer is not running, it is said to be in the 'IDLE' state.
→ The Timer is said to be in the 'READY' state when the timer is loaded with the count corresponding to the required time delay.
→ The timer remains in the 'READY' state until a 'Start timer' event occurs.
→ The Timer changes its state to 'RUNNING' from the 'READY' state on receiving a 'start timer' event and remains in the 'RUNNING' state until the timer count expires or a 'Stop timer' event occurs.
→ The Timer state changes to 'IDLE' from 'RUNNING' on receiving a 'Stop Timer' or 'Timer Expire' event.

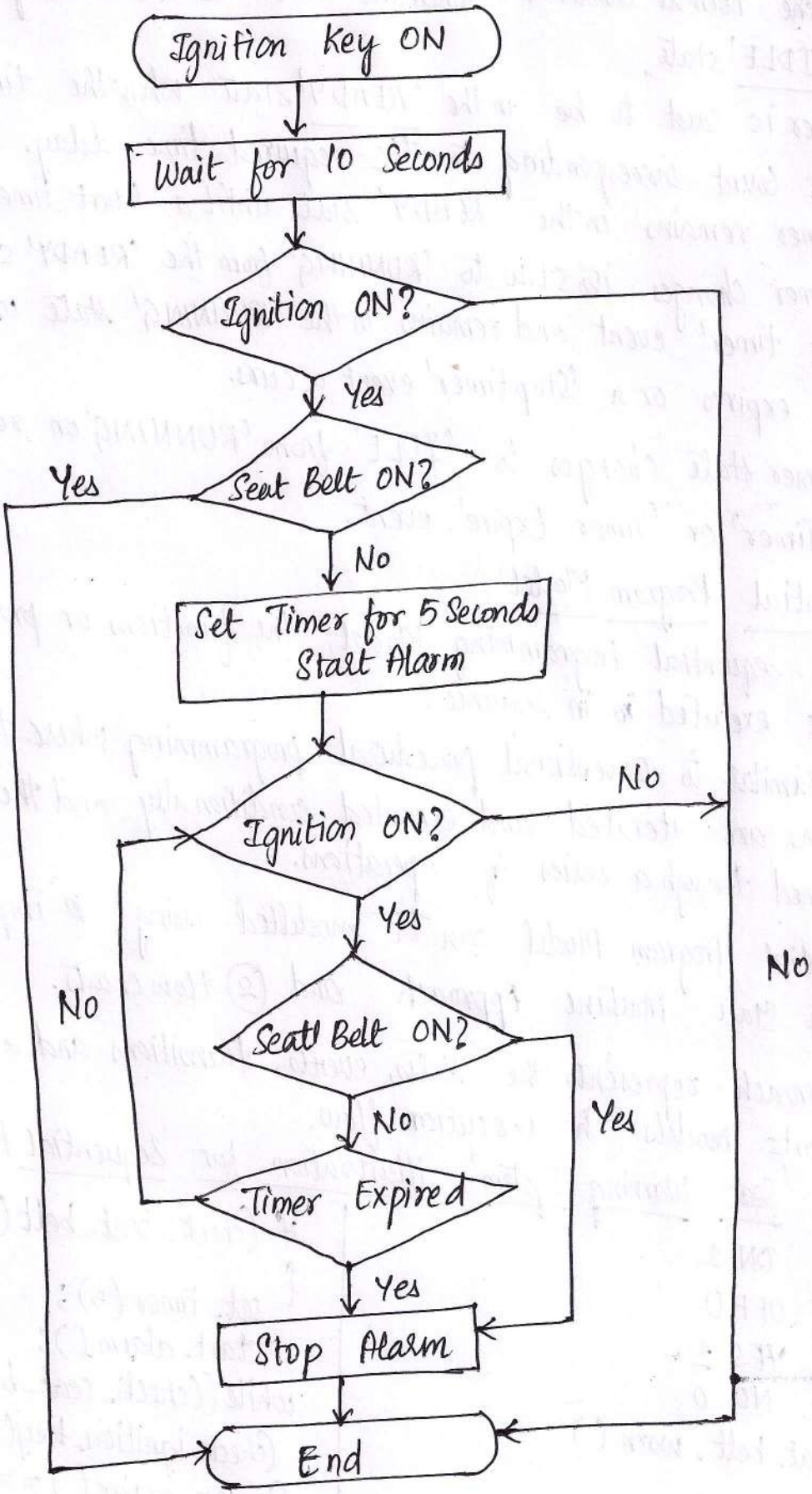
(4) Sequential Program Model

→ In the Sequential Programming Model, the functions or processing requirements are executed in sequence.
→ It is similar to conventional procedural programming where the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
→ Sequential Program Model can be modelled using 2 important tools like (1) Finite State Machine Approach and (2) Flow charts.
→ FSM approach represents the states, events, transitions and actions.
→ Flowcharts models the execution flow.

Example - 'Seat Warning System' illustration for Sequential Program Model

```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn()
{
  wait_10 sec();
  if (check_ignition_key() == ON)
  {
    if (check_seat_belt() == OFF)
    {
      set_timer(5);
      start_alarm();
      while ((check_seat_belt() == OFF) &&
        (check_ignition_key() == OFF) &&
        (timer_expire() == NO));
      stop_alarm();
    }
  }
}
```

(2) Flow chart based Illustration of 'Seat Belt Warning System'



(5) Concurrent / Communicating Process Model

→ The concurrent or communicating process model models concurrently executing tasks/processes.

→ It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution.

→ Concurrent processing model requires additional overheads in task scheduling, task synchronisation and communication.

Example - Concurrent processing model used to implement the 'Seat Belt Warning' system - It is split into 5 Tasks.

1. Timer tasks for waiting 10 seconds (wait timer task)
2. Task for checking the ignition key status (ignition key status monitoring task)
3. Task for checking the seat belt status. (seat belt status monitoring task).
4. Task for starting and stopping the alarm (alarm control task).
5. Alarm timer task for waiting 5 seconds (alarm timer task)

(a) Tasks for 'Seat Belt Warning System'

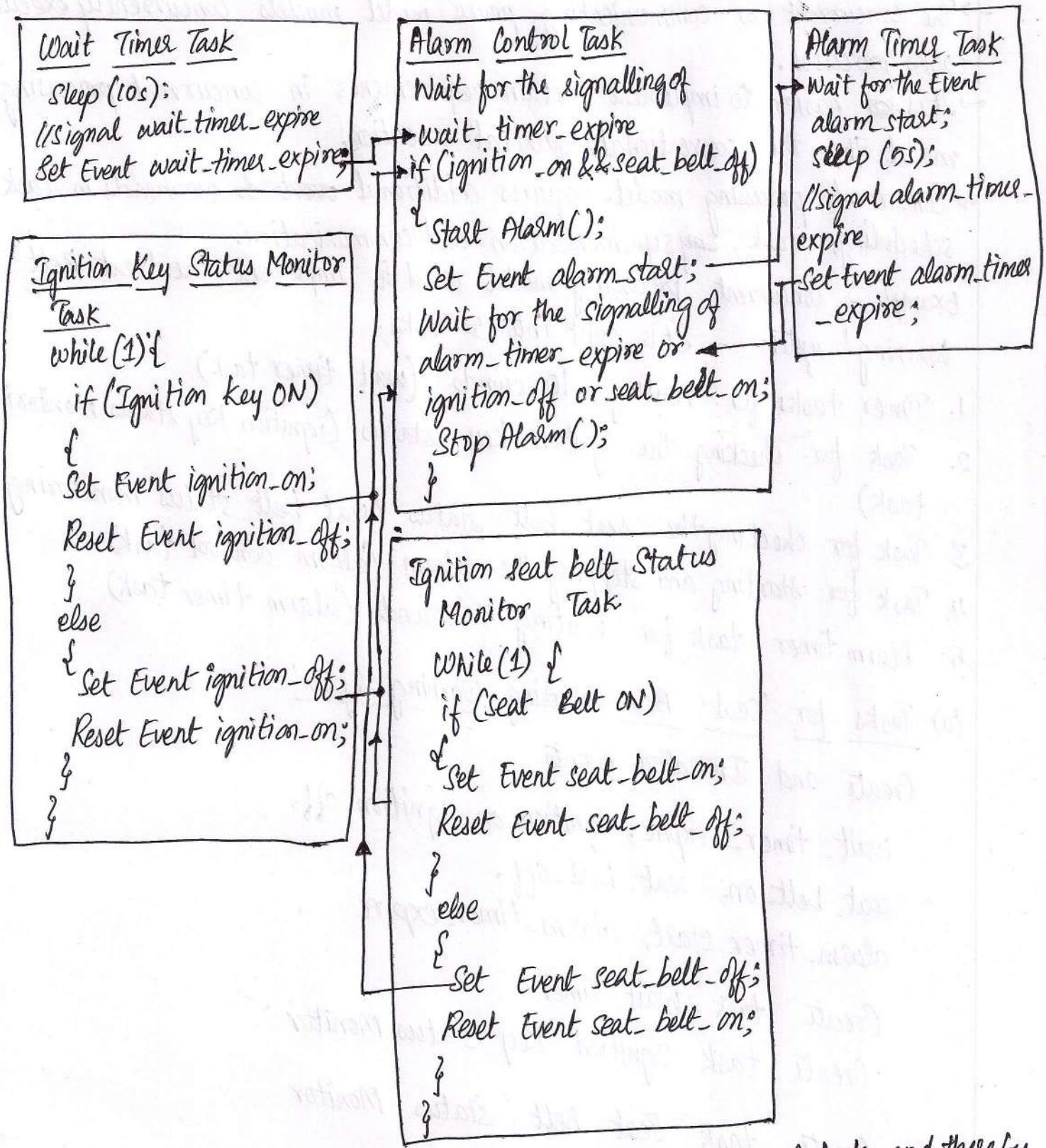
Create and Initialize events

wait_timer_expire, ignition_on, ignition_off,
seat_belt_on, seat_belt_off,
alarm_timer_start, alarm_timer_expire

Create task Wait Timer
Create task Ignition Key Status Monitor
Create task Seat Belt Status Monitor
Create task Alarm Control
Create task Alarm Timer

→ Concurrent processing model requires additional overheads in task scheduling, task synchronization and communication.

(b) Concurrent processing Program model for 'Seat Belt Warning System'



→ Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilisation when the task involves I/O waiting, sleeping for specified duration etc.

→ In Concurrent execution, the task is split into multiple subtasks, it is possible to tackle CPU usage effectively, when the subtask under execution goes to a wait or sleep mode, by switching the task execution.

⑥ Object - Oriented Model

→ The Object-oriented model is an object based model for modelling system requirements. The concept of object and class brings abstraction, hiding and protection.

→ It disseminates a complex software requirement into simple well defined pieces called Objects.

→ Object is an entity used for representing or modelling a particular piece of the system. Each object is characterised by a set of unique behaviour and state.

→ A Class is an abstract description of a set of objects and it can be considered as a 'blue print' of an object

→ It represents the state of an object through member variables and object behaviour through member functions.

→ The member variables and member functions of a class can be private, public or protected.

* Private member variables and functions are accessible only within the class, ~~where as~~

* Public variables and functions are accessible within the class as well as outside the class.

* Protected variables and functions are protected from external access.

Embedded Firmware Design and Development -

Embedded firmware design approaches is dependent on the complexity of the functions to be performed, the speed of operation required etc.

Two basic approaches of Embedded firmware design are -

① 'Conventional Procedural Based ~~Firmware~~ Firmware Design' or 'Super Loop Model'

② 'Embedded Operating System (OS) Based design'.

① The Super loop Based Approach

→ It is very similar to a conventional procedural programming where the code is executed task by task.

→ It is adopted for applications that are not time critical and where the response time is not so important (for example - embedded systems where missing deadlines are acceptable).

→ It is procedural where task listed at the top of the program code is executed first and the tasks just below top are executed after completing the first task.

→ In a multiple task based system, each task is executed in serial in this approach.

The firmware execution flow —

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

The operational sequence in terms of a 'C' program code

```
void main ()
{
  Configurations ();
  Initializations ();
  while (1)
  {
    task 1 ();
    task 2 ();
    :
    task n ();
  }
}
```

→ This repetition is achieved by using an infinite loop, here the while 1 { } loop. It is also refered to as 'Super loop based Approach'

→ Since 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.

→ Superloop based design is simple and straight forward design without any OS related overheads.

→ It doesn't require an Operating System where there is no need for scheduling which task is to be executed and assigned priority to be each task.

→ In a Super loop based design, the priorities are fixed and the order in which the tasks to be executed are also fixed.

→ The Code performing these tasks will be residing in the code memory without an operating system image.

→ Application - It is deployed in low-cost embedded products where response time is not time critical.

Example - An Electronic video game toy contains keypad and display unit.

Note - (* The program running inside the product may be designed in such a way that it reads the keys to detect whether the user has given any input and if any key press is detected the graphic display is updated.

* If application misses a key press, it is not a critical issue and it will be treated as a bug in the firmware).

→ Drawbacks

(1) Any failure in any part of a single task will affect the total system.

Example - If the program hangs up at some point while executing a task, it will remain there forever and ultimately the product stops functioning.

(Remedial measures - Hardware and Software Watch Dog Timers (WDTs) helps in coming out from the loop when an unexpected failure occurs (or) when the processor hangs up. and it may cause additional hardware cost and firmware overheads.)

(2) It lacks of real timeliness. If the no. of tasks to be executed within an application increases, the time at which each task is repeated also increases.

It brings the probability of missing out some events.

2) The Embedded Operating System (OS) Based Approach

→ It contains operating systems like General Purpose Operating system (GPOS) or a Real Time Operating Systems (RTOS) to host the user written application firmware.

General Purpose Operating System (GPOS)

→ The General Purpose OS (GPOS) based design is very similar to a conventional PC based application development where the device contains an operating system like Windows / Unix / Linux etc for Desktop PCs.

→ Applications are created on top and running over GPOS.

Examples - ① Microsoft Windows XP Embedded is an example of GPOS used in embedded product development.

② Microsoft Windows XP, Embedded products ~~examp~~ examples are Personal Digital Assistants (PDAs), Handheld devices / Portable devices and Point of Sales (POS) terminals.

→ The OS supports APIs (Application programming Interface) for developing applications on top of the OS where the use of GPOS in ~~terms~~ embedded products merges the demarcation of Embedded Systems and General Computing systems.

→ OS based applications also require 'Driver software' (similar to different hardware specific drivers) for different hardware present on the board to communicate with them.

Real Time Operating System (RTOS)

→ It is employed in embedded products demanding Real-Time response. and It responses in a timely and predictable manner to events.

→ It contains a Real Time Kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc.

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

Examples

① → 'Windows CE', 'pSOS', 'VxWorks', 'ThreadX', 'micro C/OS-II', 'Embedded Linux', 'Symbian' etc are examples of RTOS employed in embedded product development.

② → Mobile phones, PDAs, handheld devices, etc. are examples of 'Embedded products' based on RTOS.

Embedded Firmware Development Languages

Assembly Language based Development

→ 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.

→ 'Assembly language' is the human readable notation of 'machine language' where it is a processor understandable language. Machine language is made readable by using specific symbols called 'Mnemonics'.

→ Processor deals only with binaries (1's and 0's). Machine language is a binary representation and it consists of 1's and 0's.

→ Assembly language instructions are written one per line. A machine code program thus consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand).

→ Each line (or) statement of an assembly language program is split into four fields as given below-

LABEL OPCODE OPERAND COMMENTS

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

→ LABELs are used for representing subroutines names, jump locations, memory location, address of a program, & code portion etc in Assembly language programming.

→ Labels are always suffixed by a colon and begin with a valid character where it can contain number from 0 to 9 and special character - (underscore).

→ The general format of an assembly language instruction is an Opcode followed by Operands.

→ 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.

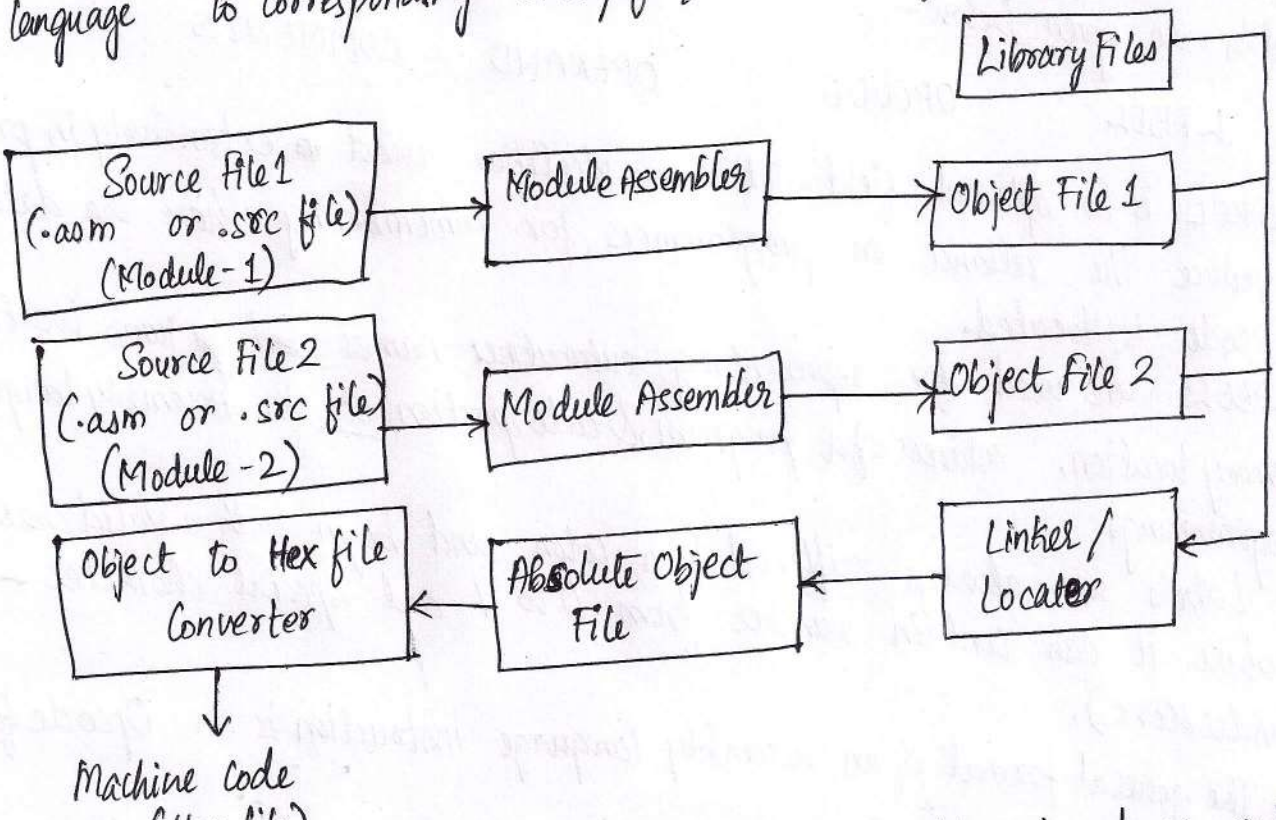
→ Some of the Opcode implicitly contains the operand and in such situation no operand is required (and it is not necessary that all opcode should have operands following them).

→ It is written in assembly code is saved as .asm (Assembly file) file or an .src (source) file.
→ Similar to 'C' and other high level language programming, multiple source files can be saved called as 'Modules' in assembly language programming.
→ Each module is represented by an '.asm' or '.src' file similar to the '.c' files in C programming. It is known as 'Modular programming'.
→ Modular programming is used only when the program is too complex or too big. where ~~it is employed when~~ the entire code is divided into submodules and each module is made reusable.
→ Modular programs are usually easy to code, debug and alter.

Assembly language to machine language conversion process.

① Source File to Object file Translation

→ Translation of assembly code to machine code is performed by assembler.
→ The various steps involved in the conversion of a program written in assembly language to corresponding binary file/ machine language is as shown in figure.



→ Each source module is written in Assembly and is stored as .asm file.
→ Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions.

→ ~~On successful~~ Assembling of each .src/.asm file a corresponding object file is created with extension '.obj'.

→ The object file does not contain the absolute address of where the generated code needs to be placed on the program memory and hence it is called a re-locatable segment.

→ It can be placed at any code memory location and it is the responsibility of the linker / locater to assign absolute address for this module.

→ Absolute address allocation is done at the absolute object file creation stage.

→ Each module can share variables and subroutines (functions) among them.

→ Exporting a variable/function from a module (making a variable/function from a module available to all other modules) is done by declaring that variable / function as PUBLIC in the source module.)

(2) Library File Creation and Usage

→ Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time.

→ Library file is some kind of source code hiding technique.

→ When the linker processes a library, only those object modules in the library that are necessary to create the program are used.

→ Library files are generated with extension '.lib'. Example- 'LIB51' from Keil software.

(3) Linker and Locator -

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

→ Linker generates an absolute object module by extracting the object modules from the library and (object files created by assembler is generated by assembling the individual modules of a project).

→ The absolute object file is used for creating hex files for dumping into the code memory of the processor / controller.

Example- 'BL51' from Keil Software for a Linker and 'A51' Locater for A51 Assembler / C51 Compiler for 8051 specific controller.

④ Object to Hex file converter –

→ It is the final stage in the conversion of Assembly language (mnemonics) to machine language / code.

→ Hex file is the representation of machine code and it is dumped into code memory of the processor / controller.

→ Hex files are ASCII files that contain a hexadecimal representation of target application. It is created from the final 'Absolute Object File' using the Object to Hex file Converter utility.

Advantages of Assembly language Based Development –

① Efficient Code memory and Data Memory Usage (Memory Optimisation) :–

(→ Developer should be well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations.)

→ It leads to less utilisation of code memory and efficient utilisation of data memory.

② High Performance –

Optimised code not only improves the code memory usage but also improves the total system performance.

③ 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.

④ Code Reverse Engineering –

Reverse engineering is the process of understanding the technology behind a product by extracting the information from a finished product.

Drawbacks of Assembly language based development –

① High Development Time –

→ Assembly language is much harder to program than high level languages because developer must have thorough knowledge of the architecture, memory organisation and register details of the target processor in use.

② Developer Dependency -

In assembly language programming, the developers will have the freedom to choose the different memory location and registers of the target processor in use.

③ Non-Portable -

Target applications written in assembly instructions are valid only for that particular family of processors (Intel x86 processors) and cannot be reused for another target processors/controllers (ARM11 processors).

② High Level Language Based Development

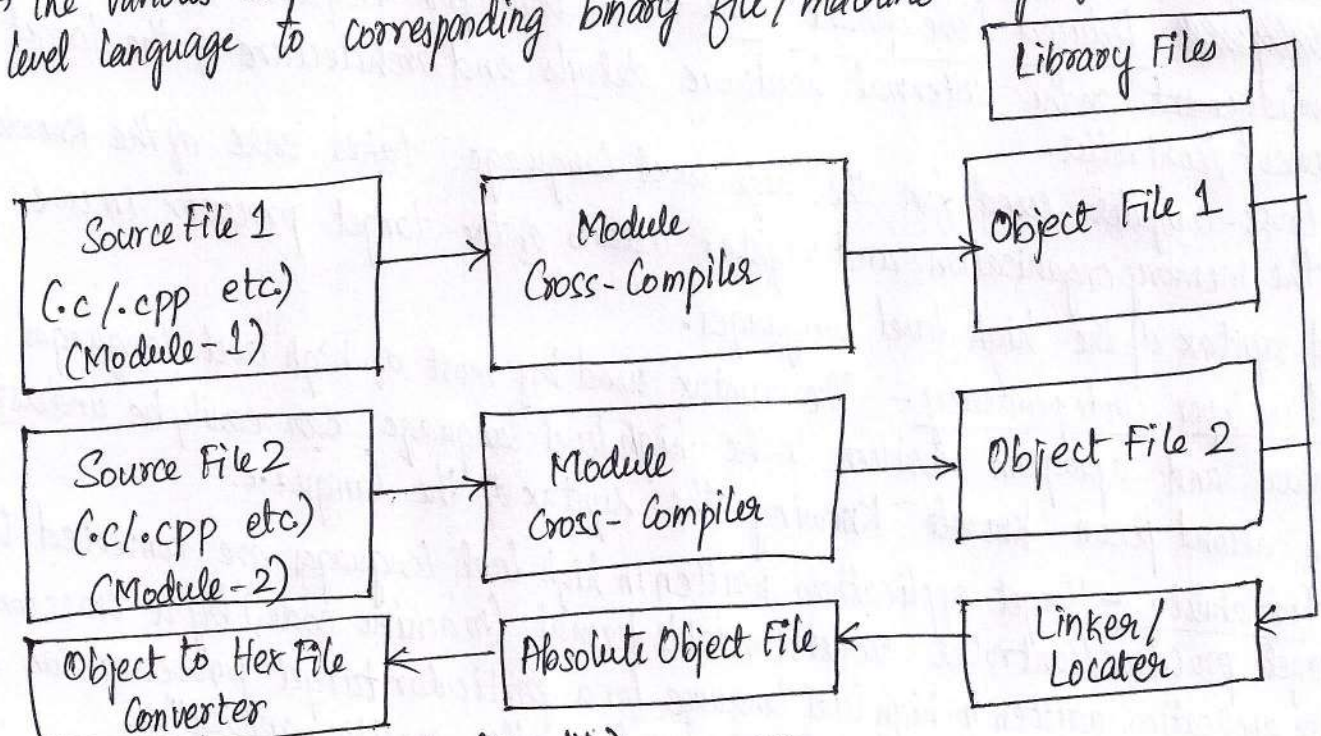
→ Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture

→ High level language (like C, C++ or Java) with a supported Cross compiler converts the application developed in high level language to target processor specific assembly code.

High level language to Machine language conversion process.

→ The program written in any of the high level language is saved with the extension (.c for C, .cpp for C++ etc).

→ The various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language as shown in figure.



→ Most of the high level languages support modular programming modular approach and hence multiples source files called modules written in high level languages.

→ The source file corresponding to each module is represented by a file with an extension (.c or .cpp).

→ Translation of High level language source code to executable object code is done by a Cross compiler.

→ Each high level language should have a cross-compiler for converting the high level language source code into the target processor machine code.

→ Without Cross-compiler support a high level language cannot be used for embedded firmware development.

Example - C51 Cross compiler from Keil software is an example for 'C' language for the 8051 family of micro controller.

→ Conversion of each module's source code to corresponding object file is performed by the cross compiler.

→ Rest of the steps involved in the conversion of high level language to target processor's machine code are same as that of the steps involved in assembly language based development.

Advantages

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

Cross-compiler used for the high level language takes care of the knowledge of the memory organisation and register details of the target processor in use and syntax of the high level languages.

(2) Developer Independency - The syntax used by most of 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.

(3) Portability - Target applications written in high level languages are converted to target processor/controller understandable format (machine codes) by a cross compiler.

→ An application written in high level language for a particular target processor can be easily converted to another target processor/controller specific application.

Limitations of High level language Based Development

① Some cross-compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions.

② Target images created by such compilers maybe messy and non-optimised interms of performance as well as code size.

③ The time required to execute a task also increases with the number of applications.

④ High level language based code snippets may not be efficient in accessing low level hardware where as hardware access timing is critical in terms of micro or nano seconds.

⑤ The Investment required for high level language based development tools (IDE with cross compiler) is high compared to Assembly language based firmware tools.

③ Mixing Assembly and High level language

→ Embedded firmware development situations may demand the mixing of high level language with Assembly and vice versa.

→ These are usually mixed in three ways -

(i) Mixing Assembly language with High level language,
(ii) Mixing High level language with Assembly, and
(iii) In-line Assembly programming

(i) Mixing Assembly language with High level language (or) (Assembly language with 'C')

→ Assembly routines are mixed with 'C' in situations where the entire program is written in 'C' and the cross compiler in use do not have a built-in support for Interrupt Service Routine(ISR) functions implementation.

→ Here the Programmer wants to take advantage of speed and optimised code offered by machine code generated by hand written assembly rather than cross compiler generated machine code.

→ Writing the hardware/peripheral access routine in processor/controller specific Assembly language and invoking it from 'C'.

→ Mixing 'C' and assembly is little complicated in the sense - the programmer must be aware of how ~~caller 'C'~~ function ~~and the~~ 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.

→ Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross compiler dependent.

(ii) Mixing High Level Language with Assembly (or) ('C' with Assembly language)

→ Mixing the code written in a high level language like 'C' and Assembly language is useful in the following ways -

1. 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.
2. The entire source code is planned in Assembly code for various reasons like optimised code, optimal performance, efficient code memory utilization and proven expertise in handling the Assembly, etc.
(Few portions of the code may be very difficult and tedious to code in Assembly. For example - 16 bit multiplication and division in 8051 Assembly language.)
3. To include built in library functions written in 'C' language provided by the Cross compiler. (for example - Built in Graphics library functions and string operations supported by 'C').

(iii) Inline Assembly

→ It is another technique for inserting target processor / controller specific Assembly instructions at any location of a source code written in high level language 'C'.

→ It avoids the delay in calling an assembly routine from a 'C' code where if assembly instructions to be inserted are put in a subroutine while mixing assembly with 'C'.

→ Special keywords are used to indicate that the start and end of Assembly instructions. The keywords are cross compiler specific for example, C51 uses ~~the key~~ #pragma asm and #pragma endasm - keywords. ~~to indicate~~

Programming in Embedded C

→ The use of conventional 'C' language and its extensions for programming Embedded systems is called as 'Embedded C' programming.

→ Programming in 'Embedded C' is different from conventional Desktop application development using 'C' language for a particular OS platform.

→ For a desktop application developer, there are no restrictions imposed in the usage of RAM and ROM as resources available (i.e. are surplus in quantity.)

→ Desktop computers contain working memory in the range of Megabytes and storage memory in the range of Giga bytes.

→ Embedded application developers should develop applications in the best possible way which optimises the code memory and working memory usage as well as performance.

(1) 'C' vs 'Embedded C'

→ 'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support.

→ It offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (like payroll systems, banking applications, etc).

→ The Conventional 'C' language follows ANSI standard and it incorporates various libraries files for different OS.

→ It is a platform specific development where OS provides a specific application called as 'Compiler' which converts the programs written in 'C' to the target processor.

→ Embedded 'C' can be considered as a subset of conventional 'C' language. It supports all 'C' instructions and incorporates a few target processor specific functions/ instructions.

→ It should be noted that the standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded 'C'.

→ The implementation of target processor/controller instructions depends upon both target processor/controller and cross compiler for 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).

(2) Compiler vs. Cross-Compiler

→ Compiler is a software 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.

→ Compilers are generally termed as 'Native Compilers'. A Native compilers generates machine code for the same machine (processor) on which it is running.

→ Here the OS, the compiler program and the application making use of the source code run on the same target processor.

→ (The source code is converted to the target processor specific machine instructions. The development is platform specific which OS as well as target processor on which the OS is running).

→ Cross compilers are the software tools used in cross-platform development applications.

→ In Cross compiler platform, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor.

→ The compiler is running on (or) for an Operating system which is different from the current development environment OS.

→ Kiel C51 is an example for cross-compiler. The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications.

→ Embedded System development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD processors and the same is converted into machine code for any other target processor architecture. ex- 8051, PIC, ARM (processors/microcontrollers) etc.

① Design an automatic Tea/Coffee vending machine based on FSM model for the following requirement.

→ The Tea/Coffee vending is initiated by user inserting a 5 rupee coin. After inserting the coin, the user can either select 'Coffee', or 'Tea' or 'Cancel' to cancel the order and take back the coin.

→ The FSM representation of Automatic Tea/Coffee vending machine

→ It contains four states namely - 'Wait for coin', 'Wait for User Input', 'Dispense Tea' and 'Dispense Coffee'.

→ The event 'Insert Coin' (5 Rupee Coin insertion), transitions the state to 'Wait for User Input'.

→ The system stays in this state until a user input is received from the buttons 'Cancel', 'Tea' or 'Coffee'. (Tea and Coffee are the drink select button).

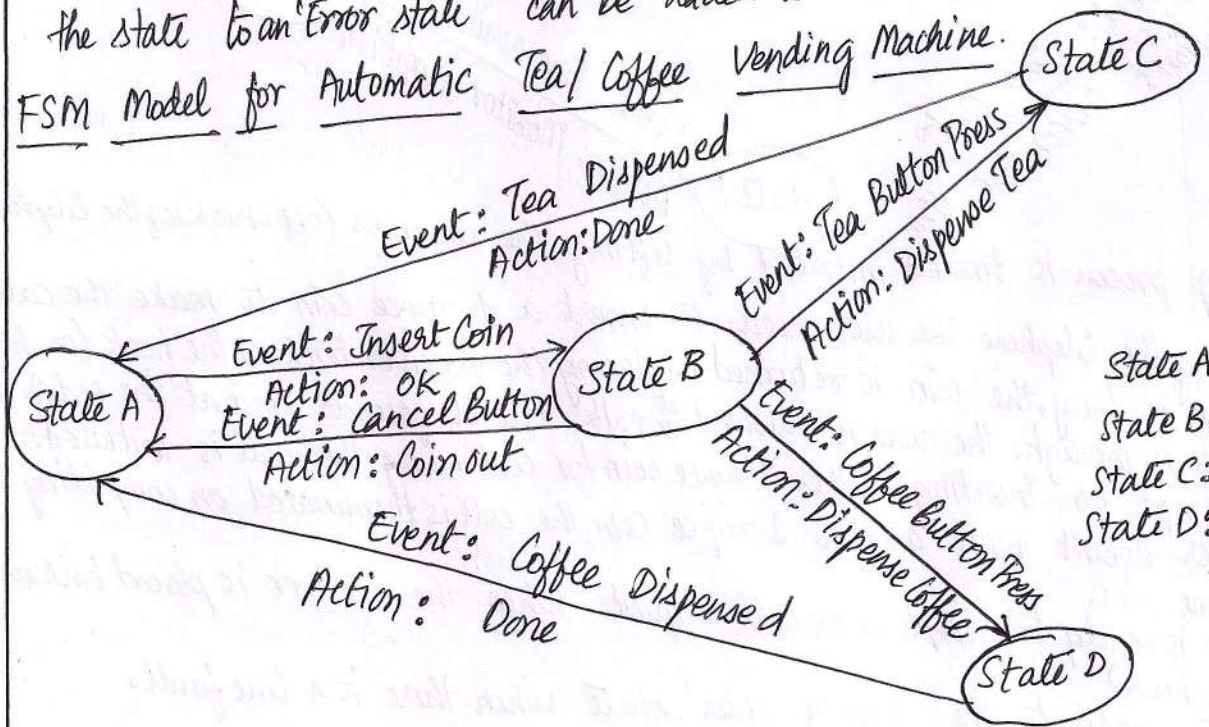
→ If the event triggered in 'Wait state' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin'.

→ If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively.

→ Once the Tea/Coffee vending is over, the respective states transition back to the 'Wait for Coin' state.

→ A few modifications like adding a timeout for the 'Wait state' and capturing events like, 'Water is not available', 'Tea/Coffee Mix not available' and changing the state to an 'Error state' can be added to enhance this design.

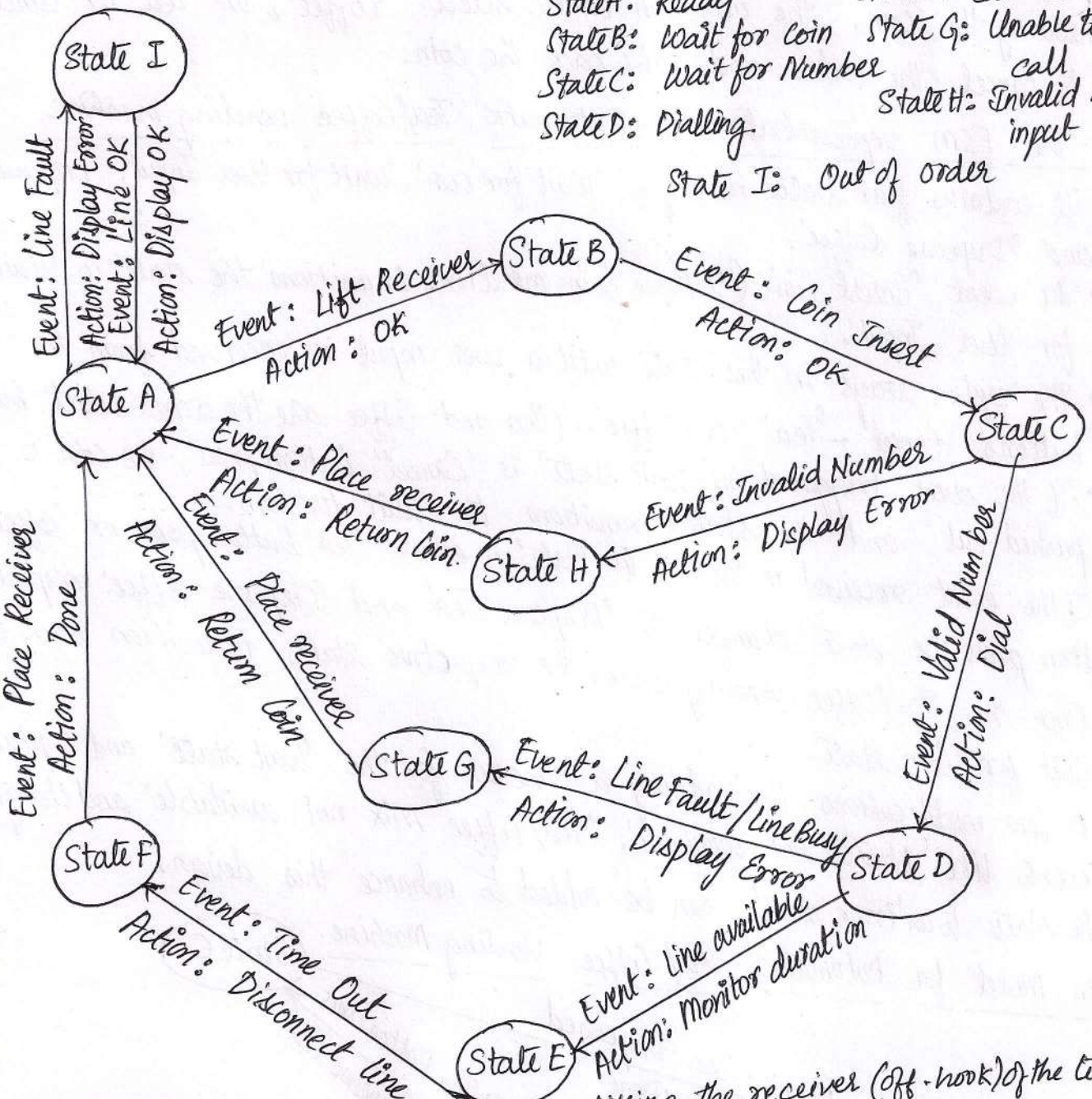
FSM Model for Automatic Tea/Coffee Vending Machine.



2) Design a coin operated public telephone unit based on FSM model for the following requirements

FSM Model for Coin Operated Telephone System

State A: Ready
State B: Wait for coin
State C: Wait for Number
State D: Dialling
State E: Call in progress
State F: Call Terminated
State G: Unable to make call
State H: Invalid number input
State I: Out of order



1. The calling process is ~~limited~~ initiated by lifting the receiver (off-hook) of the telephone unit.
2. After lifting the telephone the user needs to insert a 1 rupee coin to make the call.
3. If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook).
4. If the line is through, the user is allowed to talk till 60 seconds and at the end of 45th second, prompt for inserting another 1 rupee coin for continuing the call is initiated.
5. If the user doesn't insert another 1 rupee coin, the call is terminated on completing 60 seconds time slot.
6. The system is ready to accept new call request when the receiver is placed back on the hook (on-hook)
7. The system goes to the 'Out of Order' state when there is a line fault.

Module-5 RTOS and IDE for Embedded System Design 18.05.2018

B.S.Balaji
Asst. Prof., BGSIT.

Operating System Architecture

→ Operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.

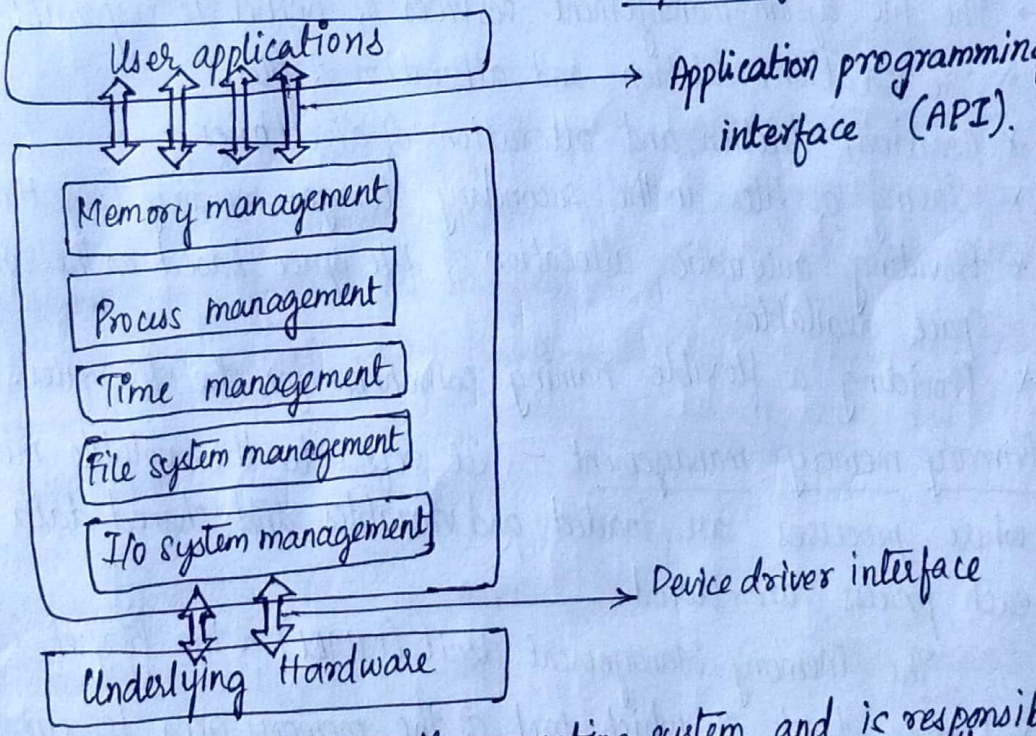
→ It manages the system resources and makes them available to the user applications/tasks on a need basis.

→ The primary functions of an operating system is

* Make the system convenient to use
* organise and manage the system resources efficiently and correctly.

The basic components of an operating system and their interfaces as shown in the architecture -

Operating System Architecture



The Kernel - It is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services.

→ It acts as the abstraction layer between system resources and user applications.

→ It contains a set of system libraries and services.

It contains -

(i) Process management - It deals with managing the processes/tasks. It includes - (i) setting up the memory space for the process,
(ii) loading the process's code into the memory space,
(iii) allocating system resources,
(iv) scheduling and managing the execution of the process,
(v) setting up and managing the Process Control Block (PCB), Inter Process Communication and Synchronisation, process termination/deletion, etc.

(iii) File System Management - It is a collection of relation information. For example- program (source code or executable), text files, image files, word documents, audio/video files etc.

→ The file system management services of Kernel is responsible for
* The creation, deletion and alteration of files.
* Creation, deletion and alteration of directories.
* Saving of files in the secondary storage memory (e.g. Hard disk storage)
* Providing automatic allocation of file space based on the amount of free space available.
* Providing a flexible naming convention for the files.

(ii) Primary memory management - It refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored.

The Memory Management Unit (MMU) of the Kernel is responsible for
* keeping track of which part of the memory area is currently used by which process.
* Allocating and De-allocating memory space on a need basis. (Dynamic memory allocation).

(iv) I/O System (Device) Management - Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.

→ In a well structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the Kernel.

→ The Kernel maintains a list of all the I/O devices of the system.

(v) Secondary storage Management -

→ It deals with managing the secondary storage memory devices, connected to the system.

→ Secondary memory is used as backup medium for programs and data since the main memory is volatile.

→ The Secondary storage management service of Kernel deals with

* Disk storage allocation.
* Disk scheduling (Time interval at which the disk is activated to backup data).
* Free Disk space management.

(vi) Protection systems - Modern operating systems are designed in such a way to support multiple users with different levels of access permissions like 'Administrator', 'Standard', 'Restricted' etc.

→ Protection deals with implementating the security policies to restrict the access to both user and system resources by different applications or processes or users.

→ In Multi users supported operating systems, one user may not be allowed to view or modify the whole / portions of another user's data or profile details.

(vii) Interrupt Handler - Kernel provides handler mechanism for all external / internal interrupts generated by the system.

Based on Kernel design, Kernels can be classified into 'Monolithic' and 'micro'.

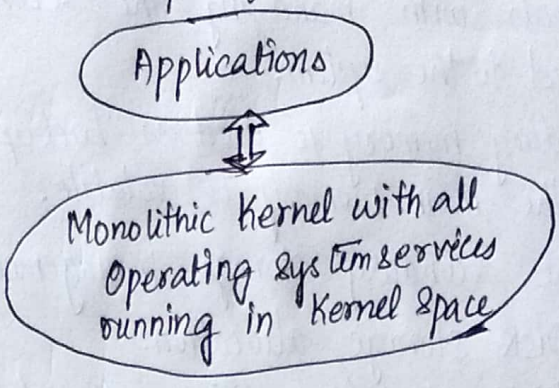
Monolithic Kernel - Monolithic Kernel architecture, all Kernel services run in the Kernel space. where all Kernel modules run within the same memory space under a single Kernel thread.

→ The tight internal integration of Kernel modules in monolithic Kernel architecture allows the effective utilisation of the low-level features of the underlying system.

→ The major drawback of is that any error or failure in any one of the Kernel modules leads to the crashing of the entire Kernel application.

→ LINUX, SOLARIS, MS-DOS kernels are examples of Monolithic Kernel.

Monolithic Kernel architecture

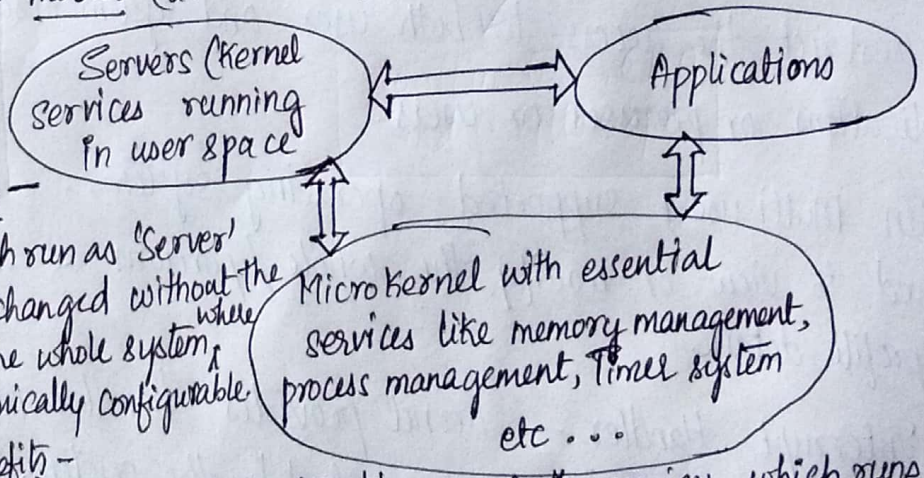


Micro Kernel - It incorporates only the essential set of Operating System services into the Kernel. (The rest of the OS services are implemented in programs known as 'Servers' which runs in user space).

→ It provides a highly modular design and OS-neutral abstraction to the Kernel.

→ Memory management, Process management, Timer systems and interrupt handlers are the essential services, which forms the part of the Micro Kernel.

The Microkernel model (architecture)



② Configurability -

Any services, which run as 'Server' application can be changed without the need to restart the whole system. The system is dynamically configurable where

Advantages / Benefits -

① Robustness - If a problem is encountered in any of the services, which runs as 'Server' application the same can be reconfigured and re-started without the need of the re-start of the entire OS. It is highly useful for systems which demands high availability. (where Servers run on a different memory space, the chances of corruption of kernel services are ideally zero).

Types of Operating Systems.

Operating systems are classified into two types - 1) General Purpose Operating System (GPOS) and 2) Real Time Operating System (RTOS)

① **General Purpose Operating System (GPOS)** - The operating systems are deployed in general computing systems, are refered as General Purpose Operating Systems (GPOS).

→ The GPOS Kernel is more generalised and it contains all kinds of services required for executing generic applications. (where it is quite non-deterministic behaviour.)

→ GPOS services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.

→ These are usually deployed in computing systems where deterministic behaviour is not an important criterion.

Example - Windows XP/ms-DOS.

② **Real Time Operating System (RTOS)** - It is defined as an operating system which implies deterministic timing behaviour.

→ Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time regardless the no. of services.

→ It implements policies and rules concerning time-critical allocation of a system's resources.

→ It decides which applications should run in which order and how much time needs to be allocated for each application.

Example - Windows CE, QNX, Vx Works, MicroC/OS-II, etc.

The Real Time Kernel -

→ The Kernel of a Real Time Operating System is refered as Real Time Kernel.

→ It is highly specialised and contains only the minimal set of ~~set~~ services for running the user applications/tasks.

Basic functions of a Real-Time Kernel - 7 Types

1) Task/Process management
2) Task/~~Schedule~~ Process Scheduling
3) Task/Process Synchronization
4) Error/Exception handling.
5) Memory management
6) Interrupt handling
7) Time management.

(1) Task / Process Management -

→ It deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion.

→ A Task Control Block (TCB) is used for holding the information corresponding to a task. It contains the following set of information.

Task ID: Task Identification Number

Task State: The current state of the task (e.g. State = 'Ready' for a task which is ready to execute).

Task Type: It indicates the type of the task. It can be a hard real time or soft real time or background task.

Task Priority: Task priority = 1 (for task with priority 1).

Task Context Pointer: Context Pointer. Pointer for context saving.

Task Memory Pointers: Pointers to the code memory, data memory and stack memory for the task.

Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc) used by the task.

Task Pointers: Pointers to other TCBs (TCBs for preceding, next and waiting tasks).

→ Task management service utilises the TCB of a task in the following way

* Creates a TCB for a task on creating a task.
* Delete/remove the TCB of a task when the task is terminated or deleted
* Reads the TCB to get the state of a task.
* Update the TCB with updated parameters on need basis
* Modify the TCB to change the priority of the task dynamically.

(2) Task/Process Scheduling - It deals with sharing the CPU among various tasks/processes. A Kernel application called 'Scheduler' handles the task scheduling.

→ Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behaviour.

(3) Task/Process Synchronisation - It deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

(4) Error/Exception handling - It deals with registering and handling the errors occured/exceptions raised during the execution of tasks.
→ Insufficient memory, timeouts, deadlocks, deadline missing, bus error divide by zero, unknown instruction execution, etc. are examples of errors/exceptions.
→ Errors/Exceptions can happen at the Kernel level services or at ~~task~~ task level.
→ Deadlock is an example for Kernel level exception, where as timeout is an example for a task level exception.

(5) Memory Management - The memory management function of an RTOS Kernel is slightly different. In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block.
→ The predictable timing and deterministic behaviour are the primary focus of an RTOS, where it is achieved by comprising the effectiveness of memory allocation.
→ RTOS Kernel uses blocks of fixed size of dynamic memory ~~allocation techniques~~ and the block is allocated for a task on a need basis. The blocks are stored in a 'Free Buffer Queue'.
→ Some RTOS Kernels allow memory protection as optional and the Kernel enters a 'fail-safe' mode when an illegal memory access occurs.

(6) Interrupt Handling - It deals with the handling of various types of interrupts. Interrupts provide Real-Time behaviour to systems.
→ Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
→ Interrupts can be either Synchronous or Asynchronous.
Interrupts which occurs in sync with the currently executing task is known as 'Synchronous Interrupts'. Software Interrupts like Divide by zero, memory segmentation error, etc are examples of Synchronous Interrupts.

Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.
→ For asynchronous interrupts, the interrupt handler is usually written as separate task and it runs in a different context.
→ It is generated by external devices connected to the processor/controller, timer over-flow interrupts, serial data reception/transmission interrupts etc are examples for asynchronous interrupts.

(7) Time Management – Accurate time management is essential for providing precise time reference for all applications.
→ The time reference to Kernel is provided by a high-resolution Real-Time Clock (RTC) hardware chip (hardware timer).
→ The hardware timer is programmed to interrupt the processor/controller at a fixed rate. This timer interrupt is referred as 'Timer tick'.
→ The 'Timer tick' is taken as the timing reference by the kernel.
→ The 'Timer tick' interval may vary depending on the hardware timer.

Hard Real Time – RTOS that strictly adhere to the timing constraints for a task is referred as 'Hard Real-Time' systems.
→ A Hard Real Time system must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard-Real-Time systems, including permanent data lose and irrecoverable damages to the system/users.
→ Hard Real-Time systems emphasise the principle 'A late answer is a wrong answer!'
Examples – Air Bag control systems and Anti-lock Brake systems (ABS).

Soft Real-Time – RTOS ~~does~~ that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as 'Soft Real-Time' systems.
→ Missing deadlines for tasks are acceptable for a Soft Real-Time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
→ A Soft Real-Time system emphasises the principle 'A late answer is an acceptable answer, but it could have done bit faster!'
Example – Automatic Teller Machine (ATM), Audio-video playback system.

Multiprocessing, Multitasking and Multiprogramming

→ In the operating systems context, multiprocessing describes the ability to execute multiple processes simultaneously.

→ Systems which are capable of performing multiprocessing, are known as multiprocessor systems.

→ Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.

→ The ability of the operating systems to have multiple programs in memory, which are ready for execution, is referred as multiprogramming.

→ The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as Multitasking.

→ Multitasking creates the illusion of multiple tasks executing in parallel. It involves the switching of CPU from executing one task to another.

→ In a multitasking environment, when task/process switch happens, the virtual processor (task/process) gets its properties converted into that of the physical processor.

→ The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel.

→ Multitasking involves 'Context switching', 'Context saving' and 'Context retrieval'.

→ The act of switching CPU among the processes or changing the current execution context is known as 'Context switching'.

→ The act of saving the current context which contains the context details for the currently running process at the time of CPU switching is known as 'Context saving'.

→ The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as 'Context retrieval'.

→ Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching.

Tasks, Process and Threads

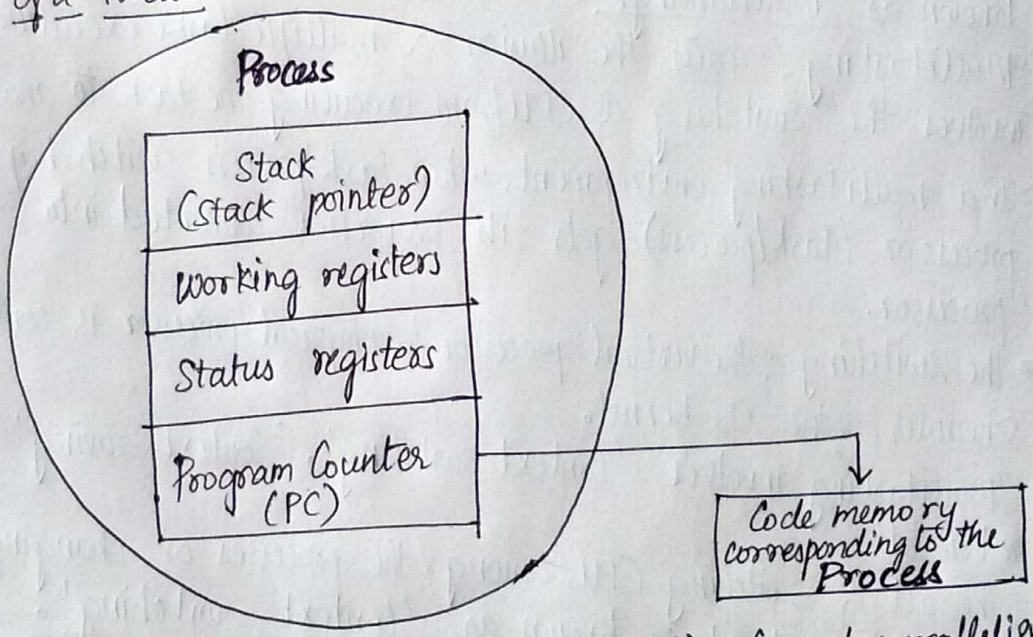
The term 'task' refers to something that needs to be done. In the operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program.

→ Task is also known as 'Job' in the operating system context.

→ A 'Process' is a program, or part of it, in execution. A process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously.

→ A process is sequential in execution while it requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.

The Structure of a Process



→ The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and the efficient utilisation of the CPU and other system resources.

→ Concurrent execution is achieved through the sharing of CPU among the processes.

→ A Process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process.

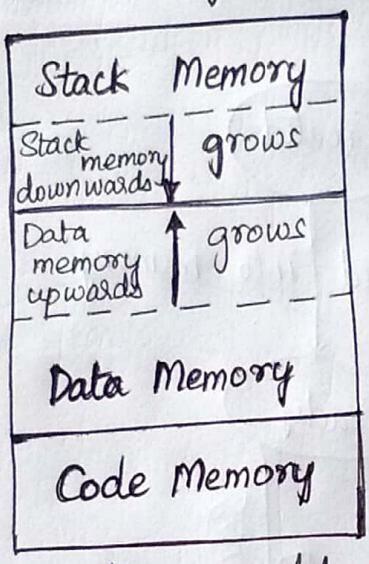
→ Also a stack for holding the local variables associated with the process and the code corresponding to the process.

Memory organisation of a Process

→ A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor.

→ When the processor gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU.

→ The memory occupied by the process is segregated into three regions, namely - Stack memory, Data memory and Code memory.



→ The 'Stack' memory holds all temporary data such as variables local to the process.

→ Data memory holds all global data for the process.

→ The Code memory contains the program code (instructions) corresponding to the process.

→ On loading a process into the main memory, a specific area of memory is allocated for the process.

→ The stack memory usually starts at the highest memory address from the memory area allocated for the process.

Example:- the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accomodate the variables local to the process.

Process States and State Transition

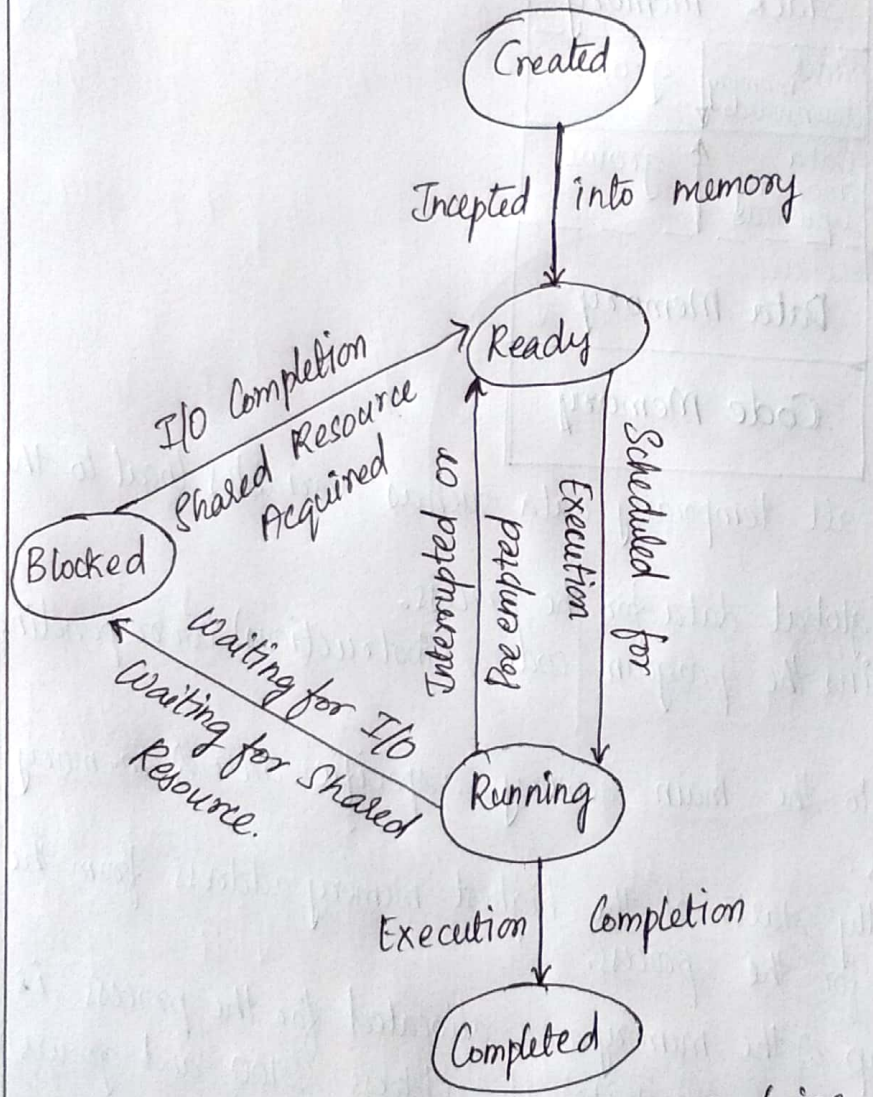
→ The creation of a process to its termination is not a single step operation.

→ The process traverses through a series of states during its transition from the newly created state to the terminated state.

→ The cycle through which a process changes its state from 'newly created' to 'execution completed' is known as 'Process Life Cycle'.

→ The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next.

Process states and state transition representation



→ The state at which a process is being created is referred as 'Create State'. The Operating System recognises a process in the 'Create State' but no resources are allocated to the process.

→ The state, where a process is incepted into the memory and awaiting the Processor time for execution is known as 'Ready state'.
At this stage, the process is placed in the 'Ready list' queue maintained by the OS.

→ The state where in the source code instructions corresponding to the process is being executed is called 'Running State'.
Running state is the state ~~where~~ at ~~running~~ which the process execution happens.

→ 'Blocked State / Wait State' refers to a state ~~where~~ where a running process is temporarily suspended from execution and does not have immediate accesses to resources.
The blocked state might be invoked by various conditions like:
(i) the process enters a wait state for an event to occur (eg. waiting for user inputs such as keyboard input) (or)
(ii) Waiting for getting access to a shared resource.

→ A State where the process completes the execution is known as 'Complete State'.

→ The transition of a process from one state to another is known as 'State Transition'.

When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

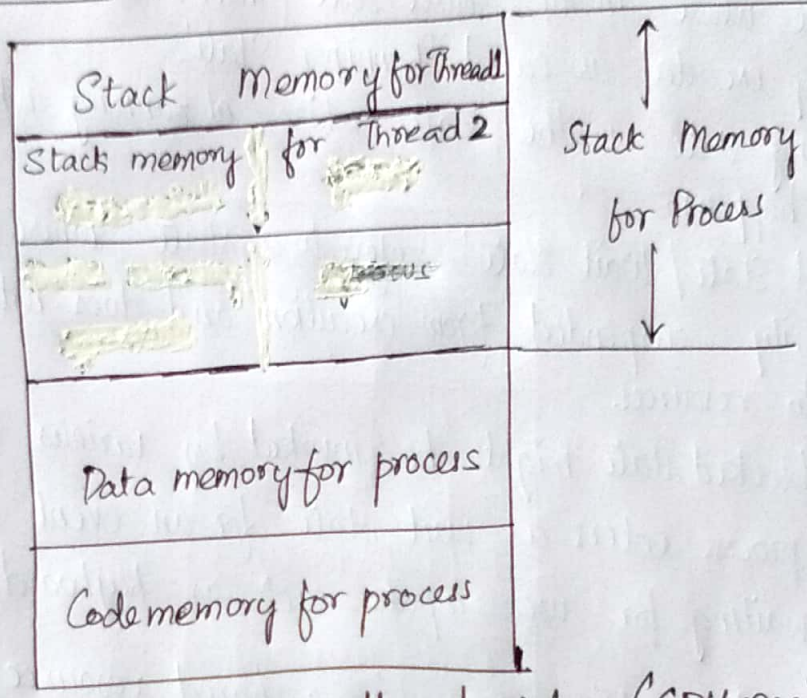
Threads

→ A Thread is the primitive that can execute code. A Thread is a single sequential flow of control within a process.

→ 'Thread' is also known as light weight process where as process can have many threads of execution.

→ Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area.

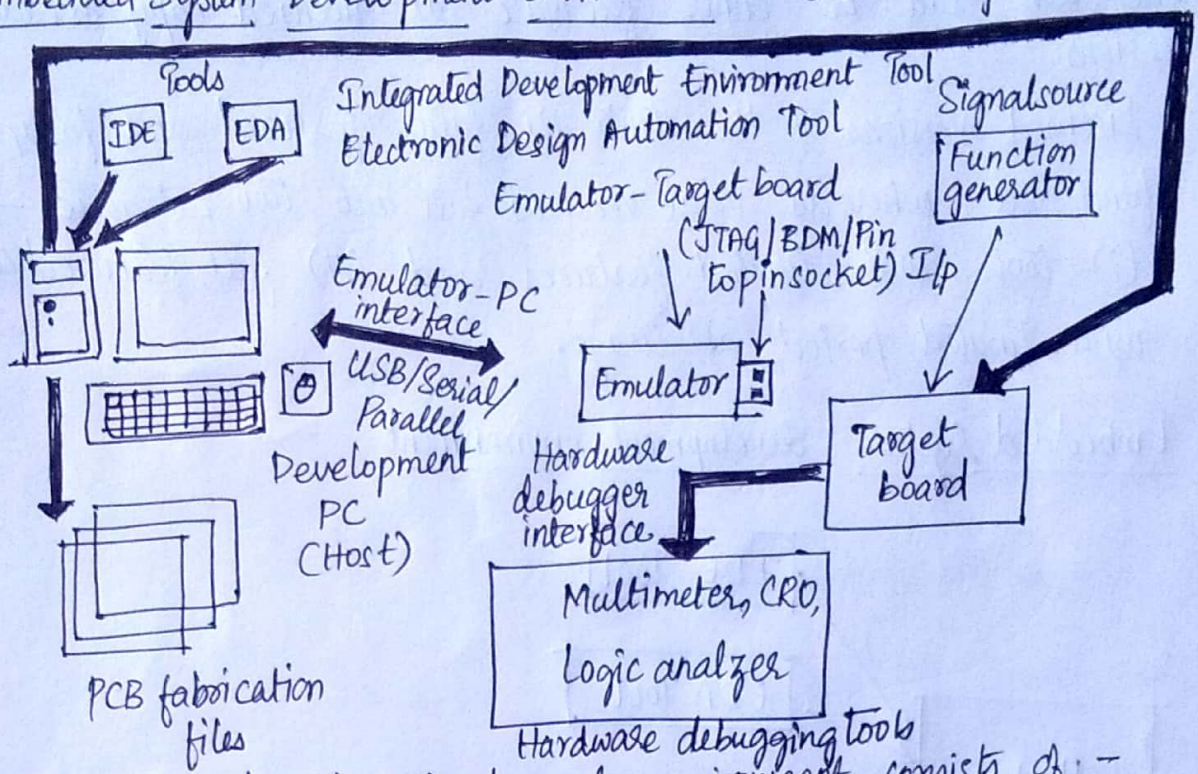
Memory Organisation of a Process and its associated Threads



→ Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack.

→ The memory model for a process and its associated threads are given in figure above.

Embedded System Development environment - Block diagram



The Embedded system Development environment consists of :-

(i) The Development PC (or) Host which acts as a heart of the development environment

(ii) Integrated Development Environment (IDE) Tool for embedded firmware development and debugging.

(iii) Electronic Design Automation (EDA) Tool for Embedded Hardware design.

(iv) An emulator hardware for debugging the target board.

(v) Signal sources (like function generator) for simulating the inputs to the target board.

(vi) Target hardware debugging tools (Digital CRO, Multimeter, logic analyzer, etc.) and the target hardware.

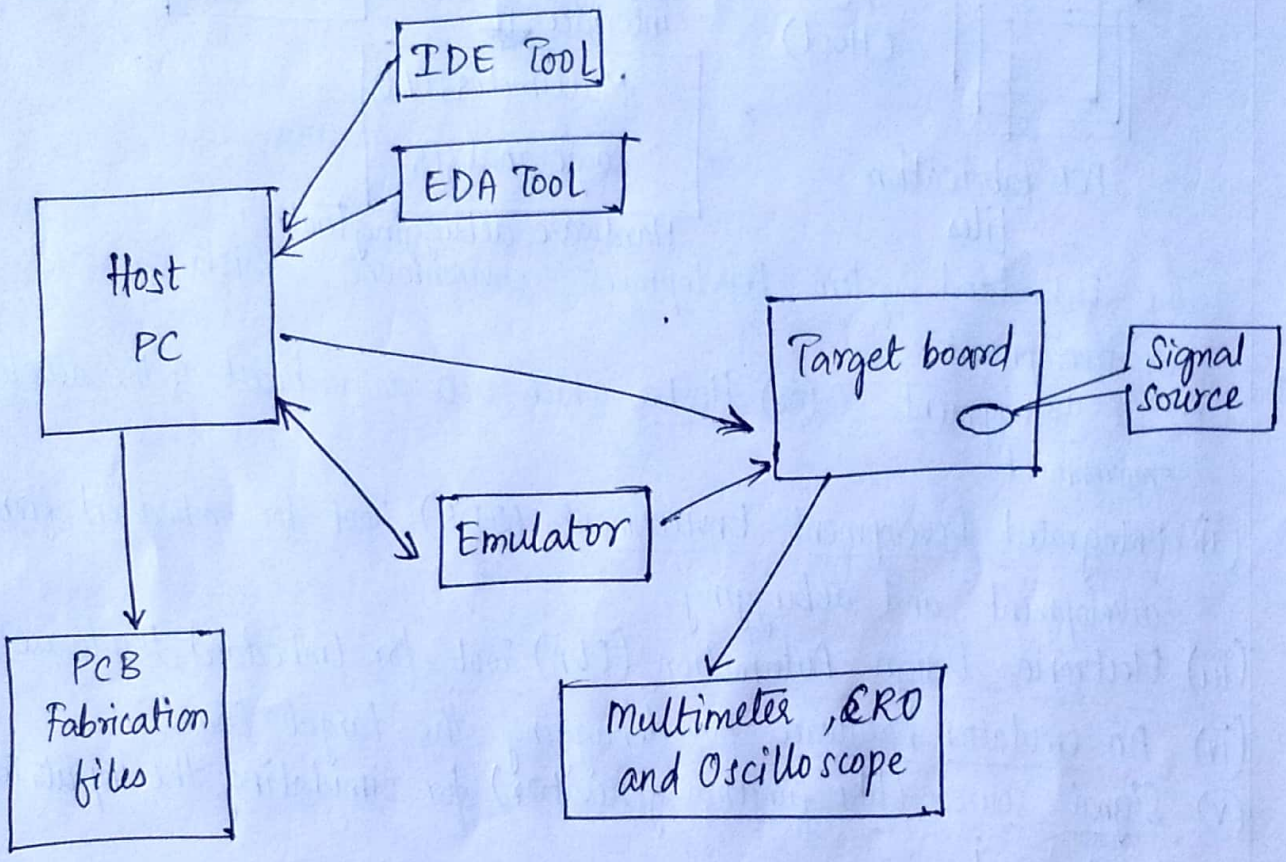
→ The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement. (and they are supplied as installable files in CDs by vendors.)

→ These tools need to be installed on the host PC used for development activities.

→ These tools can be either freeware or licensed copy or evaluation versions.

→ Licensed versions of the tools are fully featured and fully functional whereas trial versions fall into two categories — (i) tools with limited features, and (ii) full featured copies with limited period of usage.

Embedded System development environment



Out of Circuit Programming - It 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 programmer device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals.

→ The programmer contains a ZIF socket with locking pin to hold the device to be programmed.

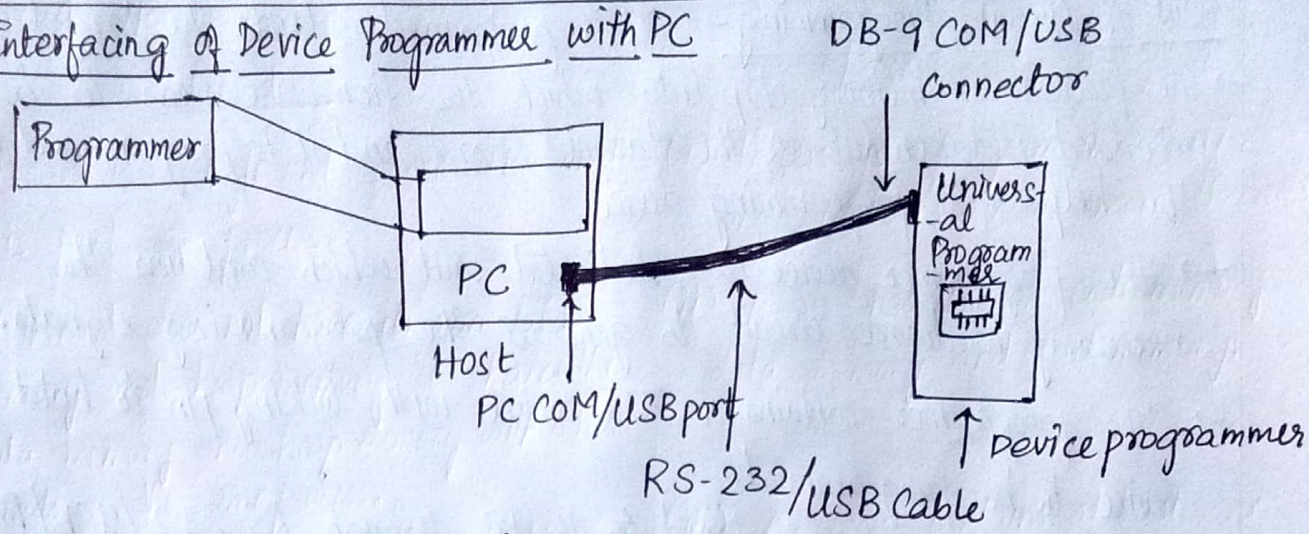
→ The programmer is interfaced to the PC through RS-232/USB/Parallel Port Interface and it is under control of a utility program running on a PC.

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

The sequence of operations for ~~embedded~~ 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 the PC and ensure proper connectivity is established between PC and programmer.
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 tile the completion of programming operation.
11. Ensure that programming is successful by checking the status LED on the programmer (or) by noticing the feedback from the utility program.
12. Unlock the ZIF socket and take the device out of programmer.

Interfacing of Device Programmer with PC



In System Programming (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.

→ It requires target board, PC, ISP cable and ISP utility.

→ Chips supporting ISP generates the necessary programming signals internally, using the chip's supply voltage.

→ The target board can be interfaced to the utility program running on PC through Serial Port/ Parallel Port/ USB.

→ 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)'.

→ A special ISP mode is used to perform ISP operations the target device and allows to communicate the device to communicate with an external host through a serial interface, such as a PC or terminal.

In Application Programming

→ It is a technique used by the firmware running on the target device for modifying a selected portion of the code memory.

→ It modifies the program code memory under the control of the embedded application. where updating calibration data, look up tables, etc which are stored in ~~cache~~ code memory. ~~under the control of the Core~~

→ The Boot ROM ~~resident~~ 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.

→ A common entry point to these API routines is provided for interfacing them to the end-user's application.

→ Functions are performed by setting up specific registers as required by a specific operation and performing a call to the common entry point.

→ For example, in case of subroutine call, after the completion of the function, control will return to the end-user's code.

→ 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.

Thread v/s Process

Thread

(i) Thread is a single unit of execution and is part of process.

(ii) A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process.

(iii) A thread cannot live independently; it lives within the process.

(iv) There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.

(v) Threads are very inexpensive to create

(vi) Context switching is inexpensive and Fast.

(vii) If a thread expires, its stack is reclaimed by the process.

(viii) It is a light weight process

(ix) It uses fewer resources

(x) Operating system is not required for thread switching.

Process

(i) Process is a program in execution and contains one or more threads.

(ii) Process has its own code memory, data memory and stack memory.

(iii) A process contains at least one thread.

(iv) Threads with in a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process).

(v) Processes are very expensive to create. Involves many OS overhead.

(vi) Context switching is complex and involves lot of OS overhead and is comparatively slower.

(vii) If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

(viii) It is a heavy weight process.

(ix) It uses more resources.

(x) Operating system interface is required for process switching.

Simulators, Emulators and Debuggers.

(1) **Simulator** is a software tool used for simulating the various conditions for checking the functionality of the application firmware.

→ IDE provides simulator support and helps in debugging the firmware for checking its required functionality.

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

→ The features of simulator based debugging are –

1. Purely software based
2. Doesn't require a real target system
3. Very primitive (Lack of featured I/O support. Everything is a simulated one).
4. Lack of Real-time behaviour.

→ It is based on debugging techniques are simple and straight forward.

→ For example, if the product under development is a handheld device, to test the functionalities of the various menu and user interfaces, a soft form model of the product with all UI (user interface) as given * in the end product can be developed in software.

→ Soft phone is an example for simulator.

Advantages – (i) No need for Original Target board
(ii) Simulate I/O Peripherals
(iii) Simulatis Abnormal conditions

Disadvantages – (i) Deviation from Real Behaviour
(ii) Lack of real timeliness.

Emulator is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.

→ It is a self contained hardware device which emulates the target CPU.

→ It contains necessary emulation logic and it is hooked to the debugging application running on the development PC and connects to the target board through some interface.

→ Emulators ~~is a~~ are special hardware devices used for emulating the functionality of a processor / controller and performing various debug operations like halt firmware execution, set breakpoints, get or set

→ Emulators can be classified into -

1) Software Emulator - Pure software applications which perform the functioning of a hardware emulator. It is also called as 'Emulators' for example, it is an application for emulating the operation of a PDA phone for application development. ~~is an example~~

2) Hardware Emulator - It is controlled by a debugger application running on the development PC. The debugger application may be part of the Integrated Development Environment (IDE) or a third party supplied tool.

Disadvantages - (i) Emulator is the accuracy of replication of target CPU functionalities.

(ii) It is easy to implement for simple target CPUs but for complex target CPUs it is quite difficult.

Debuggers

→ 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 and checking the signals from various buses of the embedded hardware.

→ Debugging process in embedded application is broadly classified into two namely – Hardware debugging and Firmware debugging.

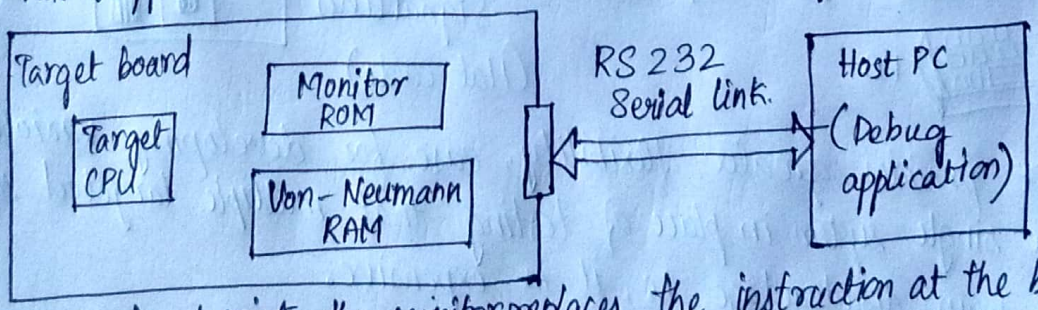
→ Hardware debugging deals with the monitoring of various bus signals and checking the status lines of the target hardware.

→ Firmware debugging deals with the examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design.

→ Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behaviour.

Simulation based Debugging (or) Monitor Program Based Firmware Debugging

→ The ROM monitor is a piece of software running on the target controller that can be seen as a rudimentary operating system, where it uses a numeric display and a hex keypad to allow the user interactive debugging



→ To implement breakpoints, the monitor replaces the instruction at the breakpoint address with a jump to the monitor code, which allows to check the contents of registers and variables.

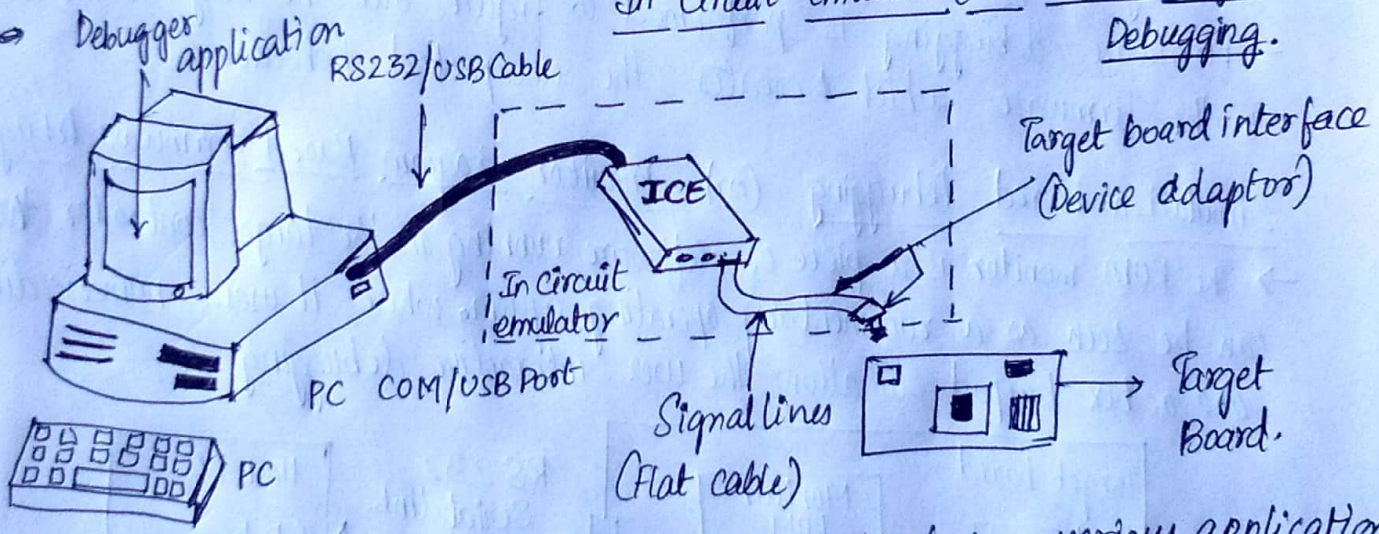
→ To resume program executing, the monitor simply restores the original instruction and transfers control back to the application.

→ Since such software breakpoints require that the program memory can be written by controller itself, which is not supported by all controller architectures,
→ Some microcontrollers also offer hardware breakpoints.

In Circuit Emulator (ICE) based target debugging techniques

→ In Circuit Emulator takes the place of the target processor. It contains a copy of target processor, plus RAM, ROM, and its own embedded software.
→ It allows to examine the state of the processor while the program is running.
→ It uses the remote debugger for human interface. It supports software and hardware breakpoints.
→ It has real time tracing. It stores the information about each processor cycle which is executed.
→ It allows to see in what order things happen.

In Circuit Emulator (ICE) Based Target Debugging.



→ ICE provides greater flexibility, ease for developing various applications on a single system in place of testing that multiple targeted systems.
→ The main disadvantage is, it is expensive.

Problems - Model Question Paper — Shortest Remaining Time (SRT)

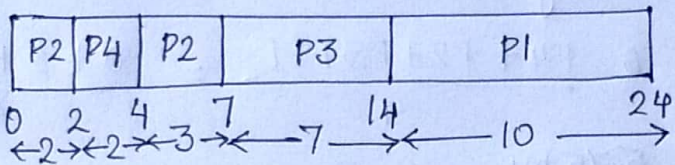
1) Describe preemptive SJF (Shortest Job First) scheduling. Determine average turn around time and average waiting time, if processes P1 P2 and P3 with estimated completion time of 10, 5, 7 milliseconds enter ready queue together and later P4 with a completion time of 2msec enters ready queue after 2 msec. (5 marks)

Note: Preemptive SJF (Shortest Job First) Scheduling

(i) At the beginning, there are only 3 processes (P1, P2, and P3) available in the 'Ready' queue and the SJF scheduler picks up the process with the shortest remaining time for execution completion for scheduling.

(ii) The execution sequence diagram is as shown below. where at the beginning it was P2, P3, P1.

(Assume



(iii) Now process P4 with estimated execution completion times 2ms enters the 'Ready' queue after 2ms of start of execution of P2.

(iv) Since SRT/SJF algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4.

(v) The remaining time for completion of P2 is 3ms which is greater than that of the remaining time for completion of the newly entered process P4 (2ms). Hence P2 is preempted and P4 is scheduled for execution.

(vi) P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution.

(vii) After 2ms of scheduling P4 terminates and now the scheduler again sorts the 'Ready' queue based on the remaining time for completion of the processes present in the 'Ready' queue.

(viii) Since the remaining time for P2 (3ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution.

(ix) Due to the arrival of the process P4 with execution time 2ms, the 'Ready' queue is re-sorted in the order P2, P4, P2, P3, P1.

Solution–

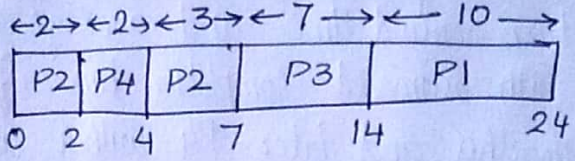
The waiting time for all the processes are given as

(i) Waiting time for P2 = 0ms + (4-2) ms = 2ms

(ii) Waiting time for P4 = 0ms

(iii) Waiting time for P3 = 7ms

(iv) Waiting time for P1 = 14ms



Where (i) P2 starts executing first and is interrupted by P4, waits for next CPU slot.

(ii) P4 starts executing by preemptive P2

(iii) P3 starts executing after completing P4 and P2

(iv) P1 starts executing after completing P4, P2 and P3.

Average waiting time = (Waiting time for all the processes) / No. of Processes

$$= \frac{P4 + P2 + P3 + P1}{4} = \frac{0 + 2 + 7 + 14}{4} = \frac{23}{4}$$

Average waiting time = 5.75 ms

Turn Around Time (TAT) for P2 = 7ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2ms → (Time spent in Ready Queue + Execution Time)
= (Execution start time – Arrival Time) + Estimated Execution Time)
= (2–2) + 2 = 2ms

Turn Around Time (TAT) for P3 = 14ms

Turn Around Time (TAT) for P1 = 24ms

Average Turn around Time = $\frac{\text{Turn around Time for all the processes}}{\text{No. of Processes}}$

$$= \frac{\text{Turn around Time for } (P2 + P4 + P3 + P1)}{4}$$

$$= \frac{(7 + 2 + 14 + 24)}{4} = \frac{47}{4} = 11.75 \text{ ms}$$

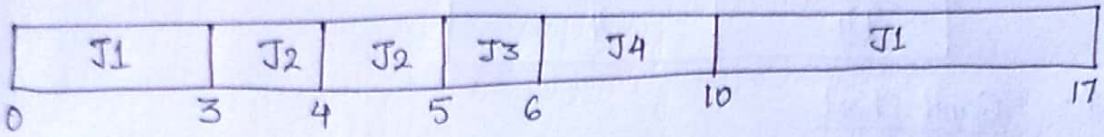
Average Turn around Time = 11.75 ms

Preemptive Shortest Job First / Shortest Remaining Time

2) For the following jobs calculate the turnaround time, waiting time using preemptive SJF scheduling algorithm.

Jobs	CPU burst time (ms)	Arrival Time (ms)
1	10	0.0
2	2	3.0
3	1	4.0
4	4	5.0

Solution -



Waiting Time - J1 = 0-0 + 10-3 = 7ms

J2 = 3-3 = 0ms

J3 = 5-4 = 1ms

J4 = 6-5 = 1ms

$$\text{Avg Waiting Time} = \frac{J1 + J2 + J3 + J4}{4} = \frac{7+0+1+1}{4} = \frac{9}{4}$$

Avg Waiting Time = 2.25 ms

Turn around Time (TAT) = Burst time + Waiting Time

J1 = 10 + 7 = 17ms

J2 = 2 + 0 = 2ms

J3 = 1 + 1 = 1ms

J4 = 4 + 1 = 5ms

$$\text{Average Turn around Time} = \frac{J1 + J2 + J3 + J4}{4} = \frac{17+2+1+5}{4} = \frac{26}{4}$$

Average Turn around Time = 6.5 ms

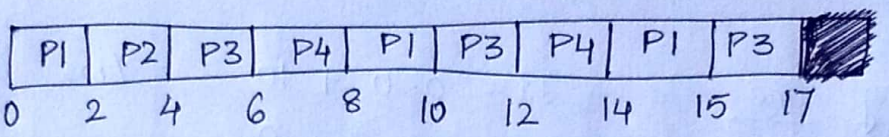
Round Robin Scheduling

3) Consider the following set of process that arrive at time 0ns, with the length of CPU burst given in nano seconds. Calculate the average waiting time and average turnaround time.

Provide the Gantt chart for the same (Timeslice = 2ns).

Process	Burst Time
P1	5
P2	2
P3	6
P4	4

Solution- Gantt Chart



Waiting Time

Process	Waiting Time
P_1	0-0+8-2+14-10 = 10ns
P_2	2-0 = 2ns
P_3	4-0+10-6+15-12 = 11ns
P_4	6-0+12-8 = 10ns

Average Waiting Time = $\frac{10+2+11+10}{4} = \frac{33}{4} = 8.25$ ns

Turnaround Time = waiting time + Burst time

Process	Turnaround Time
P_1	10+5 = 15ns
P_2	2+2 = 4ns
P_3	11+6 = 17ns
P_4	10+4 = 14ns

Average Turnaround Time = $\frac{15+4+17+14}{4} = \frac{50}{4} = 12.5$ ns

Priority Based Scheduling

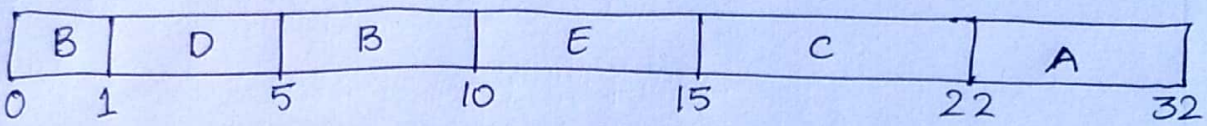
4) Assume the following processes arrive for execution at the time indicated and also mention with the length of the CPU-burst time given in milliseconds.

Job	Burst time (ns)	Priority	Arrival Time (ns)
A	10	5	0
B	6	2	0
C	7	4	1
D	4	1	1
E	5	3	2

Calculate the average waiting time and average turnaround time for preemptive priority scheduling algorithm.

Solution

Gantt Chart for Preemptive Priority



Waiting Time and Turn around Time

Jobs	Waiting Time	Turnaround Time
A	22 ns	32 ns
B	4 ns	10 ns
C	14 ns	7 ns
D	0 ns	4 ns
E	8 ns	13 ns

Average Waiting Time = $\frac{22+4+14+0+8}{5} = \frac{48}{5} = 9.6$ ns

Average turnaround Time = $\frac{32+10+21+4+13}{5} = \frac{80}{5} = 16$ ns