Atria Institute of Technology

Department of Information Science and Engineering

Bengaluru-560024

ACADEMIC YEAR: 2021-2022

EVEN SEMESTER NOTES

Semester : 4th Semester

Subject Name : Microcontroller and Embedded system

Subject Code : 18CS44

Faculty Name : Mrs. Kavitha S Patil

MODULE – 1

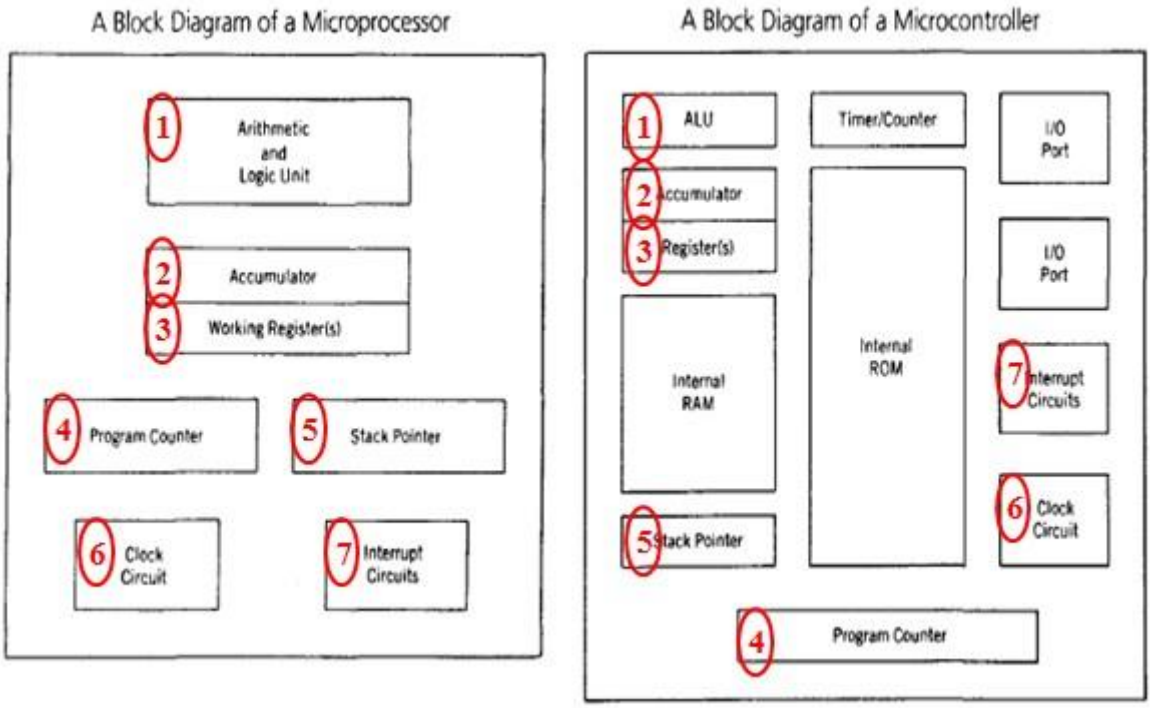
ARM EMBEDDED SYSTEMS & ARM PROCESSOR FUNDAMENTALS

MICROPROCESSORS versus MICROCONTROLLERS:

A *microprocessor* is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.

A *microcontroller* is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.

Microprocessors	Microcontrollers
Microprocessors generally does not have RAM, ROM and I/O pins.	Microcontroller is 'all in one' processor, with RAM, I/O ports, all on the chip.
Microprocessors usually use its pins as a bus to interface to RAM, ROM, and peripheral devices. Hence, the controlling bus is expandable at the board level.	Controlling bus is internal and not available to the board designer.
Microprocessors are generally capable of being built into bigger general purpose applications.	Microcontrollers are usually used for more dedicated applications.
Microprocessors, generally do not have power saving system.	Microcontrollers have power saving system, like idle mode or power saving; mode so overall it uses less power.
The overall cost of systems made with Microprocessors is high, because of the high number of external components required.	Microcontrollers are made by using complementary metal oxide semiconductor technology; so they are far cheaper than Microprocessors.
Processing speed of general microprocessors is above 1 GHz; so it works much faster than Microcontrollers.	Processing speed of Microcontrollers is about 8 MHz to 50 MHz.
Microprocessors are based on von-Neumann model; where, program and data are stored in same memory module.	Microcontrollers are based on Harvard architecture; where, program memory and data memory are separate.



ARM EMBEDDED SYSTEMS

The ARM processor core is a key component of many successful 32-bit embedded systems. ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

The first ARM1 prototype was designed in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001. The ARM Company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation.

For example, one of ARM's most successful cores is the ARM7TDMI. It provides up to 120 Dhrystone MIPS and is known for its high code density and low power consumption, making it ideal for mobile embedded devices.

THE RISC DESIGN PHYLOSOPHY:

- ✓ The ARM core uses *reduced instruction set computer (RISC)* architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed.
- ✓ The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler.

- In contrast, the traditional *complex instruction set computer (CISC)* relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. The following Figure illustrates these major differences.

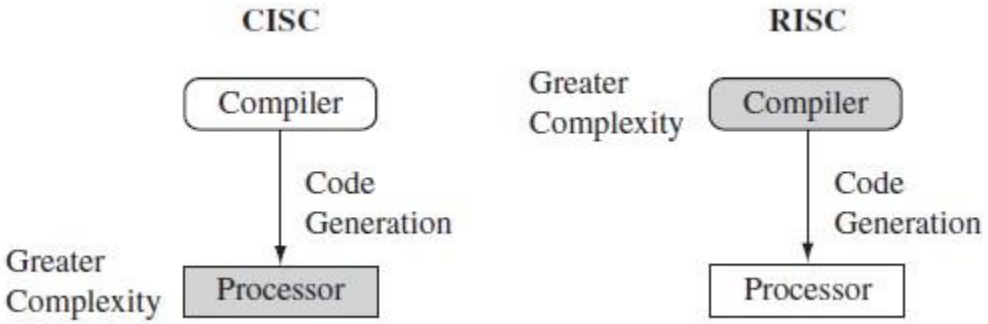


Fig: CISC vs. RISC

CISC	RISC
1. Complex instructions, taking multiple clock	1. Simple instructions, taking single clock
2. Emphasis on hardware, complexity is in the micro-program/processor	2. Emphasis on software, complexity is in the complier
3. Complex instructions, instructions executed by micro-program/processor	3. Reduced instructions, instructions executed by hardware
4. Variable format instructions, single register set and many instructions	4. Fixed format instructions, multiple register sets and few instructions
5. Many instructions and many addressing modes	5. Fixed instructions and few addressing modes
6. Conditional jump is usually based on status register bit	6. Conditional jump can be based on a bit anywhere in memory
7. Memory reference is embedded in many Instructions	7. Memory reference is embedded in LOAD/STORE instructions

The RISC philosophy is implemented with four major **design rules**:

1. *Instructions*—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is having fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
    - In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.

2. *Pipelines*—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.
    - There is no need for an instruction to be executed by a mini-program called microcode as on CISC processors.
3. *Registers*—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.
    - In contrast, CISC processors have dedicated registers for specific purposes.
4. *Load-store architecture*—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.
    - In contrast, with a CISC design the data processing operations can act on memory directly.

- These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.
    - In contrast, traditional CISC processors are more complex and operate at lower clock frequencies.

THE ARM DESIGN PHYLOSOPHY:

There are a number of physical features that have driven the ARM processor design.

- ✓ Portable embedded systems require ***battery power***. The ARM processor has been specially designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).
- ✓ ***High code density*** is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions—useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.
- ✓ Embedded systems are ***price sensitive***
    - Hence, *use slow and low-cost memory devices* to get substantial savings—essential for high-volume applications like digital cameras.
    - Also, *reduce the area of the die* taken up by the embedded processor; smaller the area used by the embedded processor, reduced cost of the design and manufacturing for the end product.

- ✓ ARM has incorporated ***hardware debug technology*** within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster.
- ✓ The ARM core is ***not a pure RISC architecture*** because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far.

Instruction Set for Embedded Systems:

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

- ✓ *Variable cycle execution for certain instructions*—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.
- ✓ *Inline barrel shifter leading to more complex instructions*—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.
- ✓ *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.
- ✓ *Conditional execution*—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
- ✓ *Enhanced instructions*—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations. These instructions allow a faster-performing ARM processor.

These ***additional features*** have made the ARM processor one of the most commonly used 32-bit embedded processor cores.

EMBEDDED SYSTEM HARDWARE:

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components.

The following Figure shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data.

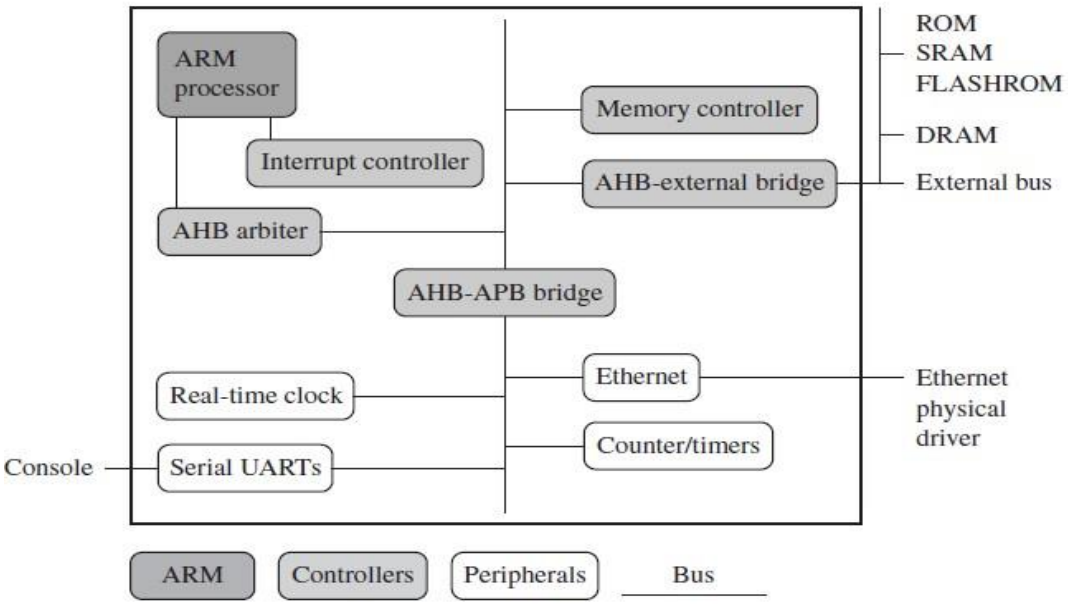


Figure: An ARM-based Embedded Device, a Microcontroller

We can separate the device into ***four main hardware components***:

1. The *ARM processor* controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components (memory and cache) that interface it with a bus.
2. *Controllers* coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
3. The *peripherals* provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
4. A *bus* is used to communicate between different parts of the device.

ARM Bus Technology:

Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

There are ***two different classes of devices*** attached to the bus:

1. The *ARM processor core* is a bus master—a logical device capable of initiating a data transfer with another device across the same bus.
2. *Peripherals* tend to be bus slaves—logical devices capable only of responding to a transfer request from a bus master device.

A bus has ***two architecture*** levels:

A *physical level*—covers the electrical characteristics and bus width (16, 32, or 64 bits).

The *protocol*—the logical rules that govern the communication between the processor and a peripheral.

AMBA Bus Protocol:

- ✓ The *Advanced Microcontroller Bus Architecture (AMBA)* was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
- ✓ The first AMBA buses introduced were the *ARM System Bus (ASB)* and the *ARM Peripheral Bus (APB)*. Later ARM introduced another bus design, called the *ARM High Performance Bus* (AHB).
- ✓ Using AMBA, peripheral designers can reuse the same design on multiple projects. A peripheral can simply be bolted onto the on-chip bus without having to redesign an interface for each different processor architecture. This plug-and-play interface for hardware developers improves availability and time to market.
- ✓ AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design. This change allows the AHB bus to run at higher clock speeds.
- ✓ ARM has introduced ***two variations*** on the AHB bus: *Multi-layer AHB* and *AHB-Lite*.
  - The Multi-layer AHB bus allows multiple active bus masters.
  - AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.
- ✓ The example device shown in the above Figure has three buses:
  - an *AHB bus* for the high- performance peripherals
  - an *APB bus* for the slower peripherals
  - a third *bus for external peripherals*, proprietary to this device.

Memory:

An embedded system has to have some form of memory to store and execute code. You have to compare price, performance, and power consumption when deciding upon specific memory characteristics, such as hierarchy, width, and type.

Hierarchy: All computer systems have memory arranged in some form of hierarchy. The following Figure shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away. Generally the closer memory is to the processor core, the more it costs and the smaller its capacity.

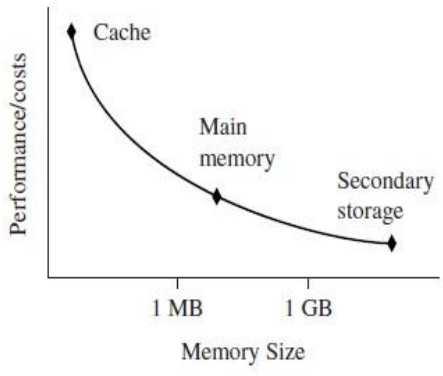


Figure: Memory Storage Trade-offs

- ✓ The *cache* is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. A cache provides an overall increase in performance but with a loss of predictable execution time. Although the cache increases the general performance of the system, it does not help real-time system response.
- ✓ The *main memory* is large—around 256 KB to 256 MB (or even greater), depending on the application—and is generally stored in separate chips. Load and store instructions access the main memory unless the values have been stored in the cache for fast access.
- ✓ *Secondary storage* is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage.

Width: The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits.

- ✓ The memory width has a direct effect on the overall performance and cost ratio. Lower bit memories are less expensive, but reduce the system performance.

The following Table summarizes theoretical cycle times on an ARM processor using different memory width devices.

Table: Fetching Instruction from Memory

Instruction Size	8-bit Memory	16-bit Memory	32-bit Memory
ARM 32-bit	4 cycles	2 cycles	1 cycles
Thumb 16-bit	2 cycles	1 cycles	1 cycles

Types: There are many *different types of memory*:

- ✓ *Read-only memory (ROM)* is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed.
  - ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
- ✓ *Flash ROM* can be written to as well as read, but it is slow to write so you shouldn't use it for holding dynamic data.
  - Its main use is for holding the device firmware or storing long-term data that needs to be preserved after power is off. The erasing and writing of flash ROM are completely software controlled with no additional hardware circuitry required, which reduces the manufacturing costs.
- ✓ *Dynamic random access memory (DRAM)* is the most commonly used RAM for devices. It has the lowest cost per megabyte compared with other types of RAM. DRAM is dynamic—it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.

- ✓ *Static random access memory (SRAM)* is faster than the more traditional DRAM, but requires more silicon area. SRAM is static—the RAM does not require refreshing. The access time for SRAM is considerably shorter than the equivalent DRAM because SRAM does not require a pause between data accesses. But cost of SRAM is high.
- ✓ *Synchronous dynamic random access memory (SDRAM)* is one of many subcategories of DRAM. It can run at much higher clock speeds than conventional memory. SDRAM synchronizes itself with the processor bus, because it is clocked. Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst.

Peripherals:

Embedded systems that interact with the outside world need some form of peripheral device. A ***peripheral device*** performs input and output functions for the chip by connecting to other devices or sensors that are off-chip.

- ✓ Each peripheral device usually performs a single function and may reside on-chip.
- ✓ Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.
- ✓ All ARM peripherals are *memory mapped*—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.
- ✓ *Controllers* are specialized peripherals that implement higher levels of functionality within an embedded system.
  - Two important types of controllers are memory controllers and interrupt controllers.

Memory Controllers: Memory controllers connect different types of memory to the processor bus.

- On power-up a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed.

Some memory devices must be set up by software; for example, when using DRAM, you first have to set up the memory timings and refresh rate before it can be accessed.

Interrupt Controllers: When a peripheral or device requires attention, it raises an *interrupt* to the processor. An *interrupt controller* provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are *two types of interrupt controller* available for the ARM processor: the standard interrupt controller and the vector interrupt controller.

1. The *standard interrupt controller* sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices.
    - The *interrupt handler* determines which device requires servicing by reading a device bitmap register in the interrupt controller.
2. The *vector interrupt controller (VIC)* is more powerful than the standard interrupt controller, because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.
    - Depending on the type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler.

EMBEDDED SYSTEM SOFTWARE:

An embedded system needs software to drive it. The following Figure shows four typical software components required to control an embedded device.

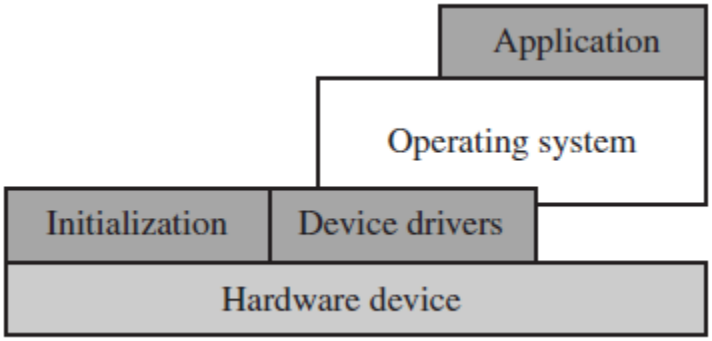


Figure: Software Abstraction Layers Executing on Hardware

- ✓ The *initialization code* is the first code executed on the board and is specific to a particular target or group of targets. It sets up the minimum parts of the board before handing control over to the operating system.
- ✓ The *operating system* provides an infrastructure to control applications and manage hardware system resources.
- ✓ The *device drivers* provide a consistent software interface to the peripherals on the hardware device.
- ✓ An *application* performs one of the tasks required for a device.
    - For example, a mobile phone might have a diary application.

There may be multiple applications running on the same device, controlled by the operating system.

Initialization (Boot) Code:

- ✓ Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices.
- ✓ The initialization code handles a number of administrative tasks prior to handing control over to an operating system image.
    - We can group these different tasks into *three phases*: initial hardware configuration, diagnostics, and booting.

1. ***Initial hardware configuration*** involves setting up the target platform, so that it can boot an image. The target platform comes up in a standard configuration; but, this configuration normally requires modification to satisfy the requirements of the booted image.
    - For example, the memory system normally requires reorganization of the memory map, as shown in the following Example.

Example: Initializing or organizing memory is an important part of the initialization code, because many operating systems expect a known memory layout before they can start.

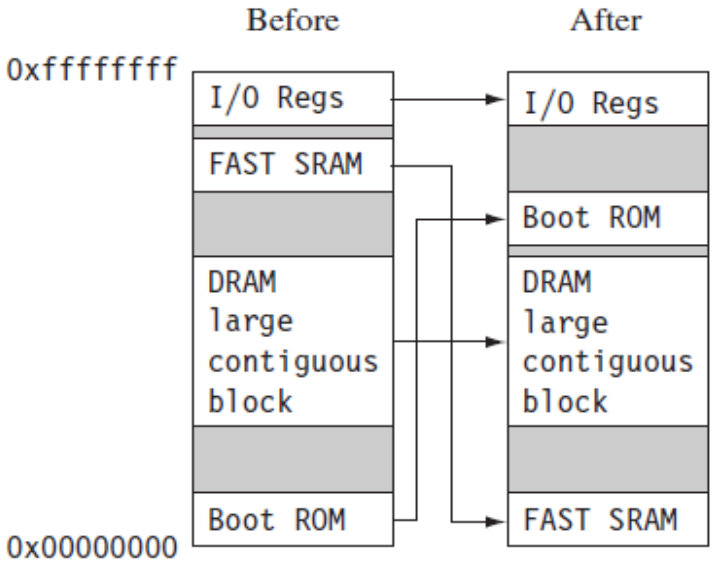


Figure: Memory Remapping

The above Figure shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up. The initialization code then redefines or remaps the memory map to place RAM at address 0x00000000—an important step because then the exception vector table can be in RAM and thus can be reprogrammed.

2. ***Diagnostic***s are often embedded in the initialization code. Diagnostic code tests the system by exercising the hardware target to check if the target is in working order. It also tracks down

standard system-related issues. The primary purpose of diagnostic code is fault identification and isolation.

3. ***Booting*** involves loading an image and handing control over to that image. The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system.
    - Booting an image is the final phase, but first you must load the image. Loading an image involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM. Once booted, the system hands over control by modifying the program counter to point into the start of the image.

Operating System:

- ✓ The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time.
- ✓ ARM processors support over 50 operating systems. We can divide operating systems into *two main categories*: real-time operating systems (RTOSs) and platform operating systems.

1. ***RTOSs*** provide guaranteed response times to events. Different operating systems have different amounts of control over the system response time.
    - A *hard real-time* application requires a guaranteed response to work at all.
    - In contrast, a *soft real-time* application requires a good response time, but the performance degrades more gracefully if the response time overruns.
2. ***Platform operating systems*** require a memory management unit to manage large, non-real-time applications and tend to have secondary storage.
    - The Linux operating system is a typical example of a platform operating system.

Applications:

- ✓ The operating system schedules *applications*—code dedicated to handle a particular task. An application implements a processing task; the operating system controls the environment.
    - An embedded system can have one active application or several applications running simultaneously.
- ✓ ARM processors are found in numerous market segments, including networking, auto-motive, mobile and consumer devices, mass storage, and imaging.
- ✓ ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communications.
- ✓ The mobile device segment is the largest application area for ARM processors, because of mobile phones.

- ✓ ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume.

- In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

ARM PROCESSOR FUNDAMENTALS

A programmer can think of an ARM core as functional units connected by data buses, as shown in the following Figure.

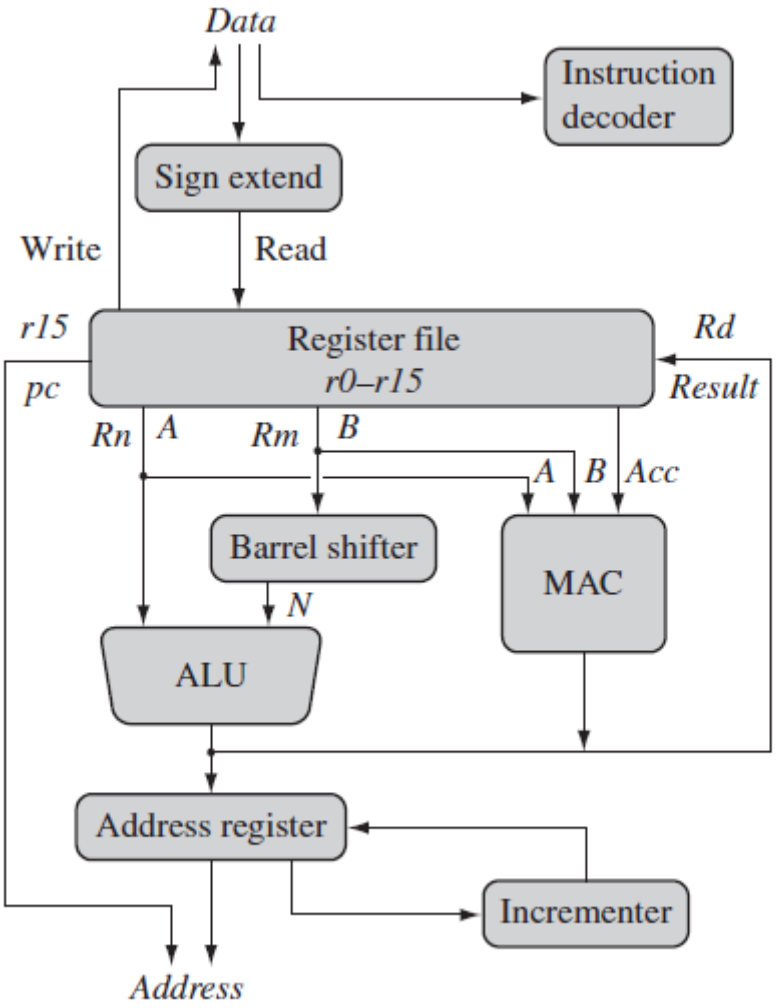


Figure: ARM Core dataflow Model

The arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.

- ✓ Data enters the processor core through the Data bus. The data may be an instruction to execute or a data item.

- Figure shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. (In contrast, Harvard implementations of the ARM use two different buses).

✓ The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

✓ The ARM processor, like all RISC processors, uses ***load-store architecture***—means it has two instruction types for transferring data in and out of the processor:

- load instructions copy data from memory to registers in the core
- store instructions copy data from registers to memory

✓ There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out in registers.

✓ Data items are placed in the register file—a storage bank made up of 32-bit registers.

- Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

✓ ARM instructions typically have two *source registers*, *Rn* and *Rm*, and a single result or *destination register*, *Rd*. Source operands are read from the register file using the internal buses A and B, respectively.

✓ The *ALU (arithmetic logic unit)* or *MAC (multiply-accumulate unit)* takes the register values *Rn* and *Rm* from the A and B buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file.

✓ Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.

- One important feature of the ARM is that register *Rm* alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

✓ After passing through the functional units, the result in *Rd* is written back to the register file using the *Result bus*.

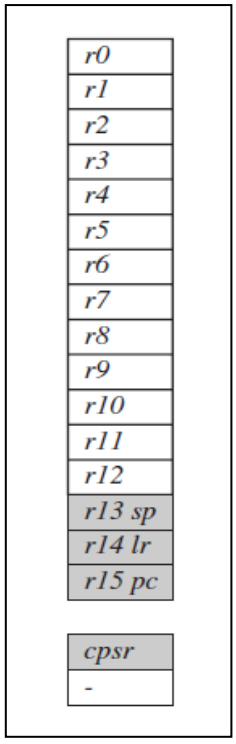
✓ For load and store instructions the *Incrementer* updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

✓ The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

REGISTERS:

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*.

The Figure shows the active registers available in user mode. (A protected mode is normally used when executing applications).



- ✓ The processor can operate in seven different modes.
- ✓ All the registers shown are 32 bits in size.
- ✓ There are up to 18 active registers:
  - 16 data registers and 2 processor status registers.
  - The data registers visible to the programmer are *r0* to *r15*.
- ✓ The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are given with different labels to differentiate them from the other registers.
  - *Register r13* is traditionally used as the ***stack pointer (sp)*** and stores the head of the stack in the current processor mode.
  - *Register r14* is called the ***link register (lr)*** and is where the core puts the return address whenever it calls a subroutine.
  - *Register r15* is the ***program counter (pc)*** and contains the address of the next instruction to be fetched by the processor.
- ✓ In ARM state the registers *r0* to *r13* are orthogonal—any instruction that you can apply to *r0* you can equally well apply to any of the other registers.
- ✓ In addition to the 16 data registers, there are two program status registers: ***cpsr (current program status register)*** and ***spsr (saved program status register)***.

CURRENT PROGRAM STATUS REGISTER:

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. The following Figure shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

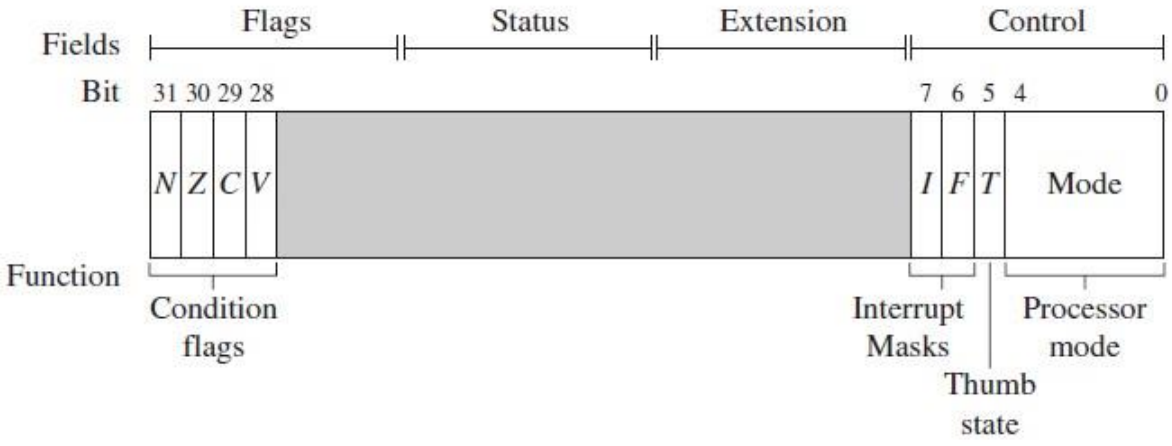


Figure: A Generic Program Status Register (psr)

The *cpsr* is divided into four fields, each 8 bits wide: *flags*, *status*, *extension*, and *control*. In current designs the extension and status fields are reserved for future use.

- ✓ The ***control field*** contains the *processor mode*, *state*, and *interrupt mask* bits.
- ✓ The ***flags field*** contains the *condition flags*.

Some ARM processor cores have extra bits allocated. For example, the *J bit*, which can be found in the flags field, is only available on *Jazelle-enabled processors*, which execute 8-bit instructions.

It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

Processor Modes:

- ✓ The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or non-privileged:
  - A *privileged mode* allows full read-write access to the *cpsr*.
  - A *non-privileged mode* only allows read access to the control field in the *cpsr*, but still allows read-write access to the condition flags.
- ✓ There are *seven processor modes* in total:
  - *six privileged modes* (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
    - The processor enters ***abort mode*** when there is a failed attempt to access memory.
    - ***Fast interrupt request*** and ***interrupt request modes*** correspond to the two interrupt levels available on the ARM processor.
    - ***Supervisor mode*** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
    - ***System mode*** is a special version of user mode that allows full read-write access to the *cpsr*.
    - ***Undefined mode*** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
  - *one non-privileged mode* (user).
    - ***User mode*** is used for programs and applications.

Banked Registers:

The following Figure shows all 37 registers in the register file.

- ✓ Of these, 20 registers are hidden from a program at different times.
- ✓ These registers are called *banked registers* and are identified by the shading in the diagram.

- ✓ They are available only when the processor is in a particular mode; for example, abort mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.
- ✓ Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.
- ✓ Every processor mode except user mode can change mode by writing directly to the mode bits of the *cpsr*.
- ✓ All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
- ✓ A banked register maps one-to-one onto a user mode register.
- ✓ If you change processor mode, a banked register from the new mode will replace an existing register.
    - For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The user mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

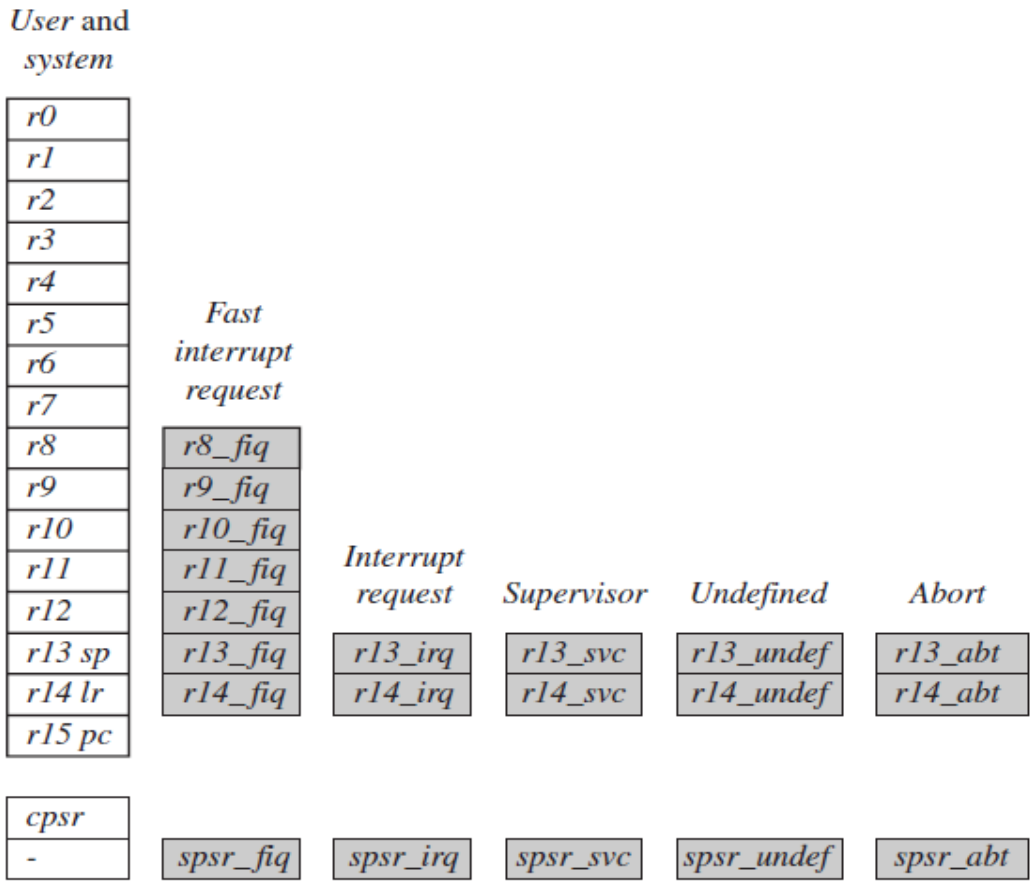


Figure: Complete ARM Register Set

- ✓ The processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.
- ✓ The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.
- ✓ Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.
- ✓ The following Figure illustrates what happens when an interrupt forces a mode change.

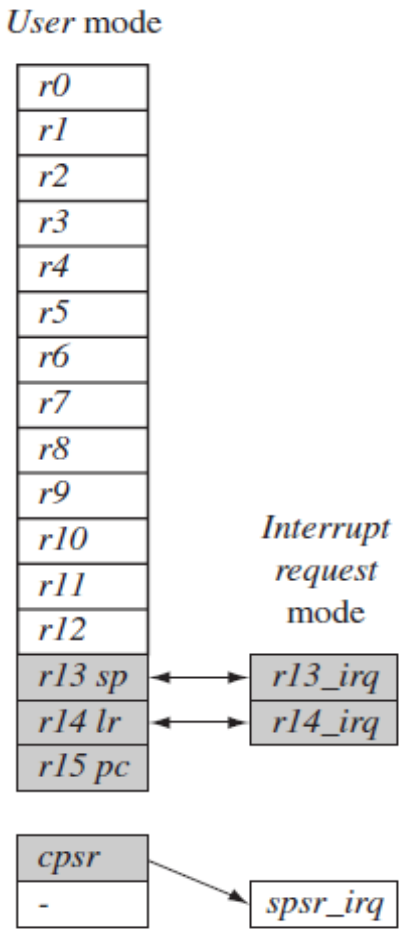


Figure: Changing Mode on an Exception

- ✓ The Figure shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core.
- ✓ This change causes user registers *r13* and *r14* to be banked. The user registers are replaced with registers *r13_irq* and *r14_irq*, respectively.
    - Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for interrupt request mode.

- ✓ The above Figure also shows a new register appearing in interrupt request mode: the *saved program status register (spsr)*, which stores the previous mode's *cpsr*. The *cpsr* being copied into *spsr_irq*.
- ✓ To return back to user mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the user registers *r13* and *r14*.
- ✓ Note that, the *spsr* can only be modified and read in a privileged mode. There is no *spsr* available in user mode.

- ✓ Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.
- ✓ When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.
- ✓ The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

Table: Processor Mode

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast Interrupt Request	fiq	yes	10001
Interrupt Request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

State and Instruction Sets:

- ✓ The state of the core determines which instruction set is being executed. There are three instruction sets:
  - ARM
  - Thumb
  - Jazelle.
- ✓ The ***ARM instruction set*** is only active when the processor is in ARM state.
- ✓ The ***Thumb instruction set*** is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions.

- ✓ You cannot inter-mingle sequential ARM, Thumb, and Jazelle instructions.
- ✓ The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor.
  - When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor.
  - When the *T* bit is 1, then the processor is in Thumb state.
- ✓ To change states the core executes a specialized branch instruction.

The following Table compares the ARM and Thumb instruction set features.

Table: ARM and Thumb Instruction Set Features

-	**ARM (*cspr T* = 0)**	**Thumb (*cspr T* = 1)**
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +pc	8 general-purpose registers +7 high registers +pc

- ✓ The ARM designers introduced a third instruction set called Jazelle. **Jazelle** executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.
- ✓ To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

The following Table gives the Jazelle instruction set features.

Table: Jazelle instruction set features

-	**Jezelle (*cspr T* = 0, *J* – 1)**
Instruction size	8-bit
Core Instructions	Over 60% of the Java byte-codes are implemented in hardware; the rest of the codes are implemented in software

Interrupt Masks:

- ✓ *Interrupt masks* are used to stop specific interrupt requests from interrupting the processor.
- ✓ There are two interrupt request levels available on the ARM processor core—
  - interrupt request (IRQ)
  - fast interrupt request (FIQ).

- ✓ The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively.
- ✓ The *I* bit masks IRQ when set to binary 1; and similarly, the *F* bit masks FIQ when set to binary 1.

Condition Flags:

- ✓ Condition flags are updated by comparisons and the result of ALU operations that specify the **S** instruction suffix.
  - o For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* Flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr*.
- ✓ With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.
- ✓ In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.
- ✓ Most ARM instructions can be executed conditionally on the value of the condition flags.

The following Table lists the condition flags and a short description on what causes them to be set.

Table: Condition Flags

Flag	Flag Name	Set When
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero
N	Negative	bit 31 of the result is a binary 1

These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution. The following Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle.

Figure: Example: *cspr* = nzCvqjiFt_SVC

- ✓ For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.
- ✓ In the *cpsr* example shown in above Figure, the *C* flag is the only condition flag set. The rest nzvq flags are all clear.
- ✓ The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled.
- ✓ Finally, you can see from the Figure, the processor is in *supervisor (SVC) mode*, since the mode[4:0] is equal to binary 10011.

Conditional Execution:

- ✓ Conditional execution controls whether or not the core will execute an instruction.
- ✓ Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*. If they match, then the instruction is executed; otherwise the instruction is ignored.
- ✓ The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction.
- ✓ The following Table lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Table: Condition Mnemonics

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

PIPELINE:

- ✓ A *pipeline* is the mechanism in a RISC processor, which is used to execute instructions.
- ✓ Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.

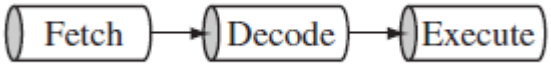


Figure: ARM7 Three-stage Pipeline

The above Figure shows a three-stage pipeline:

- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

The following Figure illustrates pipeline using a simple example.

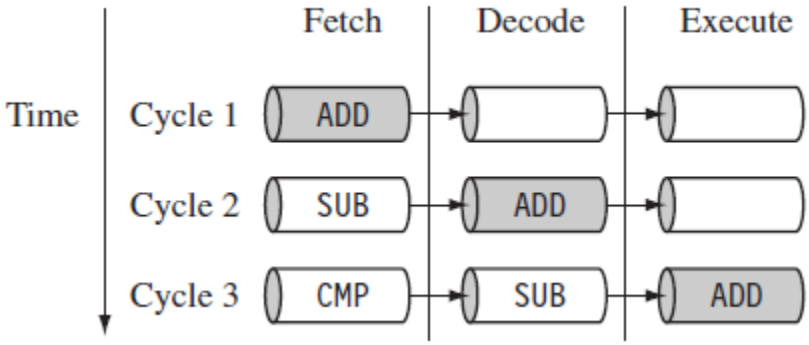


Figure: Pipelined Instruction Sequence

- ✓ The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor.
  - The three instructions are placed into the pipeline sequentially.
  - In the first cycle, the core fetches the ADD instruction from memory.
  - In the second cycle, the core fetches the SUB instruction and decodes the ADD instruction.
  - In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.
- ✓ This procedure is called ***filling the pipeline***.
- ✓ The pipeline allows the core to execute an instruction every cycle.
- As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn *increases the performance*.
- The increased pipeline length also means increased *system latency* and there can be *data dependency* between certain stages.

- The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure.

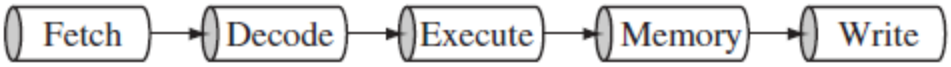


Figure: ARM9 Five-stage Pipeline

- The ARM9 adds a memory and writeback stage, which allows the ARM9 to –
  - process on average 1.1 Dhrystone MIPS per MHz
  - increase the instruction throughput in ARM9 by around 13% compared with an ARM7.
- The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in the following Figure.

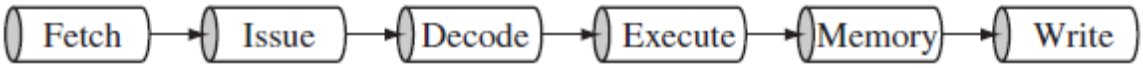


Figure: ARM10 Six-stage Pipeline

- The ARM10 –
  - can process on average 1.3 Dhrystone MIPS per MHz
  - have about 34% more throughput than an ARM7 processor core
  - but again at a higher latency cost.

NOTE: Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7. Hence, code written for the ARM7 will execute on an ARM9 or ARM10.

Pipeline Executing Characteristics:

- ✓ The ARM pipeline will not process an instruction, until it passes completely through the execute stage.
  - For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

The following Figure shows an instruction sequence on an ARM7 pipeline.

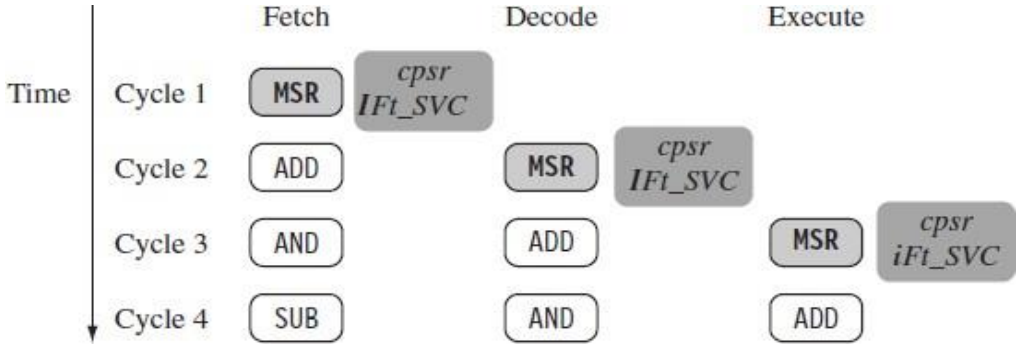


Figure: ARM Instruction Sequence

- ✓ The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts.
- ✓ Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

The following Figure illustrates the use of the pipeline and the program counter *pc*.

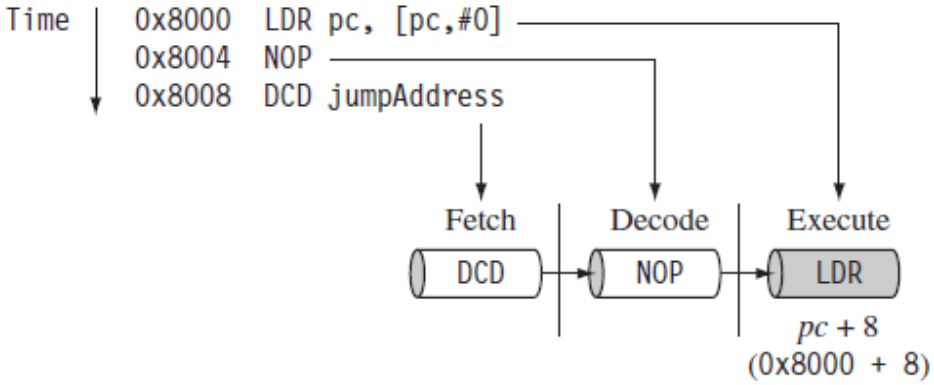


Figure: Example: *pc* = address + 8

- ✓ In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.
- ✓ Note when the processor is in Thumb state the *pc* is the instruction address plus 4.

- ✓ There are three other characteristics of the pipeline.
  - First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.
  - Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
  - Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline.

EXCEPTIONS, INTERRUPTS AND THE VECTOR TABLE:

- ✓ When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*.
  - The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
  - The memory map address 0x00000000 (or in some processors starting at the offset 0xffff0000) is reserved for the vector table, a set of 32-bit words.

- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see the following Table).

Table: The Vector Table

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0x00000000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	SABT	0x00000010	0xffff0010
Reserved	–	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
  - ***Reset*** vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
  - ***Undefined*** instruction vector is used when the processor cannot decode an instruction.
  - ***Software interrupt*** vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
  - ***Prefetch abort*** vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
  - ***Data abort*** vector is similar to a prefetch abort, but is raised when an instruction attempts to access data memory without the correct access permissions.
  - ***Interrupt request*** vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
  - ***Fast interrupt request*** vector is similar to the interrupt request, but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

CORE EXTENSIONS:

- *Core extensions* are the standard hardware components placed next to the ARM core.
- They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications.

Each ARM family has different extensions available. There are *three hardware extensions*: cache and tightly coupled memory, memory management, and the coprocessor interface.

Cache and Tightly Coupled Memory:

- ✓ The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- ✓ Most ARM-based embedded systems use a single-level cache internal to the processor.

- ✓ ARM has *two forms of cache*. The first is found attached to the Von Neumann–style cores. It combines both data and instruction into a single unified cache, as shown in the following Figure.

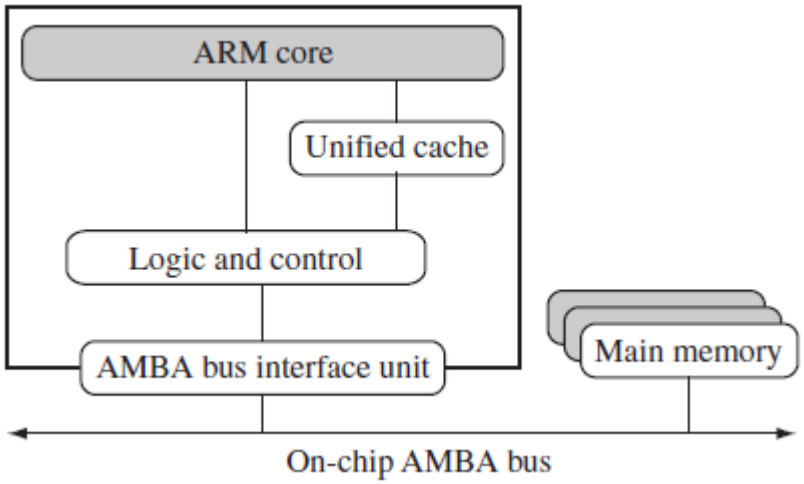


Figure: Von Neumann Architecture with Cache

- ✓ The second form, attached to the Harvard-style cores, has separate caches for data and instruction, as shown in the following Figure.

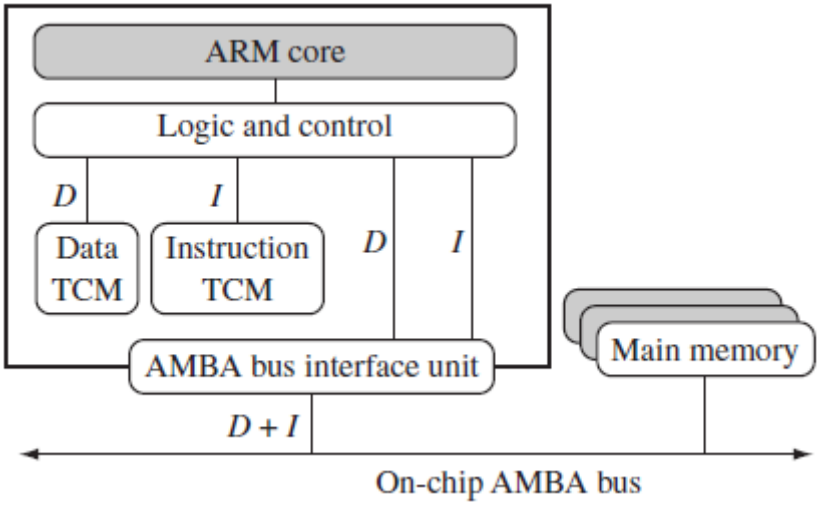


Figure: Harvard Architecture with TCMs

- ✓ A cache provides an overall increase in performance, but at the expense of predictable execution. But the real-time systems require the code execution to be deterministic— the time taken for loading and storing instructions or data must be predictable.
- ✓ This is achieved using a form of memory called *tightly coupled memory (TCM)*. TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.
- ✓ TCMs appear as memory in the address map and can be accessed as fast memory.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. The following Figure shows an example core with a combination of caches and TCMs.

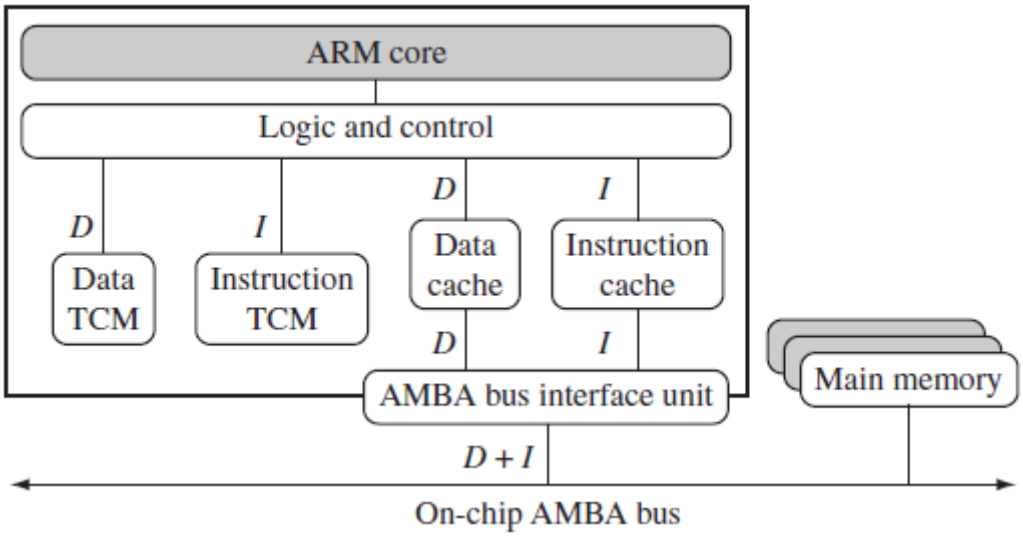


Figure: Harvard Architecture with Caches and TCMs

Memory Management:

- ✓ Embedded systems often use multiple memory devices. It is usually necessary to have a method to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.
- ✓ ARM cores have *three different types of memory management hardware*—
  - o no extensions providing no protection
  - o a memory protection unit (MPU) providing limited protection
  - o a memory management unit (MMU) providing full protection
- ✓ ***Non protected memory*** is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

- ***MPU***s employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map.
- ***MMU***s are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking.

Coprocessors:

- Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.
- The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
  - For example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- The coprocessor can also extend the instruction set by providing a specialized group of new instructions.
  - For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.
- These new instructions are processed in the decode stage of the ARM pipeline.
  - If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
  - If the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

****

****

MODULE – 2

ARM INSTRUCTION SET & ARM PROGRAMMING

INTRODUCTION TO THE ARM INSTRUCTION SET

Different ARM architecture revisions support different instructions. However, new revisions usually add instructions and remain backwardly compatible. Code you write for architecture ARMv4T should execute on an *ARMv5TE* processor.

The following Table provides a complete list of ARM instructions available in the *ARMv5E instruction set architecture* (*ISA*). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set.

Table: ARM Instruction Set

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory

Mnemonics	ARM ISA	Description
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register

Mnemonics	ARM ISA	Description
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ($(16 \times 16) + 32 = 32$-bit)
SMLAL	v3M	signed multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ($(16 \times 16) + 64 = 64$-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction ($((32 \times 16) \gg 16) + 32 = 32$-bit)
SMULL	v3M	signed multiply long ($32 \times 32 = 64$-bit)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions ($16 \times 16 = 32$-bit)
SMULW*y*	v5E	signed multiply instruction ($(32 \times 16) \gg 16 = 32$-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
UMULL	v3M	unsigned multiply long ($32 \times 32 = 64$-bit)

In the following sections, the hexadecimal numbers are represented with the prefix *0x* and binary numbers with the prefix *0b*. The examples follow this format:

PRE *<pre-conditions>*

<instruction/s>

POST *<post-conditions>*

In the pre- and post-conditions, memory is denoted as

mem<data_size>[address]

This refers to *data_size* bits of memory starting at the given byte address. For example, *mem32[1024]* is the 32-bit value starting at address 1 KB.

ARM instructions process data held in registers and memory is accessed only with load and store instructions.

ARM instructions commonly take two or three operands. For instance, the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

ARM instructions classified as—data processing instructions, branch instructions, load-store instructions, software interrupt instruction, and program status register instructions.

DATA PROCESSING INSTRUCTIONS:

The data processing instructions manipulate data within registers. They are—

- ✓ move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.

Most data processing instructions can process one of their operands using the barrel shifter.

If you use the S suffix on a data processing instruction, then it updates the flags in the *cpsr*.

Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.

- The *C* flag is set from the result of the barrel shift as the last bit shifted out.
- The *N* flag is set to bit 31 of the result.
- The *Z* flag is set if the result is zero.

MOVe Instructions:

Move instruction copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

Example: This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value *5*, and overwriting the value *8* in register *r7*.

```
PRE   r5 = 5
      r7 = 8
MOV   r7, r5   ; let r7 = r5
POST  r5 = 5
      r7 = 5
```

Barrel Shifter:

In above Example, we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

- ✓ Data processing instructions are processed within the arithmetic logic unit (ALU).
- ✓ A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- ✓ Pre-processing or shift occurs within the cycle time of the instruction.
  - This shift increases the power and flexibility of many data processing operations.
  - This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- ✓ There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

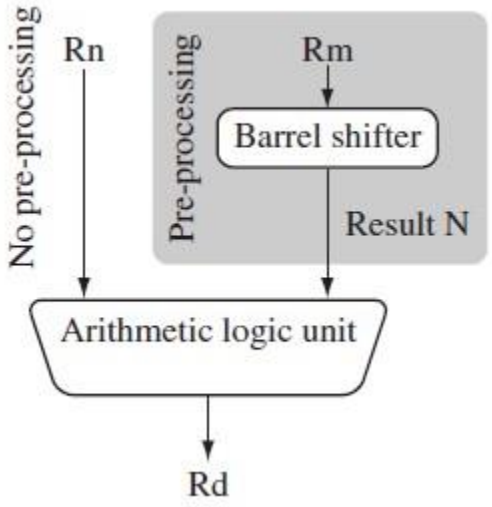


Figure: Barrel Shifter and ALU

- ✓ Figure shows the data flow between the ALU and the barrel shifter.
- ✓ Register *Rn* enters the ALU without any pre- processing of registers.
- ✓ We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard *C* language shift operator « to the register.

- ✓ The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in the following Table.

Table: Barrel Shifter Operations

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

- ✓ The five different shift operations that you can use within the barrel shifter are summarized in the above Table.

```
PRE   r5 = 5
      r7 = 8
MOV   r7, r5, LSL #2   ; let r7 = r5*4 = (r5 << 2)
POST  r5 = 5
      r7 = 20
```

- ✓ The above example multiplies register *r5* by four and then places the result into register *r7*.
- ✓ The following Figure illustrates a logical shift left by one.

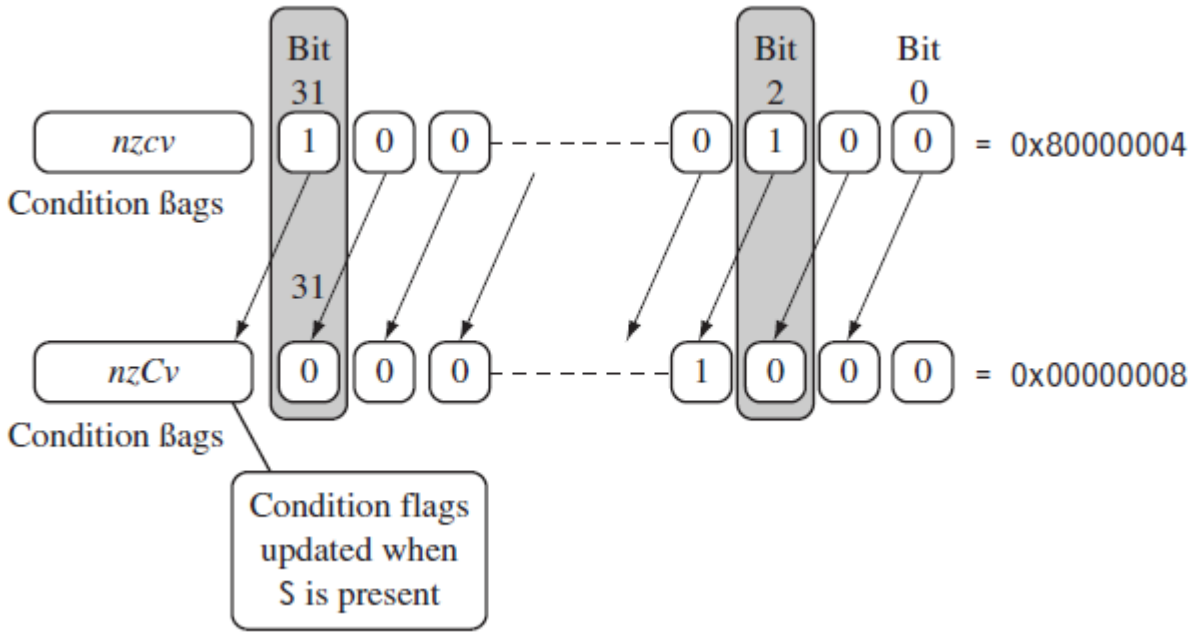


Figure: Logical Shift Left by One

- ✓ For example, the contents of bit *0* are shifted to bit *1*. Bit *0* is cleared. The *C* flag is updated with the last bit shifted out of the register. This is bit (*32 - y*) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Example: This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the C flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

PRE *cpsr = nzcvqiFt_USER*
r0 = 0x00000000
r1 = 0x80000004

MOVS r0, r1, LSL #1

POST *cpsr = nzCvqiFt_USER*
r0 = 0x00000008
r1 = 0x80000004

The following Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

Table: Barrel Shifter Operation Syntax for data Processing Instructions

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation.

Example: The following simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

PRE *r0 = 0x00000000*

r1 = 0x00000002

r2 = 0x00000001

SUB r0, r1, r2

POST *r0 = 0x00000001*

Example: The following reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

PRE *r0 = 0x00000000*

r1 = 0x00000077

RSB r0, r1, #0 ; Rd = 0x0 - r1

POST *r0 = -r1 = 0xffffff89*

Example: The SUBS instruction is useful for decrementing loop counters. In this example, we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the ZC flags being set.

PRE *cpsr = nzcvqiFt_USER*

r1 = 0x00000001

SUBS r1, r1, #1

POST *cpsr = nZCvqiFt_USER*

r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions:

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. The following Example illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register *r1* by three.

Example: Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

PRE *r0 = 0x00000000*

r1 = 0x00000005

ADD r0, r1, r1, LSL #1

POST *r0 = 0x0000000f*

r1 = 0x00000005

Logical Instructions:

Logical instructions perform bitwise logical operations on the two source registers.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example: This example shows a logical OR operation between registers *r1* and *r2*. Register *r0* holds the result.

PRE *r0 = 0x00000000*

r1 = 0x02040608

r2 = 0x10305070

ORR r0, r1, r2

POST *r0 = 0x12345678*

Example: This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

PRE *r1 = 0b1111*

r2 = 0b0101

BIC r0, r1, r2

POST *r0 = 0b1010*

This is equivalent to – *Rd = Rn **AND NOT** (N)*

In this example, register *r2* contains a binary pattern where every binary *1* in *r2* clears a corresponding bit location in register *r1*.

This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

NOTE: The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

Comparison Instructions:

- ✓ The comparison instructions are used to compare or test a register with a 32-bit value.
- ✓ They update the *cpsr* flag bits according to the result, but do not affect other registers.

- ✓ After the bits have been set, the information can then be used to change program flow by using conditional execution.
- ✓ It is not required to apply the S suffix for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

N is the result of the shifter operation.

Example: This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *Z* flag prior to execution is *0* and is represented by a lowercase *z*. After execution the *Z* flag changes to *1* or an uppercase *Z*. This change indicates equality.

PRE *cpsr = nzcvqiFt_USER*
r0 = 4
r9 = 4
CMP r0, r9
POST *cpsr = nZcvqiFt_USER*

- ✓ The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- ✓ For each, the results are discarded but the condition bits are updated in the *cpsr*.
- ✓ It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

Multiply Instructions:

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.

The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^* Rs) + Rn$
MUL	multiply	$Rd = Rm^* Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

Example: This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value *2*, and *r2* is equal to *2*. The result, *4*, is then placed into register *r0*.

PRE *r0 = 0x00000000*

r1 = 0x00000002

r2 = 0x00000002

*MUL r0, r1, r2 ; r0 = r1*r2*

POST *r0 = 0x00000004*

r1 = 0x00000002

r2 = 0x00000002

Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *RdLo* and *RdHi*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result. The following shows an example of a long unsigned multiply instruction.

Example: The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

PRE r0 = 0x00000000

r1 = 0x00000000

r2 = 0xf0000002

r3 = 0x00000002

UMULL r0, r1, r2, r3 ; [r1,r0] = r2*r3

POST r0 = 0xe0000004 ; = RdLo

r1 = 0x00000001 ; = RdHi

BRANCH INSTRUCTIONS:

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops.

The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- ✓ The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- ✓ *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example: This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B       forward
        ADD r1, r2, #4
        ADD r0, r6, #2
        ADD r3, r7, #4
forward
        SUB r1, r2, #4
-------------------------------------
backward
        ADD r1, r2, #4
        SUB r1, r2, #4
        ADD   r4, r6, r7
        B       backward
```

In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

- ✓ The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call.

Example: This example shows a simple fragment of code that, branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

```
    BL      subroutine      ; branch to subroutine
    CMP r1, #5              ; compare r1 with 5
    MOVEQ r1, #0            ; if (r1==5) then r1 = 0
    :
subroutine
    <subroutine code>
    MOV pc, lr              ; return by moving  pc = lr
```

- ✓ The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction.
- ✓ The BX instruction uses an absolute address stored in register *Rm*. It is primarily used to branch to and from Thumb code. The *T* bit in the *cpsr* is updated by the least significant bit of the branch register.
- ✓ Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.

LOAD-STORE INSTRUCTIONS:

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer:

- ✓ These instructions are used for moving a single data item in and out of a register.
- ✓ The data types supported are signed and unsigned words (32-bit), half-words (16-bit), and bytes.

Here are the various load-store single-register transfer instructions.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

- ✓ LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.
  - For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on.

Example: This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address  pointed  to by
; register r1.
;
        STR r0, [r1]            ; = STR r0, [r1, #0]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex.

Table: Index Methods

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

- ✓ *Preindex with writeback* calculates an address from a base register plus address offset and then updates that address base register with the new address.
- ✓ *Preindex* offset is the same as the preindex with writeback but does not update the address base register.
  - The preindex mode is useful for accessing an element in a data structure.
- ✓ *Postindex* only updates the address base register after the address is used.
  - The postindex and preindex with writeback modes are useful for traversing an array.

Example:

PRE *r0 = 0x00000000*

r1 = 0x00090000

mem32[0x00009000] = 0x01010101

mem32[0x00009004] = 0x02020202

LDR r0, [r1, #4]!

Preindexing with writeback:

POST(1) *r0 = 0x02020202*

r1 = 0x00009004

LDR r0, [r1, #4]

Preindexing:

POST(2) *r0 = 0x02020202*

r1 = 0x00009000

LDR r0, [r1], #4

Postindexing:

POST(3) *r0 = 0x01010101*

r1 = 0x00009004

- ✓ The above Example used a preindex method. This example shows how each indexing method affects the address held in register *r1*, as well as the data loaded into register *r0*.

The addressing modes available with a particular load or store instruction depend on the instruction class. The following Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Table: Single-Register Load-Store Addressing, Word or Unsigned Byte

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

- ✓ A signed offset or register is denoted by "+/-", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- ✓ Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- ✓ Register means the address is calculated using the base address register and a specific register's contents.
- ✓ Scaled means the address is calculated using the base address register and a barrel shift operation.

The following Table provides an example of the different variations of the LDR instruction.

Table: Examples of LDR Instructions using Different Addressing Modes

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

The following Table shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data.

Table: Single-Register Load-Store Addressing, Halfword, Signed Halfword, Signed Byte and Doubleword

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.

The following Table shows the variations for STRH instructions.

Table: Variations of STRH Instructions

	Instruction	Result	*r1* +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Multiple-Register Transfer:

- ✓ Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- ✓ The transfer occurs from a base address register *Rn* pointing into memory.
  - Multiple-register transfer instructions are more efficient from single-register transfers for
    - moving blocks of data around memory and
    - saving and restoring context and stacks.
- ✓ Load-store multiple instructions can increase interrupt latency.
- ✓ ARM implementations do not usually interrupt instructions while they are executing.
  - For example, on an ARM7 a load multiple instruction takes $2 + Nt$ cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory.
- ✓ If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

- ✓ Compilers, such as *armcc*, provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

LDM	load multiple registers	{Rd}*N <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}*N -> mem32[start address + 4*N] optional Rn updated

The following Table shows the different addressing modes for the load-store multiple instructions. Here *N* is the number of registers in the list of registers.

Table: Addressing Mode for Load-Store Multiple Instructions

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
IB	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
DA	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
DB	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

- ✓ Any subset of the current bank of registers can be transferred to memory or fetched from memory.
- ✓ The base register *Rn* determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similar to the single-register load-store using preindex with writeback.

Example: In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

PRE *mem32[0x80018] = 0x03*

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0 = 0x00080010

r1 = 0x00000000

r2 = 0x00000000

r3 = 0x00000000

LDMIA r0!, {r1–r3}

POST *r0 = 0x0008001c*

r1 = 0x00000001

r2 = 0x00000002

r3 = 0x00000003

The following Figure shows a graphical representation.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

Figure: Pre-condition for LDMIA Instruction

- ✓ The base register *r0* points to memory address 0x80010 in the PRE condition.
- ✓ Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively.
- ✓ After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Figure: Post Condition for LDMIA Instruction

- ✓ The base register *r0* now points to memory address 0x8001c after the last loaded word.
- ✓ Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions.
- ✓ The first word pointed to by register *r0* is ignored and register *r1* is loaded from the next memory location as shown in the following Figure.

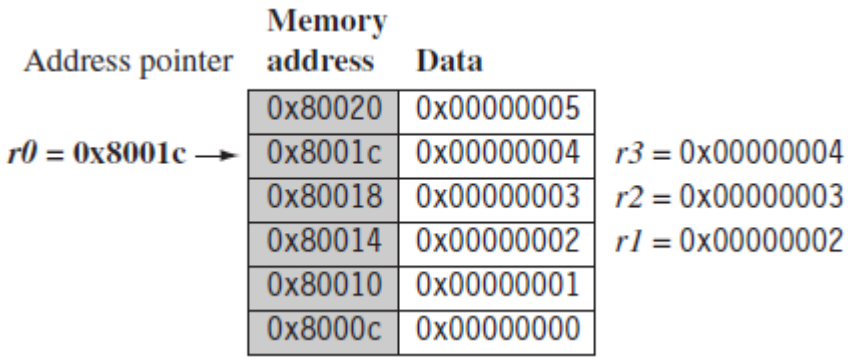


Figure: Post Condition for LDMIB Instruction

- ✓ After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.

- The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register list in reverse order.
- With the increment and decrement load multiples; you can access arrays forwards or backwards.
- They also allow for stack push and pull operations.

The following Table shows a list of load-store multiple instruction pairs.

Table: Load-Store Multiple Pairs when Base Update used

Store Multiple	Load Multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

- If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer.
- This is useful when you need to temporarily save a group of registers and restore them later.

Example: This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

```
PRE   r0 = 0x00009000
      r1 = 0x00000009
      r2 = 0x00000008
      r3 = 0x00000007
STMIB r0!, {r1–r3}
MOV r1, #1
MOV r2, #2
```

```
MOV r3, #3
PRE(2) r0 =  0x0000900c
       r1 = 0x00000001
       r2 = 0x00000002
       r3 = 0x00000003
LDMDA r0!, {r1–r3}
POST   r0 = 0x00009000
       r1 = 0x00000009
       r2 = 0x00000008
       r3 = 0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example: We illustrate the use of the load-store multiple instructions with a block memory copy example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-store multiple instructions, which use the same increment after addressing mode.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source
        loop
; load 32 bytes from source and update r9 pointer
        LDMIA r9!, {r0–r7}
; store 32 bytes to destination and update r10 pointer
        STMIA r10!, {r0–r7}     ; and store them
; have we reached the end
        CMP    r9, r11
        BNE loop
```

- ✓ This routine relies on registers *r9*, *r10*, and *r11* being set up before the code is executed.
- ✓ Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data.
- ✓ LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9* to point to the next block of data to be copied.
- ✓ STMIA copies the contents of registers *r0* to *r7* to the destination memory address pointed to by register *r10*. It also updates *r10* to point to the next destination location.

- ✓ CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached.
- ✓ If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.
- The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed.

The following Figure shows the memory map of the block memory copy and how the routine moves through memory.

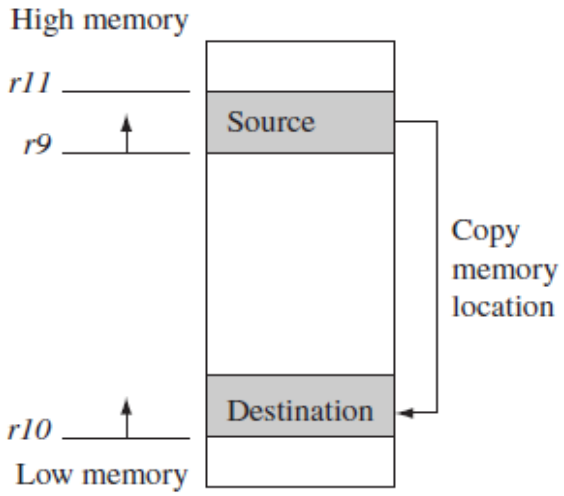


Figure: Block Memory Copy in the Memory map

Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory.

Stack Operation: The ARM architecture uses the load-store multiple instructions to carry out stack operations.

- The *pop operation* (removing data from a stack) uses a load multiple instruction.
- The *push operation* (placing data onto the stack) uses a store multiple instruction.

- ✓ When using a stack you have to decide whether the stack will grow up or down in memory.
  - A stack is either –
    - *ascending (A)* – stacks grow towards higher memory addresses or
    - *descending (D)* – stacks grow towards lower memory addresses.
- ✓ When you use a *full stack (F)*, the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).

✓ If you use an *empty stack (E)* the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

- There are number of load-store multiple addressing mode aliases available to support stack operations (see the following Table).

Table: Addressing Methods for Stack Operations

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

- Next to the *pop* column is the actual load multiple instruction equivalent.
    - For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.
- ARM has specified an *ARM-Thumb Procedure Call Standard (ATPCS)* that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the *pop* and *push* functions, respectively.

Example: The STMFD instruction pushes registers onto the stack, updating the *sp*. The following Figure shows a *push* onto a full descending stack.

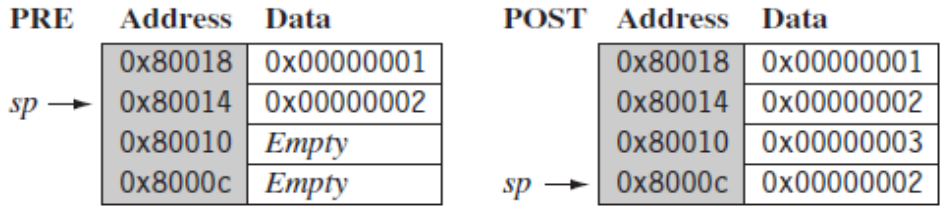


Figure: STMFD Instruction – Full Stack *push* Operation

You can see that when the stack grows the stack pointer points to the last full entry in the stack.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080014

STMFD sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x0008000c

Example: The following Figure shows a *push* operation on an empty stack using the STMED instruction.

PRE	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Figure: STMED Instruction – Empty Stack *push* Operation

The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080010

STMED sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080008

- ✓ When handling a checked stack there are three attributes that need to be preserved: the stack base, the stack pointer, and the stack limit.
- ✓ The stack base is the starting address of the stack in memory.
- ✓ The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a stack overflow error has occurred.
- ✓ Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow
        SUB sp, sp, #size
        CMP sp, r10
        BLLO _stack_overflow        ; condition
```

- ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled.
- The BLLO instruction is a branch with link instruction plus the condition mnemonic LO.
  - If *sp* is less than register *r10* after the new items are pushed onto the stack, then *stack overflow* error has occurred.
  - If the stack pointer goes back past the stack base, then a *stack underflow* error has occurred.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register.

This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Swap cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete. Also, swap instruction allows for both a word and a byte swap.

Example: The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

```
PRE   mem32[0x9000] = 0x12345678
      r0  =  0x00000000
      r1 = 0x11112222
      r2 = 0x00009000
SWP r0, r1, [r2]
POST mem32[0x9000] = 0x11112222
      r0 = 0x12345678
      r1 = 0x11112222
      r2 = 0x00009000
```

Example: This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
spin
      MOV r1, =semaphore
      MOV r2, #1
      SWP r3, r2, [r1]        ; hold the bus until complete
      CMP r3, #1
      BEQ   spin
```

The address pointed to by the semaphore either contains the value *0* or *1*. When the semaphore equals *1*, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value *0*.

SOFTWARE INTERRUPT INSTRUCTION:

A *software interrupt instruction (SWI)* causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset *0x8* in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Example: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

PRE *cpsr = nzcVqift_USER*

pc = 0x00008000

lr = 0x003fffff ;lr = r14

r0 = 0x12

0x00008000 SWI 0x123456

POST *cpsr = nzcVqIft_SVC*

spsr = nzcVqift_USER

pc = 0x00000008

lr = 0x00008004

r0 = 0x12

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter *0x12*. The return values are also passed back via registers.

Code called the ***SWI handler*** is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

SWI_Number = <SWI instruction> ***AND NOT*** *(0xff000000)*

Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

Example: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*.

You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
; Store registers r0-r12 and the link register
        STMFD sp!, {r0–r12, lr}
; Read the SWI instruction
        LDR r10, [lr, #–4]
; Mask off top 8 bits
        BIC r10, r10, #0xff000000
; r10 - contains the SWI number
        BL service_routine
; return from SWI handler
        LDMFD sp!, {r0–r12, pc}^
```

The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

PROGRAM STATUS REGISTER INSTRUCTIONS:

The ARM instruction set provides two instructions to directly control a *program status register (psr)*.

- ✓ The *MRS instruction* transfers the contents of either the *cpsr* or *spsr* into a register.
- ✓ The *MSR instruction* transfers the contents of a register into the *cpsr* or *spsr*.

Together these instructions are used to read and write the *cpsr* and *spsr*.

In the syntax we can see a *label* called fields. This can be any combination of *control (c)*, *extension (x)*, *status (s)*, and *flags (f)*.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

These fields relate to particular byte regions in a *psr*, as shown in the following Figure.

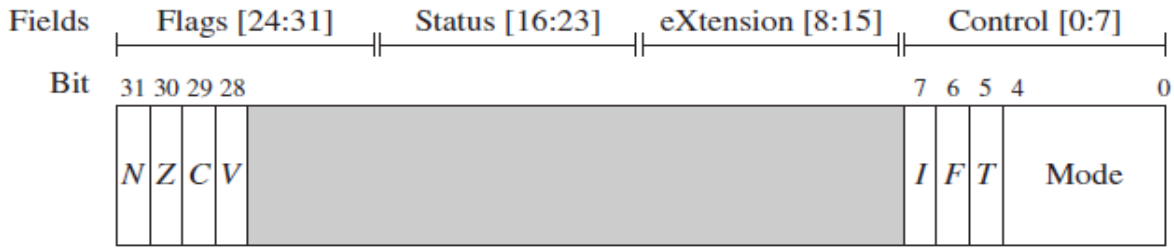


Figure: *psr* Byte Fields

The *c* field controls the interrupt masks, Thumb state, and processor mode.

The following Example shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

Example: The MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit *7* of *r1*. Register *r1* is then copied back into the *cpsr*, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

PRE *cpsr = nzcvqIFt_SVC*

MRS r1, cpsr

BIC r1, r1, #0x80 ; 0b01000000

MSR cpsr_c, r1

POST *cpsr = nzcvqiFt_SVC*

This example is in SVC mode. In user mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions:

Coprocessor instructions are used to extend the instruction set.

- ✓ A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- ✓ The coprocessor instructions include data processing, register transfer, and memory transfer instructions.
- ✓ Note that these instructions are only used by cores with a coprocessor.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

- ✓ In the syntax of the coprocessor instructions,
  - The *cp* field represents the coprocessor number between *p0* and *p15*
  - The *opcode* fields describe the operation to take place on the coprocessor.
  - The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.
- ✓ The coprocessor operations and registers depend on the specific coprocessor you are using.
- ✓ *Coprocessor 15 (CP15)* is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example: This example shows a *CP15* register being copied into a general-purpose register.

; transferring the contents of CP15 register c0 to register r10

MRC p15, 0, r10, c0, c0, 0

Here *CP15 register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*.

LOADING CONSTANTS:

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.

To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

- The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.
- The second pseudo-instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example: This example shows an LDR instruction loading a 32-bit constant *0xff00ffff* into register *r0*.

```
        LDR r0, [pc, #constant_number-8-{PC}]
:
constant_number
        DCD    0xff00ffff
```

This example involves a memory access to load the constant, which can be expensive for time-critical routines.

The following Example shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Example: Loading the constant *0xff00ffff* using an MVN.

```
PRE     none...
MVN r0, #0x00ff0000
POST    r0 = 0xff00ffff
```

As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.

The LDR pseudo-instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a literal pool—a data area embedded within the code.

The following Table shows two pseudo-code conversions.

Table: LDR pseudo-instruction Conversion

Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load.

Another useful pseudo-instruction is the ADR instruction, or address relative. This instruction places the address of the given label into register *Rd*, using a *pc*-relative add or subtract.

MICROCONTROLLER AND EMBEDDED SYSTEMS

ARM PROGRAMMING USING ASSEMBLY LANGUAGE

WRITING ASSEMBLY CODE:

This section gives examples showing how to write basic assembly code. Also, this section uses the *ARM macro assembler* ***armasm*** for examples.

Example 1:

This example shows how to convert a *C* function to an assembly function—usually the first stage of assembly optimization. Consider the simple *C* program *main.c* following that prints the squares of the integers from 0 to 9:

```
#include <stdio.h>

int square(int i);

int main(void)
{
  int i;

  for (i=0; i<10; i++)
  {
    printf("Square of %d is %d\n", i, square(i));
  }
}

int square(int i)
{
  return i*i;
}
```

Let's see how to replace square by an assembly function that performs the same action. Remove the *C* definition of square, but not the declaration (the second line) to produce a new *C* file *main1.c*. Next add an *armasm* assembler file *square.s* with the following contents:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  square

        ; int square(int i)
square
        MUL   r1, r0, r0   ; r1 = r0 * r0
        MOV   r0, r1       ; r0 = r1
        MOV   pc, lr       ; return r0
        END
```

- The *AREA* directive names the area or code section that the code lives in. If you use non-alphanumeric characters in a symbol or area name, then enclose the name in vertical bars. Many non-alphanumeric characters have special meanings otherwise. In the previous code we define a read-only code area called *.text*.
- The *EXPORT* directive makes the symbol square available for external linking. At line six we define the symbol square as a code label. Note that *armasm* treats non-indented text as a label definition.
- When square is called, the parameter passing is defined by the *ARM-Thumb procedure call standard (ATPCS)*. The input argument is passed in register *r0*, and the return value is returned in register *r0*. The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into *r1* and move this to *r0*.

- The *END* directive marks the end of the assembly file.

Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

Example 1 only works if you are compiling your *C* as ARM code. If you compile your *C* as Thumb code, then the assembly routine must return using a *BX* instruction.

Example 2: When calling ARM code from *C* compiled as Thumb, the only change required to the assembly in *Example 1* is to change the return instruction to a *BX*. *BX* will return to ARM or Thumb state according to bit *0* of *lr*. Therefore this routine can be called from ARM or Thumb. Use *BX lr* instead of *MOV pc, lr* whenever your processor supports *BX* (*ARMv4T* and above). Create a new assembly file *square2.s* as follows:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  square

        ; int square(int i)
square
        MUL   r1, r0, r0   ; r1 = r0 * r0
        MOV   r0, r1       ; r0 = r1
        BX    lr           ; return r0

        END
```

With this example we build the *C* file using the Thumb *C* compiler *tcc*. We assemble the assembly file with the interworking flag enabled so that the linker will allow the Thumb *C* code to call the ARM assembly code. You can use the following commands to build this example:

```
tcc -c main1.c
armasm -apcs /interwork square2.s
armlink -o main2.axf main1.o square2.o
```

Example 3: This example shows how to call a subroutine from an assembly routine. We will take Example 1 and convert the whole program (including main) into assembly. We will call the *C* library routine *printf* as a subroutine. Create a new assembly file *main3.s* with the following contents:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  main

        IMPORT  |Lib$$Request$$armlib|, WEAK
        IMPORT  __main    ; C library entry
        IMPORT  printf    ; prints to stdout
i       RN 4

          ; int main(void)

  main
          STMFD   sp!, {i, lr}
          MOV     i, #0
  loop
          ADR     r0, print_string
          MOV     r1, i
          MUL     r2, i, i
          BL      printf
          ADD     i, i, #1
          CMP     i, #10
          BLT     loop
          LDMFD   sp!, {i, pc}

  print_string
          DCB     "Square of %d is %d\n", 0

          END
```

- The *IMPORT* directive is used to declare symbols that are defined in other files.
- The imported symbol *Lib$$Request$$armlib* makes a request that the linker links with the standard ARM *C* library.
    - The *WEAK* specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero.
- The second imported symbol___*main* is the start of the *C* library initialization code.

You only need to import these symbols if you are defining your own main; a main defined in *C* code will import these automatically for you. Importing *printf* allows us to call that *C* library function.

- The *RN* directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*.

- Using register names makes the code more readable. It is also easier to change the allocation of variables to registers at a later date. Recall that *ATPCS* states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i* (*r4*), and calling *printf* will corrupt *lr*. Therefore we stack these two registers at the start of the function using an *STMFD* instruction. The *LDMFD* instruction pulls these registers from the stack and returns by writing the return address to *pc*.

- The *DCB* directive defines byte data described as a string or a comma-separated list of bytes.

To build this example you can use the following command line script:

```
armasm main3.s
armlink -o main3.axf main3.o
```

Note that Example 3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 2 then we must be capable of returning to Thumb code. For architectures before *ARMv5* we must use a *BX* to return. Change the last instruction to the two instructions:

```
LDMFD   sp!, {i, lr}
BX      lr
```

Example 4: This example defines a function *sumof* that can sum any number of integers. The arguments are the number of integers to sum followed by a list of the integers. The *sumof* function is written in assembly and can accept any number of arguments. Put the *C* part of the example in a file *main4.c*:

```
#include <stdio.h>

/* N is the number of values to sum in list ... */
int sumof(int N, ...);

int main(void)
{
   printf("Empty sum=%d\n", sumof(0));
   printf("1=%d\n", sumof(1,1));
   printf("1+2=%d\n", sumof(2,1,2));
   printf("1+2+3=%d\n", sumof(3,1,2,3));
   printf("1+2+3+4=%d\n", sumof(4,1,2,3,4));
   printf("1+2+3+4+5=%d\n", sumof(5,1,2,3,4,5));
   printf("1+2+3+4+5+6=%d\n", sumof(6,1,2,3,4,5,6));
}
```

Next define the *sumof* function in an assembly file *sumof.s*:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  sumof

N       RN 0    ; number of elements to sum
sum     RN 1    ; current sum

        ; int sumof(int N, ...)
sumof
        SUBS    N, N, #1        ; do we have one element
        MOVLT   sum, #0         ; no elements to sum!
        SUBS    N, N, #1        ; do we have two elements
        ADDGE   sum, sum, r2
        SUBS    N, N, #1        ; do we have three elements
        ADDGE   sum, sum, r3
        MOV     r2, sp          ; top of stack
loop
        SUBS    N, N, #1        ; do we have another element
        LDMGEFD r2!, {r3}       ; load from the stack

        ADDGE   sum, sum, r3
        BGE     loop
        MOV     r0, sum
        MOV     pc, lr          ; return r0

        END
```

The code keeps count of the number of remaining values to *sum*, *N*. The first three values are in registers *r1*, *r2*, *r3*. The remaining values are on the stack (Recall that *ATPCS* places the first four arguments in registers r0 to r3. Subsequent arguments are placed on the stack). You can build this example using the commands –

```
armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o
```

PROFILING AND CYCLE COUNTING:

- ✓ The first stage of any optimization process is to identify the critical routines and measure their current performance. A ***profiler*** is a tool that measures the proportion of time or processing cycles spent in each subroutine. You use a profiler to identify the most critical routines.

- ✓ A ***cycle counter*** measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.
- ✓ The ARM simulator used by the *ADS1.1 debugger* is called the *ARMulator* and provides profiling and cycle counting features.
    - The *ARMulator profiler* works by sampling the program counter *pc* at regular intervals. The profiler identifies the function the *pc* points to and updates a hit counter for each function it encounters. Another approach is to use the trace output of a simulator as a source for analysis.
    - The accuracy of a *pc*-sampled profiler is limited, as it can produce meaningless results if it records too few samples.
- ✓ ARM implementations do not normally contain cycle-counting hardware; so to easily measure cycle counts you should use an ARM debugger with ARM simulator.
    - You can configure the *ARMulator* to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.

INSTRUCTION SCHEDULING:

The time taken to execute instructions depends on the implementation pipeline. For this section, we assume *ARM9TDMI* pipeline timings. The following rules summarize the cycle timings for common instruction classes on the *ARM9TDMI*.

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ✓ *ALU* operations such as addition, subtraction, and logical operations take one cycle.
- ✓ This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.

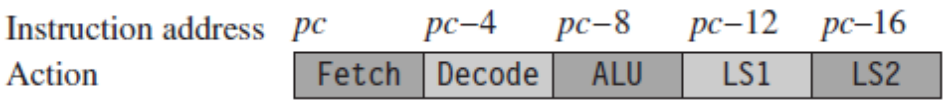
- Load instructions that load *N* 32-bit words of memory such as *LDR* and *LDM* take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle.
    - The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system. An *LDM* of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
    - Load instructions that load 16-bit or 8-bit data such as *LDRB*, *LDRSB*, *LDRH*, and *LDRSH* take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system.
- Branch instructions take three cycles.

- Store instructions that store *N* values take *N* cycles. This assumes zero-wait-state memory for an un-cached system, or a cache hit or a write buffer with *N* free entries for a cached system. An *STM* of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product.

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The *ARM9TDMI* processor performs five operations in parallel:

- ***Fetch:*** Fetch from memory the instruction at address *pc*. The instruction is loaded into the core and then processes down the core pipeline.
- ***Decode:*** Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- ***ALU:*** Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address *pc − 8* (ARM state) or *pc − 4* (Thumb state).
    - Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation.
    - Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.
- ***LS1:*** Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- ***LS2:*** Extract and zero- or sign-extend the data loaded by a byte or half-word load instruction. If the instruction is not a load of an 8-bit byte or 16-bit half-word item, then this stage has no effect.

The following Figure shows a simplified functional view of the five-stage *ARM9TDMI* pipeline.



Note that multiply and register shift operations are not shown in the figure.

After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that *pc* points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address *pc − 8* in parallel with fetching the instruction at address *pc*.

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a *pipeline hazard* or *pipeline interlock*.

Example 5: This example shows the case where there is no interlock.

```
ADD   r0, r0, r1
ADD   r0, r0, r2
```

This instruction pair takes two cycles. The ALU calculates *r0 + r1* in one cycle. Therefore this result is available for the ALU to calculate *r0 + r2* in the second cycle.

Example 6: This example shows a one-cycle interlock caused by load use.

```
LDR   r1, [r2, #4]
ADD   r0, r0, r1
```

This instruction pair takes three cycles. The ALU calculates the address *r2 + 4* in the first cycle while decoding the *ADD* instruction in parallel. However, the *ADD* cannot proceed on the second cycle because the load instruction has not yet loaded the value of *r1*. Therefore the pipeline stalls for one cycle while the load instruction completes the *LS1* stage. Now that *r1* is ready, the processor executes the *ADD* in the ALU on the third cycle.

The following Figure illustrates how this interlock affects the pipeline.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

The processor stalls the *ADD* instruction for one cycle in the ALU stage of the pipeline while the load instruction completes the *LS1* stage. Figure denotes this stall by *italic ADD*. Since the *LDR* instruction proceeds down the pipeline, but the *ADD* instruction is stalled, a gap opens up between them. This gap is sometimes called a pipeline ***bubble***. We've marked the bubble with a *dash*.

Example 7: This example shows a one-cycle interlock caused by delayed load use.

```
LDRB   r1, [r2, #1]
ADD    r0, r0, r2
EOR    r0, r0, r1
```

This instruction triplet takes four cycles. Although the *ADD* proceeds on the cycle following the load byte, the *EOR* instruction cannot start on the third cycle. The *r1* value is not ready until the load instruction completes the *LS2* stage of the pipeline. The processor stalls the *EOR* instruction for one cycle. Note that the *ADD* instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not! The following Figure shows how this sequence progresses through the processor pipeline. The *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	*EOR*	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Example 8: This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

```
        MOV   r1, #1
        B     case1
        AND   r0, r0, r1
        EOR   r2, r2, r3
        ...
case1
        SUB   r0, r0, r1
```

The three executed instructions take a total of five cycles. The *MOV* instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles. Finally, the *SUB* instruction executes normally. The following Figure illustrates the pipeline state on each cycle. The pipeline drops the two instructions following the branch when the branch takes place.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Scheduling of Load Instructions:

Load instructions occur frequently in compiled code, accounting for approximately one-third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problem of *C* limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Consider an example of a memory-intensive task. The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process.

```
void str_tolower(char *out, char *in)
{
  unsigned int c;

  do
  {
    c = *(in++);
    if (c>='A' && c<='Z')
    {
      c = c + ('a' -'A');
    }
    *(out++) = (char)c;
  } while (c);
}
```

```
str_tolower
        LDRB    r2,[r1],#1    ; c = *(in++)
        SUB     r3,r2,#0x41   ; r3 = c -'A'
        CMP     r3,#0x19      ; if (c <='Z'-'A')
        ADDLS   r2,r2,#0x20   ;    c +='a'-'A'
        STRB    r2,[r0],#1    ; *(out++) = (char)c
        CMP     r2,#0         ; if (c!=0)
        BNE     str_tolower   ;    goto str_tolower
        MOV     pc,r14        ; return
```

The compiler generates the above compiled output. Notice that the compiler optimizes the condition (*c >= 'A' && c <= 'Z'*) to the check that *0 <= c-'A' <= 'Z'-'A'*. The compiler can perform this check using a single unsigned comparison.

Unfortunately, the *SUB* instruction uses the value of *c* directly after the *LDRB* instruction that loads *c*. Consequently, the *ARM9TDMI* pipeline will stall for two cycles. The compiler can't do any better since everything following the load of *c* depends on its value.

However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by *preloading* and *unrolling*.

» *Load Scheduling by Preloading* & *Load Scheduling by Unrolling – Self Study*.

REGISTER ALLOCATION:

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the *stack pointer, r13,* and the *program counter, r15*. For a function to be *ATPCS* compliant it must preserve the callee values of registers *r4* to *r11*. *ATPCS* also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

```
routine_name
        STMFD sp!,      {r4-r12, lr}      ; stack saved registers
          ; body of routine
          ; the fourteen registers r0-r12 and lr are available
        LDMFD sp!,      {r4-r12, pc}      ; restore registers and return
```

The only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other *ATPCS* routines. For *ARMv5* and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an *ARMv4T* processor, then modify the template as follows:

```
routine_name
        STMFD sp!,      {r4-r12, lr}      ; stack saved registers
          ; body of routine
          ; registers r0-r12 and lr available
        LDMFD sp!,      {r4-r12, lr}      ; restore registers
        BX         lr                     ; return, with mode switch
```

Allocating Variables to Register Numbers:

When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are orthogonal with respect to register number. In other words, specific register numbers do not have specific roles. If you swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change.

However, there are several cases where the physical number of the register is important:

- ✓ *Argument registers:* The *ATPCS* convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack.
  - The return value must be placed in *r0*.
- ✓ *Registers used in a load or store multiple:* Load and store multiple instructions *LDM* and *STM* operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- ✓ *Load and store double word:* The *LDRD* and *STRD* instructions introduced in *ARMv5E* operate on a pair of registers with sequential register numbers, *Rd* and *Rd + 1*. Furthermore, *Rd* must be an even register number.

Using More Than 14 Local Variables:

If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack. The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

Making the Most of Available Registers:

On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance.

CONDITIONAL EXECUTION:

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an *S* suffix to the instruction mnemonic.

Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the *S* suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example 17: The following *C* code converts an unsigned integer $0 \le i \le 15$ to a hexadecimal character *c*:

```
if (i<10)
{
  c = i + '0';
}
else
{
  c = i + 'A'-10;
}
```

We can write this in assembly using conditional execution rather than conditional branches:

```
CMP     i, #10
ADDLO   c, i, #'0'
ADDHS   c, i, #'A'-10
```

The sequence works since the first *ADD* does not change the condition codes. The second *ADD* is still conditional on the result of the compare.

Conditional execution is even more powerful for cascading conditions.

Example 18: The following *C* code identifies if *c* is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

In assembly you can write this using conditional comparison:

```
TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1
```

As soon as one of the *TEQ* comparisons detects a match, the *Z* flag is set in the *cpsr*. The following *TEQNE* instructions have no effect as they are conditional on $Z = 0$. The next instruction to have effect is the *ADDEQ* that increments vowel. You can use this method whenever all the comparisons in the if statement are of the same type.

Example 19: Consider the following code that detects if c is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq c \leq limit$. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

Note that the logical operations *AND* and *OR* are related by the standard logical relations as shown in the following Table. You can invert logical expressions involving *OR* to get an expression involving *AND*, which can often be useful in simplifying or rearranging logical expressions.

Inverted expression	Equivalent
!(a && b)	(!a) || (!b)
!(a || b)	(!a) && (!b)

LOOPING CONSTRUCTS:

Most routines critical to performance will contain a loop. Note that, ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

Decremented Counted Loops:

For a decrementing loop of *N* iterations, the loop counter *i* counts down from *N* to *1* inclusive. The loop terminates with *i = 0*. An efficient implementation is

```
      MOV i, N
loop
      ; loop body goes here and i=N,N-1,...,1
      SUBS i, i, #1
      BGT  loop
```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On *ARM7* and *ARM9* this overhead costs four cycles per loop. If *i* is an array index, then you may want to count down from *N−1* to *0* inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```
      SUBS i, N, #1
loop
      ; loop body goes here and i=N-1,N-2,...,0
      SUBS i, i, #1
      BGE  loop
```

In this arrangement the *Z* flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the *EQ* and *NE* conditions. For example, if you preload data for the next loop, then you want to avoid the preload on the last loop. You can make all preload operations conditional on *NE*.

There is no reason why we must decrement by one on each loop. Suppose we require *N/3* loops; rather than attempting to divide *N* by three, it is far more efficient to subtract three from the loop counter on each iteration:

```
        MOV i, N
loop
        ; loop body goes here and iterates (round up)(N/3) times
        SUBS i, i, #3
        BGT  loop
```

Unrolled Counted Loops:

Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome.

Multiple Nested Loops:

How many loop counters does it take to maintain multiple nested loops? Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32. We can combine the loop counts within a single register, placing the innermost loop count at the highest bit positions.

Other Counted Loops:

You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from *N* to *1* or *N −1* to *0*. For example, you may want to select bits out of a data register one at a time; in this case you may want a power-of-two mask that doubles on each iteration.

The following subsections show useful looping structures that count in different patterns. They use only a single instruction combined with a branch to implement the loop.

Negative Indexing: This loop structure counts from *−N* to *0* (inclusive or exclusive) in steps of size *STEP*.

```
        RSB     i, N, #0        ; i=-N
loop
        ; loop body goes here and i=-N,-N+STEP,...,
        ADDS    i, i, #STEP
        BLT     loop            ; use BLT or BLE to exclude 0 or not
```

Logarithmic Indexing: This loop structure counts down from 2^N to *1* in powers of two. For example, if *N = 4*, then it counts 16, 8, 4, 2, 1.

```
        MOV     i, #1
        MOV     i, i, LSL N
loop
        ; loop body
        MOVS    i, i, LSR#1
        BNE     loop
```

The following loop structure counts down from an *N-bit* mask to a one-bit mask. For example, if *N = 4*, then it counts 15, 7, 3, 1.

```
        MOV     i, #1
        RSB     i, i, i, LSL N  ; i=(1<<N)-1
loop
        ; loop body
        MOVS    i, i, LSR#1
        BNE     loop
```

_____*********_____

****

MODULE – 3

EMBEDDED SYSTEM COMPONENTS

INTRODUCTION TO EMBEDDED SYSTEMS

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).

Every embedded system is unique, and the hardware as well as the firmware is highly specialized to the application domain. Embedded systems are becoming an inevitable part of any product or equipment in all fields including household appliances, telecommunications, medical equipment, industrial control, consumer products, etc.

Characteristics of Embedded Systems:

- ✓ Embedded Systems must be highly reliable and stable.
- ✓ Embedded Systems have minimal or no user interface.
- ✓ Embedded Systems are usually feedback oriented or reactive.
- ✓ Embedded Systems are typically designed to meet real time constraints.
- ✓ Embedded Systems have limited memory and limited number of peripherals.
- ✓ Embedded Systems are typically designed for specific application or purpose.
- ✓ Embedded Systems are designed for low power consumption, as they use battery power.

EMBEDDED SYSTEMS versus GENERAL COMPUTING SYSTEMS:

General Computing System	Embedded System
1. A combination of generic hardware and a General Purpose Operating System (GPOS) for executing a variety of applications.	1. A combination of special purpose hardware embedded OS for executing a specific set of applications.
2. Applications are alterable (programmable) by the user.	2. The firmware is pre-programmed and it is non-alterable by the end-user (there may be exceptions).
3. Performance is the key deciding factor in the selection of the system. Always, 'Faster is Better'.	3. Application-specific requirements (like performance, power requirements, memory usage, etc.).
4. Less/ not at all tailored towards reduced operating power requirements.	4. Highly tailored to take advantage of the power saving modes supported by the hardware and the operating system.
5. Need not be deterministic in execution behavior; response requirements are not time critical.	5. Execution behavior is deterministic for certain types of embedded systems like 'Hard Real Time' systems.

MICROCONTROLLER AND EMBEDDED SYSTEMS

HISTORY OF EMBEDDED SYSTEMS:

- Embedded systems were existing even before the Information Technology revolution. Initially, embedded systems were built around vacuum tube and transistor technologies; and embedded algorithm was developed by using low level programming languages.
    - Advances in semiconductor and nano-technology and IT revolution gave way to development of miniature embedded systems.

- ***Apollo Guidance Computer*** (AGC) developed (during 1960) by MIT Instrumentation Laboratory for the lunar expedition is the first recognized modern embedded system.
    - AGC included both Command Module (CM-to encircle the moon) and Lunar Excursion Module (LEM-to go down to the moon surface and land there safely).
    - There were 16 reaction control thrusters, a descent engine (designed to provide thrust to the lunar model out of the lunar orbit and land it safely on moon) and an ascent engine.
    - Original design was based on 4K words of fixed memory (ROM) and 256 words of erasable memory (RAM); which has been enhanced (during 1963) to 10K fixed and 1K erasable memory. The clock frequency was 1.024 MHz.
    - The computing unit of AGC consisted of approximately 11 instructions on 16-bit word logic.
    - A calculator type user interface was given and is known as DSKY (display/ keyboard).

- The first mass-produced embedded system was the guidance computer, ***Autonetics D-17***, for the Minuteman-I missile in 1961; built using discrete transistor logic and a hard-disk for main memory.

- The first microprocessor, the Intel 4004, was designed for calculators and other small systems; but still required many external memory and support chips.
- First microcontroller, TMS 1000, developed in 1974 by Texas Instruments. It had ROM, RAM, and clock circuitry on the chip along with the processing chip.
- In 1980, Intel introduced 8051 MCU and called it MCS-51 architecture.

- Laser and Inkjet printers emerged during 1980s; and early 1990, cell phones having five or six DSPs and CPUs emerged.

MICROCONTROLLER AND EMBEDDED SYSTEMS

CLASSIFICATION OF EMBEDDED SYSTEMS:

Classification Based on Generation:

Generation with Example	Description
First Generation (1G) - Digital telephone keypads - Stepper motor	✓ 8-bit microprocessor and 4-bit microcontroller like 8085 and Z80 was used in 1G. ✓ Hardware circuit was simple. ✓ Assembly code is used for developing firmware.
Second Generation (2G) - Data acquisition systems like ADC, SCADA system	✓ Uses 16-bit microprocessor and 8-bit microcontroller. ✓ They are more complex and powerful than 1G microprocessor and microcontroller.
Third Generation (3G) - Robotics	✓ Uses 32-bit microprocessor and 16-bit microcontroller. ✓ Domain specific processor and controllers are used.
Fourth Generation (4G) - Smart phones	✓ Uses 64-bit microprocessor and 32-bit microcontroller. ✓ The concept of system on chips, multi-core processors evolved. ✓ Highly complex and very powerful.

Classification Based on Complexity and Performance:

1. ***Small-Scale Embedded Systems:***
   - Embedded systems which are simple in application needs and the performance parameters are not time critical (E.g.: Electronic toy).
   - Small-scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors/ microcontrollers.
   - It may or may not contain an operating system for its functioning.
2. ***Medium-Scale Embedded Systems:***
   - Embedded systems which are slightly complex in hardware and firmware (software) requirements.
   - Medium-scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/ microcontrollers or digital signal processors.
   - They usually contain an embedded operating system (general purpose/ real-time).
3. ***Large-Scale Embedded Systems/ Complex Systems:***
   - Embedded systems which involve highly complex hardware and firmware. They are employed in mission critical applications demanding high performance.
   - Large-scale embedded 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.

- They usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritization, and management.

MAJOR APPLICATION AREAS OF EMBEDDED SYSTEMS:

Embedded systems play a vital role in our day-to-day life, starting from home to computer industry. 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 technology and wearable computing devices (Apple watch, Microsoft band, Fitbit fitness trackers, etc.).

The application areas and the products in the embedded domain are countless. A few of the important domains and products are listed below:

1. *Consumer Electronics:* Camcorders, cameras, etc.
2. *Household Appliances:* Television, DVD players, washing machine, fridge, 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:* Anti-lock breaking systems (ABS), engine control, ignition 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. *Healthcare:* 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.
12. *Wearable Devices:* Health and fitness trackers, Smartphone screen extension for notifications, etc.
13. Cloud Computing and Internet of Things (IoT).

PURPOSE OF EMBEDDED SYSTEMS:

As mentioned in the previous section, embedded systems are used in various domains like consumer electronics, home automation, telecommunications, automotive industry, healthcare, control & instrumentation, retail and banking applications, etc. Each embedded system is designed to serve the purpose of any one or a combination o the following tasks:

1. ***Data Collection, Storage, Representation***
   - Data is collection of facts, such as values or measurements. It can be numbers, words, measurements, observations, or even just description of things.

- Purpose of embedded system design is data collection. It performs acquisition of data from the external world.
- Data collection is usually done for storage, analysis, manipulation, and transmission.
- Data can be analog or digital.
- Embedded systems with analog data capturing techniques collect data directly in the form of analog signal; whereas embedded systems with digital data collection mechanism convert the analog signal to corresponding digital signal using analog to digital (A/D) converters.
- If the data is digital, it can be directly captured by digital embedded system.
    - A digital camera is a typical example of an embedded system with data collection, storage, and representation of data. Images are captured and captured image may be stored within the memory of the camera. The captured image can also be presented to the user through a graphic LCD (Liquid Crystal Display) unit.

2. Data Communication

- Embedded data communication systems are deployed in applications ranging from simple home networking systems to complex satellite communication systems.
    - Network hubs, routers, switches are examples of dedicated data transmission embedded systems.
- Data transmission is in the form of wire medium or wireless medium. Initially wired medium is used by embedded systems; and as technology changes, wireless medium becomes de-facto standard in embedded systems.
    - USB, TCP/ IP are examples of wired communication; and BlueTooth, ZigBee and Wi-Fi are examples for wireless communication.
- Data can be transmitted by analog means or by digital means.

3. Data (Signal) Processing

- Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, audio-video codec, transmission applications, etc.
    - A digital hearing aid is a typical example of an embedded system employing data processing.

4. Monitoring

- Almost all embedded products coming under the medical domain are with monitoring functions.
    - Patient heart beat is monitored by Electro cardiogram (ECG) machine.

- Digital CRO, digital multi-meters, and logic analyzers are examples of monitoring embedded systems.

5. Control

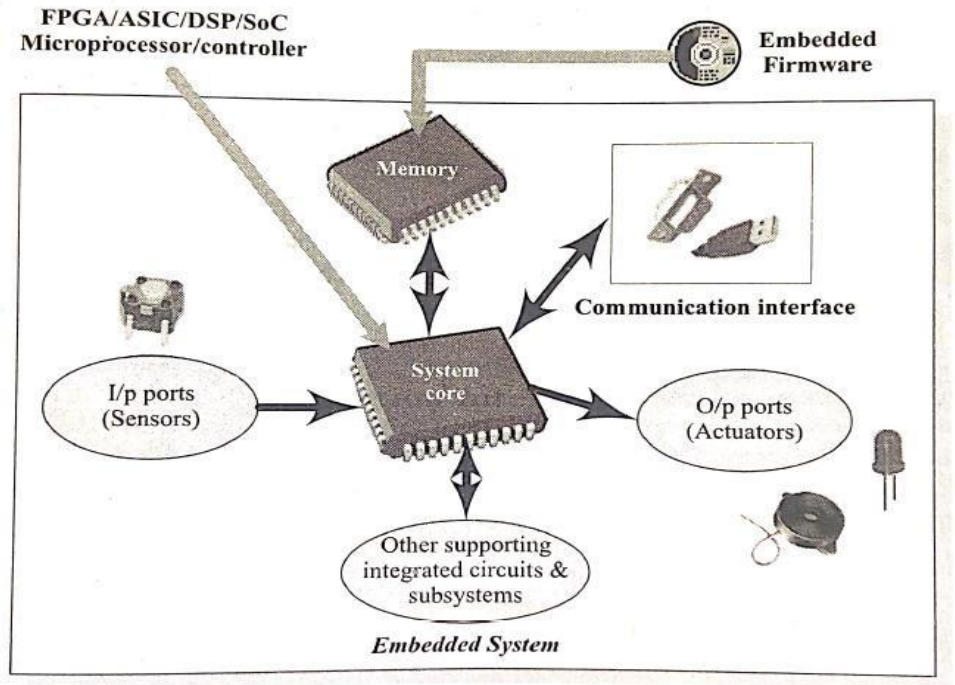
- Sensors and actuators are used for controlling the system.
  - Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable.
  - Actuators connected to output port are controlled according to the changes in input variable.
- Air conditioner system used in our home to control the room temperature to a specified limit is a typical example for embedded system for control purpose. The air conditioner's compressor unit (actuator) is controlled according to the current room temperature (sensor) and the desired room temperature set by the user.

6. Application Specific User Interface

- These are embedded systems with application-specific user interfaces like buttons, switches, keypad, lights, bells, display units, etc.
- Mobile phone is an example for this. In mobile phone, the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.

THE TYPICAL EMBEDDED SYSTEM

A typical embedded system (shown in the following Figure) contains a single chip controller, which acts as the master brain of the system.



The *controller* can be

- ✓ a Microprocessor (Intel 8085) or
- ✓ a Microcontroller (Atmel AT89C51) or
- ✓ a Field Programmable Gate Array (FPGA) device (Xilinx Spartan) or
- ✓ a Digital Signal Processor (DSP) (Blackfin® Processors from Analog Devices) or
- ✓ an Application Specific Integrated Circuit (ASIC)/
- ✓ Application Specific Standard Product (ASSP) (ADE7760 Single Phase Energy Metreing IC from Analog Devices for energy metering applications).

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 devices connected to the O/P ports of the system, in response to the input signals provided by the end users or Sensors which are connected to the input ports. Hence an embedded system can be viewed as a *reactive system*.

Key boards, push button switches, etc. are examples for *common user interface input devices* where as LEDs, liquid crystal displays, piezoelectric 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.

CORE OF THE EMBEDDED SYSTEM:

Embedded systems are domain and application specific and are built around a central core. The core of the embedded system falls into any one of the following categories:

1) General Purpose and Domain Specific Processors
    a. Microprocessors
    b. Microcontrollers
    c. Digital Signal Processors
2) Application Specific Integrated Circuits (ASICs)
3) Programmable Logic Devices (PLDs)
4) Commercial off-the-shelf Components (COTS)

General Purpose and Domain Specific Processors:

Almost 80% of the embedded systems are processor/ controller based. The processor may be a microprocessor or a microcontroller or a digital signal processor, depending on the domain and application.

MICROCONTROLLER AND EMBEDDED SYSTEMS

Microprocessors: A *Microprocessor* is a silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations. In general, the CPU contains the Arithmetic and Logic Unit (ALU), control unit and working registers. A microprocessor is a dependent unit and it requires the combination of' other hardware like memory, timer unit, and interrupt controller, etc., for proper functioning.

Intel claims the credit for developing the first microprocessor unit, Intel 4004, a 4-bit processor which was released in November 1971. It was designed for older day's calculators. In April 1974, Intel launched the first 8-bit processor, the Intel 8080, with 16-bit address bus and program counter and seven 8-bit registers. Intel 8080 was the most commonly used processors for industrial control and other embedded applications in the 1975s.

Immediately after the release of Intel 8080, Motorola also entered the market with their processor, Motorola 6800 with a different architecture and instruction set compared to 8080.

In 1976 Intel came up with the upgraded version of 8080 – Intel 8085, with two newly added instructions, three interrupt pins and serial I/0. Clock generator and bus controller circuits were built-in and the power supply part was modified to a single +5 V supply.

In July 1976 Zilog entered the microprocessor market with its Z80 processor as competitor to Intel.

Intel, AMD, Freescale, GLOBALFOUNDRIES, TI, Cyrix, NVIDIA, Qualcomm, MediaTek, etc. are the key players in the processor market. Intel still leads the market with cutting edge technologies in the processor industry.

Microcontrollers: A *Microcontroller* is a highly integrated chip that contains a CPU, scratch pad RAM, special and general purpose register arrays, on chip ROM/ FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports. Microcontrollers can be considered as a super set of microprocessors. Since a microcontroller contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in place of microprocessors. Apart from this, they are cheap, cost effective and are readily available in the market.

Texas Instrument's TMS 1000 (1974) is considered as the world's first microcontroller. TI followed Intel's 4004, 4-bit processor design and added some amount of RAM, program storage memory (ROM) and 1/O support on a single chip, there by eliminated the requirement of multiple hardware chips for self-functioning.

In 1977 Intel entered the microcontroller market with a family of controllers coming under one umbrella named MCS-48™ family. Eventually Intel came out with its most fruitful design in the 8-bit microcontroller domain-the 8051 family and its derivatives. 8051 is the most popular and powerful 8-bit microcontroller ever built. It was developed in the 1980s and was put under the family MCS-51. Almost 75% of the microcontrollers used in the embedded domain were 8051 family based controllers during the 1980-90s. 8051 processor cores are used in more than 100 devices by more than 20 independent

manufacturers like Maxim, Philips, Atmel, etc. under the license from Intel. Due to the low cost, wide availability, memory efficient instruction set, mature development tools and Boolean processing (bit manipulation operation) capability, 8051 family derivative microcontrollers are much used in high-volume consumer electronic devices, entertainment industry and other gadgets where cost-cutting is essential.

Microprocessors	Microcontrollers
Microprocessors generally does not have RAM, ROM and I/O pins.	Microcontroller is 'all in one' processor, with RAM, I/O ports, all on the chip.
Microprocessors usually use its pins as a bus to interface to RAM, ROM, and peripheral devices. Hence, the controlling bus is expandable at the board level.	Controlling bus is internal and not available to the board designer.
Microprocessors are generally capable of being built into bigger general purpose applications.	Microcontrollers are usually used for more dedicated applications.
Microprocessors, generally do not have power saving system.	Microcontrollers have power saving system, like idle mode or power saving; mode so overall it uses less power.
The overall cost of systems made with Microprocessors is high, because of the high number of external components required.	Microcontrollers are made by using complementary metal oxide semiconductor technology; so they are far cheaper than Microprocessors.
Processing speed of general microprocessors is above 1 GHz; so it works much faster than Microcontrollers.	Processing speed of Microcontrollers is about 8 MHz to 50 MHz.
Microprocessors are based on von-Neumann model; where, program and data are stored in same memory module.	Microcontrollers are based on Harvard architecture; where, program memory and data memory are separate.

Digital Signal Processors (DSPs): *Digital Signal Processors* are powerful special purpose 8/ 16/ 32 bit microprocessors designed specifically to meet the computational demands and power constraints of today's embedded audio, video, and communications applications.

Digital signal processors are 2 to 3 times faster than the general purpose microprocessors in signal processing applications. This is because of the architectural difference between the two. DSPs implement

algorithms in hardware which speeds up the execution, whereas general purpose processors implement the algorithm in firmware and the speed of execution depends primarily on the clock for the processors.

In general, DSP can be viewed as a microchip designed for performing high speed computational operations for 'addition', 'subtraction', 'multiplication' and 'division'.

A typical digital signal processor incorporates the following four key units:

1. Program Memory: Memory for storing the program required by DSP to process the data.
2. Data Memory: Working memory for storing temporary variables and data/ signal to be processed.
3. Computational Engine: Performs the signal processing in accordance with the stored program memory. Computational Engine incorporates many specialized arithmetic units and each of them operates simultaneously to increase the execution speed. It also incorporates multiple hardware shifters for shifting operands and thereby saves execution time.
4. I/O Unit: Acts as an interface between the outside world and DSP. It is responsible for capturing signals to be processed and delivering the processed signals.

Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed.

Digital signal processing employs a large amount of real-time calculations. Sum of Products (SOP) calculation, Convolution, Fast Fourier Transform (FFT), Discrete Fourier Transform (DFT), etc, are some of the operations performed by digital signal processors.

Blackfin® processors from Analog Devices is an example of DSP which delivers breakthrough signal processing performance and power efficiency while also offering a full 32-bit RlSC MCU programming model.

RISC versus CISC Processors/ Controllers: The term RISC stands for Reduced Instruction Set Computing. As the name implies, all RISC processors/ controllers possess lesser number of instructions, typically in the range of 30 to 40.

CISC stands for Complex instruction Set Computing. From the definition itself it is clear that the instruction set is complex and instructions are high in number.

From a programmers point of view RISC processors are comfortable, since s/ he needs to learn only a few instructions, whereas for a CISC processor s/ he needs to learn more number of instructions and should understand the context of usage of each instruction.

Atmel AVR microcontroller is an example for a RISC processor and its instruction set contains only 32 instructions. The original version of 8051 microcontroller (e.g.AT 89C51) is a CISC controller and its instruction set contains 255 instructions.

Remember it is not the number of instructions that determines whether a processor/ controller is CISC or RISC. There are some other factors like pipelining features, instruction set type, etc., for determining the RISC/ CISC criteria. Some of the important criteria are listed below:

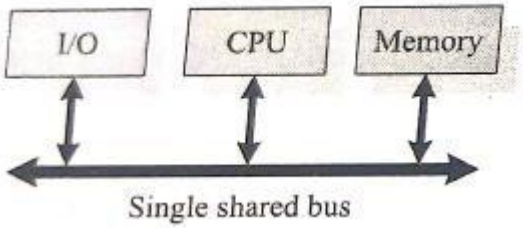
CISC	RISC
1. Complex instructions, taking multiple clock	1. Simple instructions, taking single clock
2. Emphasis on hardware, complexity is in the micro-program/processor	2. Emphasis on software, complexity is in the complier
3. Complex instructions, instructions executed by micro-program/processor	3. Reduced instructions, instructions executed by hardware
4. Variable format instructions, single register set and many instructions	4. Fixed format instructions, multiple register sets and few instructions
5. Many instructions and many addressing modes	5. Fixed instructions and few addressing modes
6. Conditional jump is usually based on status register bit	6. Conditional jump can be based on a bit anywhere in memory
7. Memory reference is embedded in many instructions	7. Memory reference is embedded in LOAD/STORE instructions

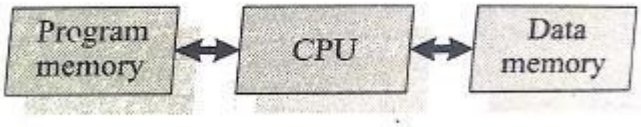
Harvard versus Von-Neumann Processor/ Controller Architecture: The terms Harvard and Von-Neumann refers to the processor architecture design.

Microprocessors/ Controllers based on the *Von-Neumann architecture* shares a single common bus for fetching both instructions and data. Program instructions and data are stored in a common main memory. Von-Neumann architecture based processors/ controllers first fetch an instruction and then fetch the data to support the instruction from code memory. The two separate fetches slows down the controller's operation. Von-Neumann architecture is also referred as *Princeton architecture*, since it was developed by the Princeton University.

Microprocessors/ Controllers based on the *Harvard architecture* will have separate data bus and instruction bus. This allows the data transfer and program fetching to occur simultaneously on both buses. With Harvard architecture, the data memory can be read and written while program memory is being accessed. These separated data memory and code memory buses allow one instruction to execute while the next instruction is fetched ("pre-fetching"). The pre-fetch theoretically allows much faster execution than Von-Neumann architecture.

The following Figure explains the Harvard and Von-Neumann architecture concept.



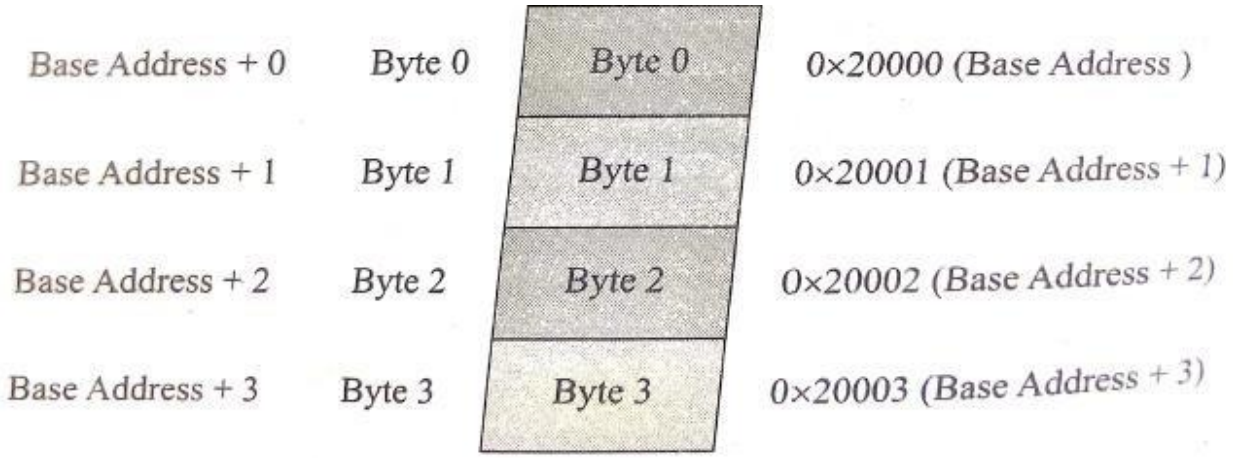


Von-Neumann Architecture	Harvard Architecture
1. Single shared bus for instruction and data fetching	1. Separate buses for instruction and data fetching
2. Chances for accidental corruption of program memory, as data memory and program memory are stored physically in the same chip	2. No chances for accidental corruption of program memory, as data memory and program memory are stored physically in different locations
3. Low performance compared to Harvard architecture; and comparatively cheaper	3. Easier to pipeline, so high performance can be achieved; and comparatively high cost
4. Allows self modifying codes – code/ instruction which modifies itself during execution	4. No memory alignment problems

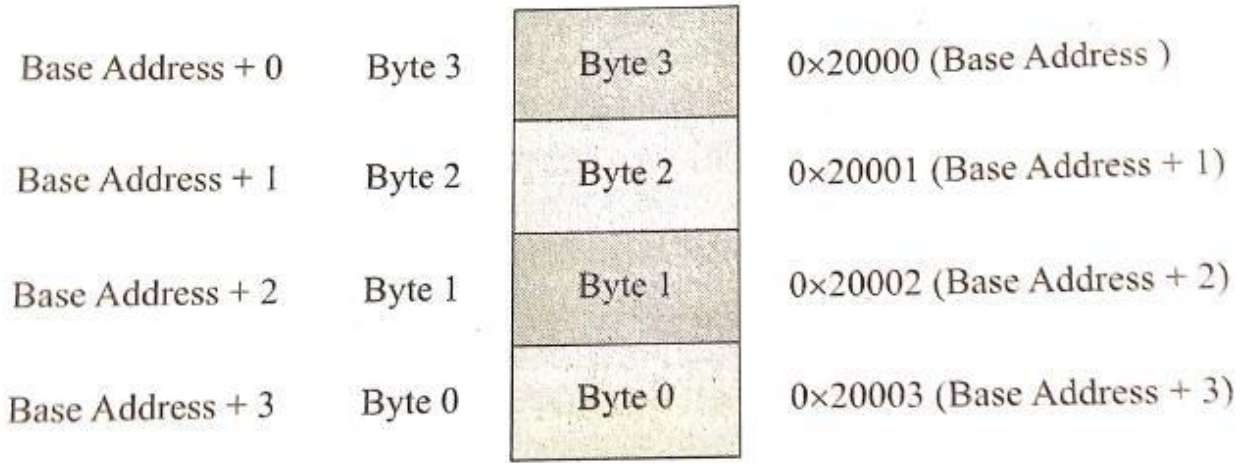
Big-Endian versus Little-Endian Processors/ Controllers: *Endianness* specifies the order in which the data is stored in the memory by processor operations in a multi-byte system (Processors whose word size is greater than one byte).

Suppose the word length is two byte; then data can be stored in memory in two different ways:

1. Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory – *Little-Endian*. E.g.: a 4 byte long integer Byte3 Byte2 Byte1 Byte0 will be stored in the memory as follows:

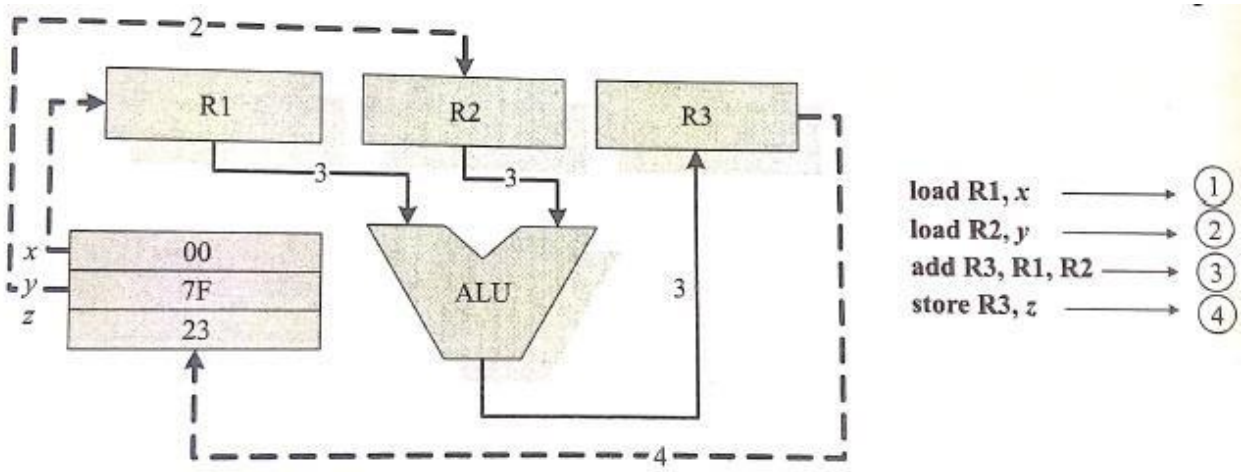


2. Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory – *Big-Endian*. E.g.: a 4 byte long integer Byte3 Byte2 Byte1 Byte0 will be stored in the memory as follows:



Load Store Operation and Instruction Pipelining: As mentioned earlier, the RlSC processor instruction set is orthogonal, meaning it operates on registers. The memory access related operations are performed by the 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 is illustrated with the following example:

Suppose *x*, *y* and *z* are memory locations and we want to add the contents of *x* and *y* and store the result in location *z*. Under the load store architecture the same is achieved with 4 instructions as shown in following Figure.



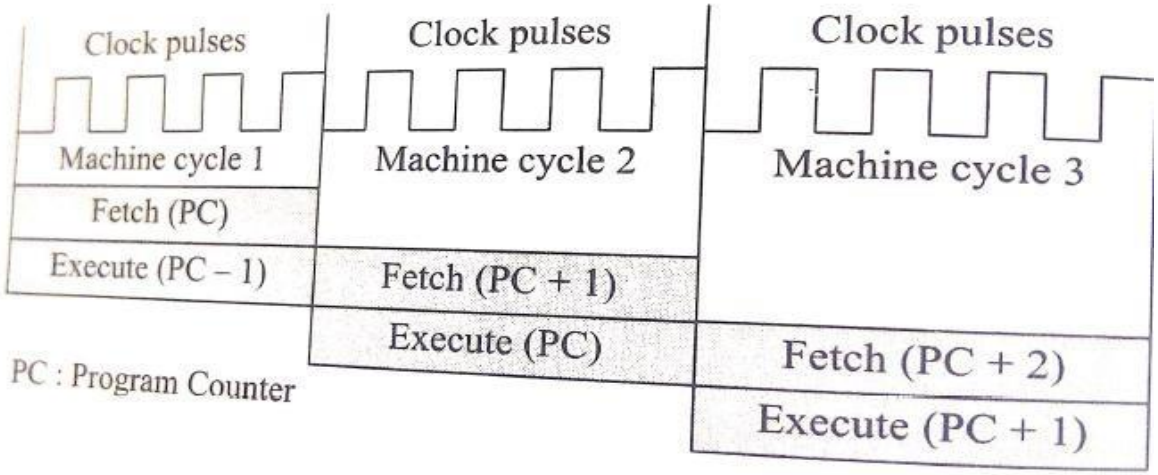
The first instruction *load R1, x* loads the register *R1* with the content of memory location *x*, the second instruction *load R2, y* loads the register *R2* with the content of memory location *y*. The instruction *add R3, R1, R2* adds the content of register *R1* and *R2* and store the result in register *R3*. The next instruction *store R3, z* stores the content of register *R3* in memory location *z*.

The conventional instruction execution by the processor follows the fetch-decode-execute sequence; where the 'fetch' part fetches the instruction from program memory or code memory, the '*decode*' part decodes the instruction to generate the necessary control signals and the '*execute*' stage reads the operands, perform ALU operations and stores the result.

In conventional program execution, the fetch and decode operations are performed in sequence. Whenever the current instruction is executing the program counter will be loaded with the address of the next instruction. In case of jump or branch instruction, the new location is known only after completion of the jump or branch instruction.

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

The following Figure illustrates the concept of Instruction pipelining for single stage pipelining.



Application Specific Integrated Circuits (ASICs):

Application Specific Integrated Circuit is a microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips. It integrates several functions into a single chip and there by reduces the system development cost. As a single chip, ASIC consumes a very small area in the total system.

ASICs can be pre-fabricated for a special application or it can be 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. Fabrication of ASICs requires a non-refundable initial investment (known as Non-Recurring Engineering Charges (NRE), a one-time expense) for the process technology and configuration expenses.

Features of ASICs:

1. NRE cost.
2. Less complex.
3. High Performance.
4. Low power consumption.

Drawbacks of ASICs:

1. Inflexible design.
2. Updates require a re-design.
3. Deployed systems cannot be upgraded.
4. Complex and expensive development tool.
5. Mistakes in product development are costly.

If Non-Recurring Engineering Charges (NRE) is borne by a third party and the Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is .referred as *Application Specific Standard Product* (*ASSP*).

ASICs	ASSPs
1. Microchip designed to perform specific application	1. If third party is ready to pay NRE cost and ASIC is made available into the market, the ASIC referred as ASSP
2. Most of the ASICs are proprietary product	2. Openly available in the market

General Purpose Processor (GPP) versus Application-Specific Instruction Set Processor (ASIP): A *General Purpose Processor* or *GPP* is a processor designed for general computational tasks. The processor running inside your laptop or desktop (Pentium 4/ AMD Athlon, etc.) is a typical example for general purpose processor. They are produced in large volumes, and hence, the per unit cost for a chip is low compared to ASIC or other specific ICs.

A typical general purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU). On the other hand, *Application Specific Instruction Set Processors* (*ASIPs*) are processors with architecture and instruction set optimized to specific-domain/ application requirements, like network processing, automotive, telecom, media applications, digital signal processing, control applications, etc. ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory.

Programmable Logic Devices:

Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.

Logic devices can be classified into two broad categories-*fixed* and *programmable*.

- As the name indicates, the circuits in a fixed logic device are permanent, they perform one function or set of functions-once manufactured, they cannot be changed.

- On the other hand, *Programmable Logic Devices* (*PLDs*) offer customers a wide range of logic capacity, features, speed, and voltage characteristics; and these devices can be re-configured to perform any number of functions at any time.

Advantages of PLDs: Programmable logic devices offer a number of advantages over fixed logic devices, including:

- PLDs offer customers much more flexibility during the design cycle because design iterations are simply a matter of changing the programming file, and results of design changes can be seen immediately in working parts.
- PLDs do not require long lead times for prototypes or production parts-the PLDs are already on a distributor's shelf and ready for shipment.
- PLDs do not require customers to pay for large NRE costs and purchase expensive mask sets-PLD suppliers incur those costs when they design their programmable devices.
- PLDs allow customers to order just the number of parts they need, when they need them, allowing them to control inventory.
- PLDs can be reprogrammed even after a piece of equipment is shipped to a customer.

CPLDs and FPGAs: The two major types of programmable logic devices are *Field Programmable Gate Arrays* (*FPGAs*) and *Complex Programmable Logic Devices* (*CPLDs*). Of the two, FPGAs offer the highest amount of logic density, the most features, and the highest performance. The largest FPGA now shipping part of the Xilinx Virtex™.

CPLDs	FPGAs
1. PLD is used for construction of CPLD	1. Logic blocks are used for construction of FPGA
2. CPLD is non-volatile & less costly	2. FPGA is volatile & costly
3. Delays are much more predictable in CPLDs	3. Prediction of delay is difficult in FPGA
4. Operating speed is low & is suitable for control circuit	4. Operating speed is high & is suitable for timing circuit
5. CPLD has less flexibility and design capacity	5. FPGA has more flexibility as well as design capacity
6. CPLD could work immediately after power up	6. FPGA could not work until the configuration is done
7. CPLDs are considered as 'coarse-grain' devices	7. FPGAs are considered as 'fine-grain' devices

FPGAs	ASICs
1. FPGA is a reprogrammable integrated circuit	1. ASIC is a unique type of integrated circuit meant for a specific application
2. FPGA is not efficient in terms of use of materials	2. ASIC wastes very little material, recurring cost is low
3. FPGA is better than ASIC when building low volume production circuits	3. Cost of ASIC is low only when it is produced in large qualtity
4. FPGA is alterable	4. Once created, ASIC can no longer be altered
5. FPGAs are useful for research and de3velopment activities. Prototype fabrication using FPGA is affordable and fast	5. ASICs are not suitable for research and development purpose, as they are reconfigurable

Commercial Off-the-Shelf Components:

A *Commercial Off-the-Shelf* (*COTS*) product is one which is used '*as-is*'. COTS products are designed in such a way to provide easy integration and interoperability with existing system components. The COTS component itself may be developed around a general purpose or domain specific processor or an Application Specific Integrated Circuit or a Programmable Logic Device.

Typical examples of COTS hardware unit are remote controlled toy car control units including the RF circuitry part, high performance, high frequency microwave electronics (2-200 GHz), high bandwidth analog-to-digital converters, devices and components for operation at very high temperatures, electro-optic IR imaging arrays, UV/IR detectors, etc.

The major advantage of using COTS is that they are readily available in the market, are cheap and a developer can cut down his/ her development time to a great extent. This in turn reduces the time to market your embedded systems.

The TCPIIP plug-in module available from various manufactures like 'WIZnet', 'Freescale', 'Dynalog', etc are very good examples of COTS product (following Figure).

Benefits of COTs-based System:

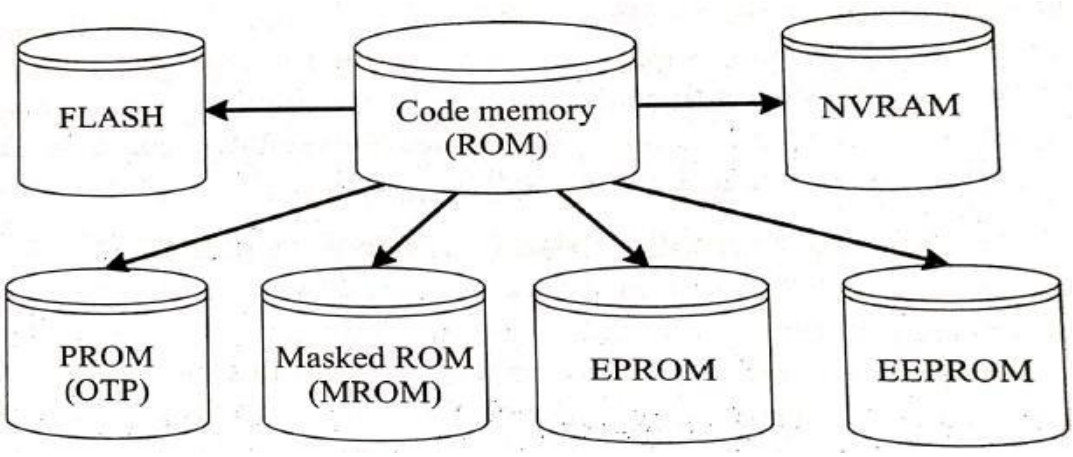
1. To reduce development cost 2. To reduce software life-cycle 3. To improve software development process 4. To reduce coding, debugging, unit testing, and code inspection.

MEMORY:

Memory is an important part of a processor/ controller based embedded systems. Some of the processors/ controllers contain built in memory and this memory is referred as *on-chip memory*. Others do not contain any memory inside the chip and requires external memory to be connected with the controller/processor to store the control algorithm. It is called *off-chip memory*. Also some working memory is required for holding data temporarily during certain operations.

Program Storage Memory (ROM):

The *program memory or code storage memory* of an embedded system stores the program instructions and it can be classified into different types as per the block diagram representation given in the following Figure.



The code memory retains its contents even after the power to it is turned off. It is generally known as *non-volatile storage memory*. Depending on the fabrication, erasing and programming techniques, they are classified into the following types:

1. **Masked Memory (MROM):** Masked ROM is a one-time programmable device. Masked ROM makes use of the hardwired technology for storing data. The device is factory programmed by masking and metallization process at the time of production itself, as per the data provided by the end user.
    - The primary advantage of this is low cost for high volume production. They are the least expensive type of solid state memory. 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.
    - Masked ROM is a good candidate for storing the embedded firmware for low cost embedded devices. Once the design is proven and the firmware requirements are tested

and frozen, the binary data (The firmware cross compiled/assembled to target processor specific machine code) corresponding to it can be given to the MROM fabricator.

- The limitation with MROM based firmware storage is the inability to modify the device firmware against firmware upgrades. Since the MROM is permanent in bit storage, it is not possible to alter the bit information.

2. **Programmable Read Only Memory (PROM)/ One Time Programmable Memory (OTP):** PROM is not pre-programmed by the manufacturer. The end user is responsible for programming these devices.
   - This memory has *nichrome* or *polysilicon* wires arranged in a matrix. These wires can be functionally viewed as fuses. It is programmed by a PROM programmer which selectively burns the fuses according to the bit pattern to be stored. Fuses which are not blown/ burned, represents logic "1"; whereas fuses which are blown/ burned represents a logic "0". The default state is logic "1".
   - OTP is widely used for commercial production of embedded systems whose proto-typed versions are proven and the code is finalized. It is a low cost solution for commercial production. OTPs cannot be reprogrammed.
   - Limitations: OTPs are not useful and worth for development purpose. During the development phase, the code is subject to continuous changes and using an OTP each time to load the code is not economical.

3. **Erasable Programmable Read Only Memory (EPROM):** EPROM gives the flexibility to reprogram the same chip.
   - EPROM stores the bit information by charging the floating gate of an FET. Bit information is stored by using an EPROM programmer, which applies high voltage to charge the floating gate.
   - EPROM contains a quartz crystal window for erasing the stored information. If the window is exposed to ultraviolet rays for a fixed duration, the entire memory will be erased.
   - Limitations: Even though the EPROM chip is flexible in terms of re-programmability, it needs to be taken out of the circuit board and put in a UV eraser device for 20 to 30 minutes. So it is a tedious and time-consuming process.

4. **Electrically Erasable Programmable Read Only Memory (EEPROM):** Electrically Erasable Programmable Read Only Memory indicates; the information contained in the EEPROM memory can be altered by using electrical signals at the register/Byte level. They can be erased and

reprogrammed in-circuit. These chips include a chip erase mode; and in this mode, they can be erased in a few milliseconds.

- It provides greater flexibility for system design.
- The only limitation is their capacity is limited (only few kilobytes) when compared with the standard ROM.

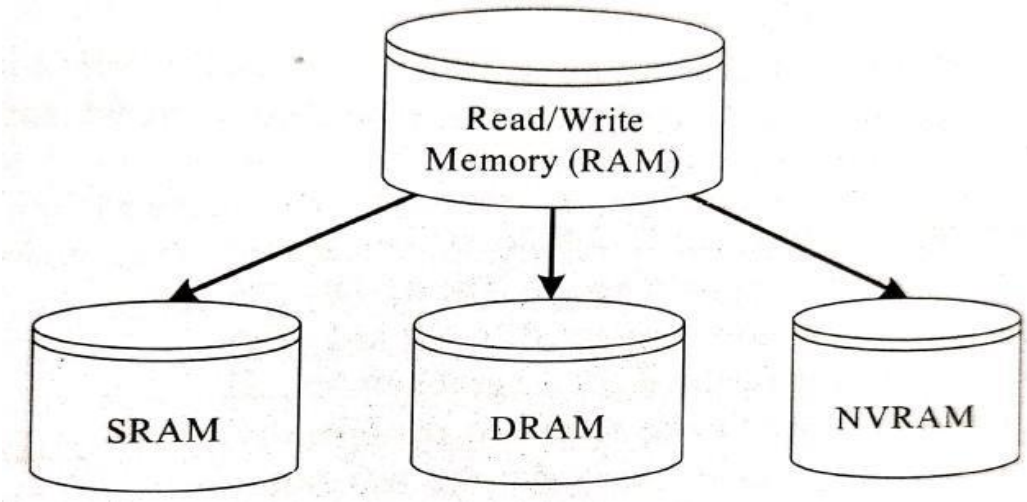
5. **FLASH:** FLASH is the latest ROM technology and is the most popular ROM technology used in today's embedded designs. FLASH memory is a variation of EEPROM technology. It combines the re-programmability of EEPROM and the high capability of standard ROMs.
   - FLASH memory is organized as sectors (blocks) or pages. FLASH memory stores information in an array of floating gate MOSFET transistors. The erasing of memory can be one 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.

6. **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 external power supply. The memory and battery are packed together in a single package.
   - The life span of NVRAM is expected to be around 10 years. DSJ644 from Maxim/ Dallas is an example of 32KB NVRAM.

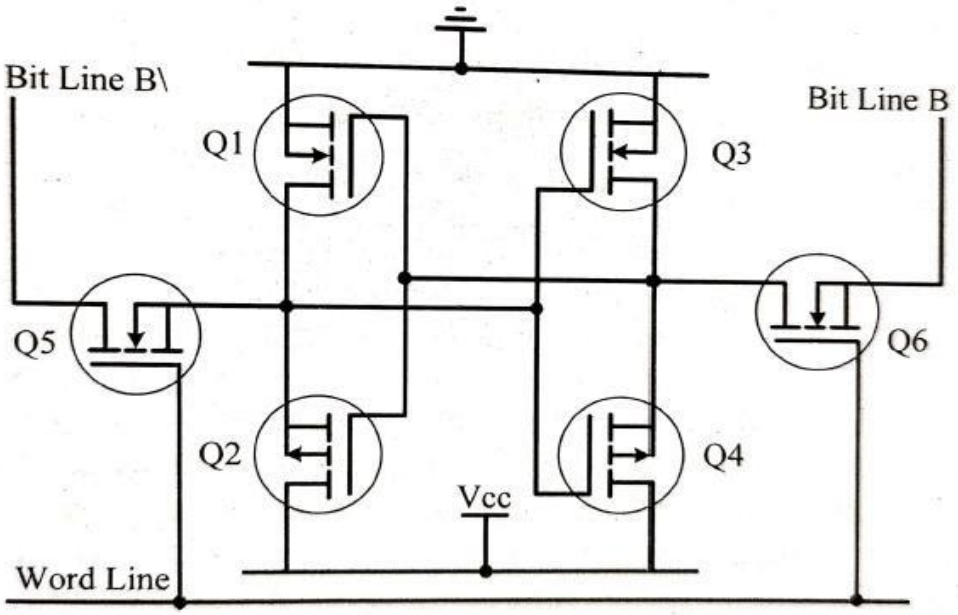
Read-Write Memory/ Random Access memory (RAM):

RAM is the data memory or working memory of the controller/ processor. Controller/ processor can read from it and write to it.

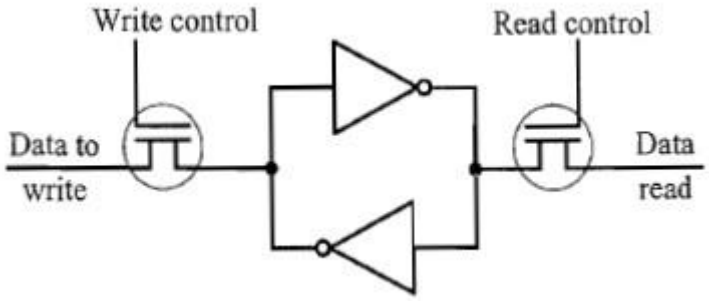
- RAM is volatile, meaning when the power is turned off, all the contents are destroyed.
- RAM is a direct access memory, meaning we can access the desired memory location directly without the need for traversing through the entire memory locations to reach the desired memory position (i.e. random access of memory location).
  - This 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. Magnetic tapes, CD ROMs, etc. are examples of sequential access memories.
- RAM generally falls into three categories: Static RAM (SRAM), dynamic RAM (DRAM) and non-volatile RAM (NVRAM).



1. **Static RAM (SRAM):** Static RAM stores data in the form of voltage. They are made up of flip-flops. Static RAM is the fastest form of RAM available.
    - In typical implementation, an SRAM cell (bit) is realized 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.
    - In simplest representation an SRAM cell can be visualized as shown in the following Figure:

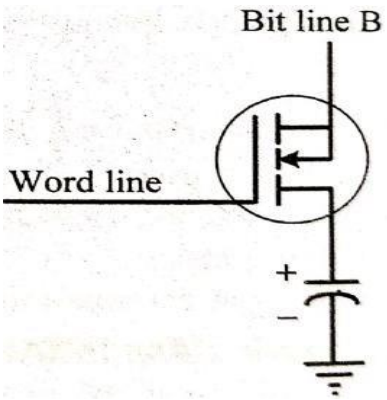


- This implementation in its simpler form can be visualized as two-cross coupled inverters with read/ write control through transistors. The four transistors in the middle form the cross-coupled inverters. This can be visualized as shown in the following Figure:



- From the SRAM implementation diagram, it is clear that access to the memory cell is controlled by the line Word Line, which controls the access transistors (MOSFETs) Q5 and Q6. The access transistors control the connection to bit lines B & B\.
  - In order 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 Word line high). This operation latches the bit written in the flip-flop.
  - 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. Since a minimum of six transistors are required to build a single memory cell, imagine how many memory cells we can fabricate on a silicon wafer.

2. **Dynamic RAM (DRAM):** Dynamic RAM stores data in the form of charge. They are made up of MOS transistor gates.
   - The advantages of DRAM are its high density and low cost compared to SRAM.
   - The disadvantage is that, since the information is stored as charge it gets leaked off with time; and to prevent this, they need to be refreshed periodically. Special circuits called DRAM controllers are used for the refreshing operation. The refresh operation is done periodically in milliseconds interval. The following Figure illustrates the typical implementation of a DRAM cell.



- The MOSFET acts as the gate for the incoming and outgoing data, whereas the capacitor acts as the bit storage unit.

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. More expensive	3. Less expensive
4. Fast in operation, typical access time is 10 ns	4. Slow in operation due to refresh requirement, typical access time is 60 ns; write operation is faster than read operation

3. **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 external power supply. The memory and battery are packed together in a single package.
    - The life span of NVRAM is expected to be around 10 years. DSJ644 from Maxim/ Dallas is an example of 32KB NVRAM.

Memory According to the Type of Interface:

The interface (connection) of memory with the processor/ controller can be of various types. It may be

- a parallel interface (the parallel data lines (DO-D7) for an 8 bit processor/ controller will be connected to DO-D7 of the memory) or
- a serial interface like I2C (Pronounced as I Square C - It is a 2 line serial interface) or
- a SPI (Serial Peripheral Interface, *2+n* line interface where *n* stands for the total number of SPI bus devices in the system)
- a single wire interconnection (like Dallas 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 that of a parallel interface memory is expressed in terms of kilobytes. Atmel Corporations AT24C512 is an example for 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 200 ns) compared to the execution from a random access memory (40 to 70 ns). From the timing parameters, it is obvious that 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 and video systems, there will be a configuration holding ROM called *Basic Input Output Configuration ROM* or simply *BIOS*.

- In personal computer system, BIOS stores the hardware configuration information like the address assigned for various serial ports and other non-plug 'n' play devices, etc. Usually it is read and the system is configured accordingly to it during system boot up and it is time consuming.
- Now, the manufactures included a RAM behind the logical layer of BIOS at its same address as a shadow to the BIOS; and the following steps happens:
    - During the boot up, BIOS is copied to the shadowed RAM
    - RAM is write protected
    - BIOS reading is disabled.
- Why both RAM and ROM are needed for holding the same data?
    - The answer is: RAM is volatile and it cannot hold the configuration data which is copied from the BIOS when the power supply is switched off. Only a ROM can hold it permanently. But for high system performance, it should be accessed from a RAM instead of accessing from a ROM.

Memory Selection for Embedded Systems:

Embedded systems require

- a *program memory* for holding the control algorithm or embedded OS and applications,
- *data memory* for holding variables and temporary data during task execution, and
- *memory* for holding non•volatile data (like configuration data, look up table, etc.) which are modifiable by the application.

- The memory requirement for an embedded system in terms of RAM and ROM (EEPROM/FLASH/NVRAM) is solely dependent on the type of the embedded system and the applications for which it is designed.
- There is no hard and fast rule for calculating the memory requirements. Lot of factors need to be considered when selecting the type and size of memory for embedded system.
    - For example, if the embedded system is designed using SoC or a microcontroller with on-chip RAM and ROM (FLASH/EEPROM), depending on the application need 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 (SoC or microcontroller with on•chip memory) used for the design, take a decision on whether the on-chip memory is sufficient or external memory is required.

- Let's consider a simple electronic toy design as an example. As the complexity of requirements are less and data memory requirement are minimal, we can think of a microcontroller with a few bytes of internal RAM, a few bytes or kilobytes (depending on the number of tasks and the complexity of tasks) of FLASH memory and a few bytes of EEPROM (if required) for designing the system. Hence there is no need for externa1 memory at all. A PIC microcontroller device which satisfies the I/O and memory requirements can be used in this case.

- If the embedded design is based on an RTOS, the RTOS requires certain amount of RAM for its execution and ROM for storing the RTOS image. Normally the binary code for RTOS kernel containing all the services is stored in a non-volatile memory (like FLASH) as either compressed or non-compressed data. During boot-up of the device, the RTOS files are copied from the program storage memory, decompressed if required and then loaded to the RAM for execution. The supplier of the RTOS usually gives a rough estimate on the run time RAM requirements and program memory requirements for the RTOS.
- On a safer side, always add a buffer value to the total estimated RAM and ROM size requirements.
  - A smart phone device with Windows mobile operating system is a typical example for embedded device with OS. Say 64MB RAM and 128MB ROM are the minimum requirements for running the Windows mobile device; indeed you need extra RAM and ROM for running user applications. So while building the system, count the memory for that also and arrive at a value which is always at the safer side, so that you won't end up in a situation where you don't have sufficient memory to install and run user applications.
- There are two parameters for representing a memory –
  - *Size of the memory chip:* There is no option to get a memory chip with the exact required number of bytes. Memory chips come in standard sizes, like 512bytes, 1024bytes (1 kilobyte), 2048bytes (2 kilobytes), 4Kb, 8Kb, 16Kb, 32Kb, 64Kb, 128Kb, 256Kb, 512Kb, 1024Kb (1 megabytes), etc.
    - Suppose your embedded application requires only 750 bytes of RAM, you don't have the option of getting a memory chip with size 750 bytes; the only option left with is to choose the memory chip with a size closer to the size needed. Hence, 1024 bytes is the least possible option.
    - Address range supported to the processor: A processor/ controller with 16-bit address bus can addressed 2^{16} = 65536 bytes = 64Kb. Hence, it is meaningless to select a 128Kb memory chip for a processor with 16-bit wide address bus.

        - Also, the entire memory range supported by the processor/ controller may not be available to the memory chip alone. It may be shared between I/O, other ICs and memory.
    - *Word size of the memory:* The word size refers to the number of memory bits that can be read/write together at a time. 4, 8, 12, 16, 24, 32 etc., are the word sizes supported by memory chips. Ensure that the word size supported by the memory chip matches with the data bus width of the processor/ controller.
- FLASH memory is the popular choice for ROM (program storage memory) in embedded applications. It is a powerful and cost-effective solid-state storage technology for mobile electronics devices and other consumer applications.
- FLASH memory comes in two major variants, namely, NAND and NOR FLASH.
    - NAND FLASH is a high-density low cost non-volatile storage memory; on the other hand, NOR FLASH is less dense and slightly expensive. But NOR FLASH 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 as in the case of conventional execution method.
- The EEPROM data storage memory is available as either serial interface or parallel interface chip. If 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 EEPR0M chip. The serial EEPROM saves the address space of the total system. The memory capacity of the serial EEPROM is usually expressed in bits or Kilobits: 512 bits, 1Kbits, 2Kbits, 4Kbits, etc. are examples for serial EEPROM memory representation.

SENSORS & ACTUATORS:

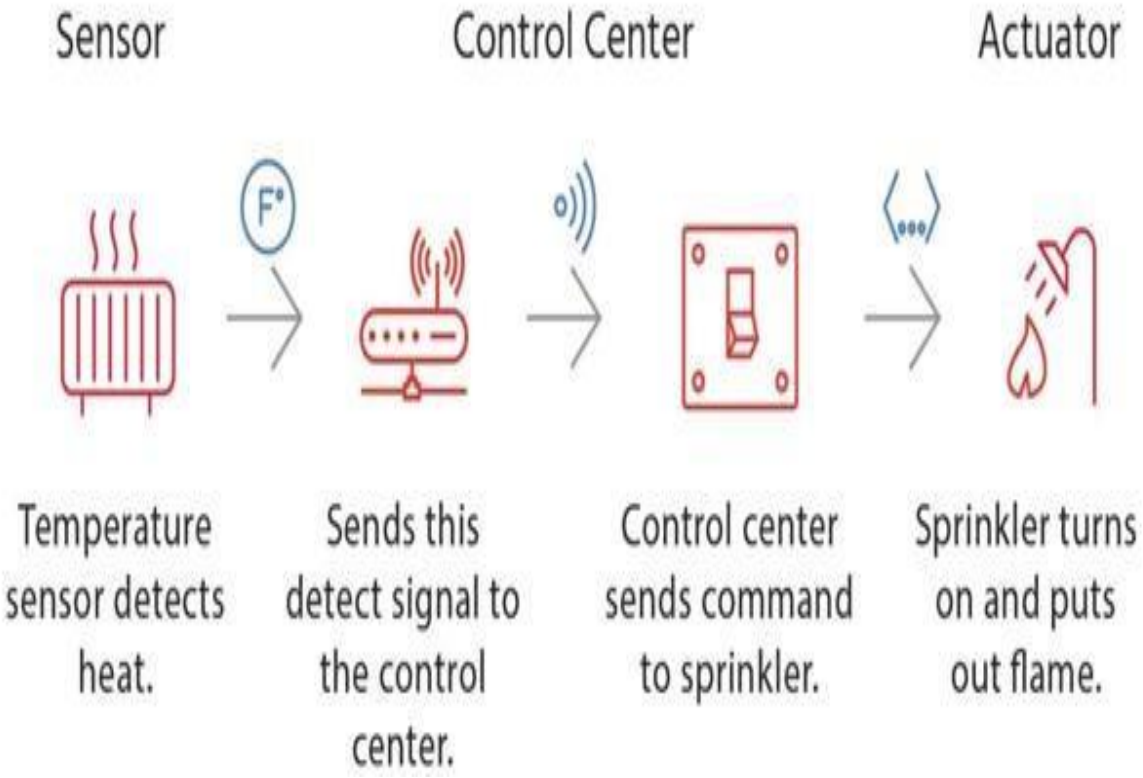
An embedded system is in constant interaction with the Real world, and the controlling/ monitoring functions, executed by the embedded system is achieved in accordance with the changes happening to the Real world. The changes in system environment or variables are detected by the *sensors* connected to the input port of the embedded system.

- A *sensor* is a transducer device that converts energy from one form to another, for any measurement or control purpose.
    - Sensor which counts steps for pedometer functionality is an Accelerometer sensor.
    - Sensor used in smart watch devices to measure the high intensity is an Ambient Light Sensor (ALS).

If the embedded system is designed for any controlling purpose, the system will produce some changes in the controlling variable to bring the controlled variable to the desired value. It is achieved through an *actuator* connected to the output port of the embedded system.

- *Actuator* is a form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device.
    - Smart watches use Ambient Light Sensor to detect the surrounding light intensity and uses an electrical/ electronic actuator circuit to adjust the screen brightness.

The following Figure shows the sensor to actuator flow:



If the embedded system is designed for monitoring purpose only, then there is no need for including an actuator in the system. For example, take the case of an ECG machine. It is designed to monitor the heart beat status of a patient and it cannot impose a control over the patient's heart beat and its order. The sensors used here are the different electrode sets connected to the body of the patient. The variations are captured and presented to the user (may be a doctor) through a visual display or some printed chart.

Sensors	**Actuators**
1. Sensor is an input device	1. Actuator is an output device
2. Convert a physical parameter to an electrical Output	2. Convert an electrical signal to a physical output
3. A device that detects events or changes in the environment and send the information to another electronic device	3. A component of a machine that is responsible for moving and controlling mechanisms
4. Sensor help to monitor the changes in the Environment	4. Actuator helps to control the environment or physical changes

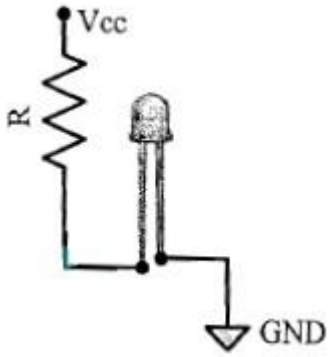
The I/O Subsystem:

The *I/O subsystem* of the embedded system facilitates the interaction of the embedded system with the external world. As mentioned earlier the interaction happens through the sensors and actuators connected to the input and output ports respectively of the embedded system.

Light Emitting Diode (LED): LED is an important output device for visual indication in any embedded system. LED can be used as an indicator for the status of various signals or situations.

- Typical examples are indicating the presence of power conditions like 'Device ON', 'Battery Low' or 'Charging of Battery' for a battery operated handheld embedded devices.

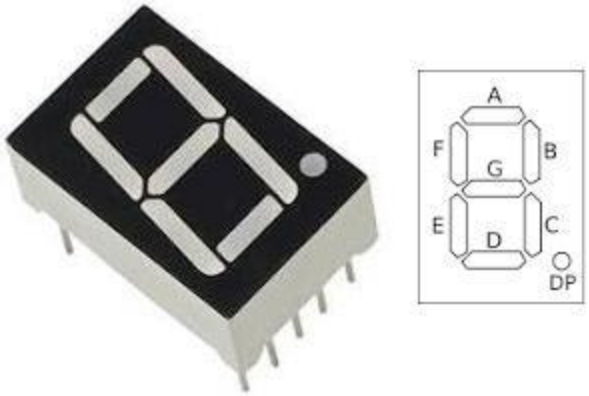
Light Emitting Diode is a p-n junction diode and it contains an anode and a cathode. For proper functioning of the LED, the anode of it should be connected to +ve terminal of the supply voltage and cathode to the –ve terminal of supply voltage. A resister is used in series between the power supply and the LED to limit the current through the LED. The ideal LED interfacing circuit is shown in the following Figure.



LEDs can be interfaced to the port pin of a processor/ controller in two 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 of processor/ controller and the anode to the supply voltage through a current limiting resistor. LED is turned on when the port pin is at logic Low (Logic '0').

7-Segment LED Display: The 7-segment LED display is an output device used for displaying alpha-numeric characters. It contains 8 light-emitting diode (LED) segments arranged in a special form. Out of the 8 LED segments, 7 are used for displaying alpha-numeric characters and 1 is used for representing ‘decimal point'. The following Figure explains the arrangement of LED segments in 7-segment LED display.

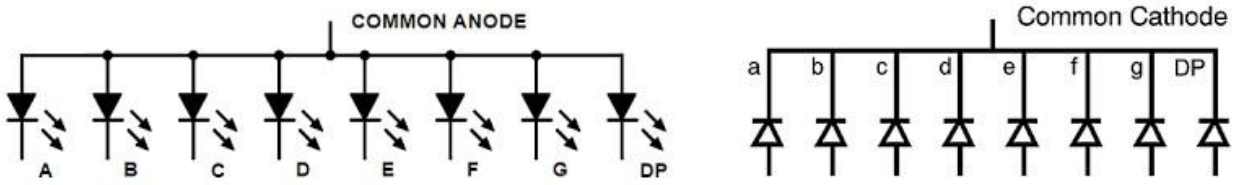


The LED segments are named A to G and the 'decimal point LED segment is named as DP. For displaying the number 4, the segments F, G, B and C are lit. For displaying 3, the segments A, B, C, D, G are lit. All these 8 LED segments need to be connected to one port of the processor/ controller for displaying alpha-numeric digits.

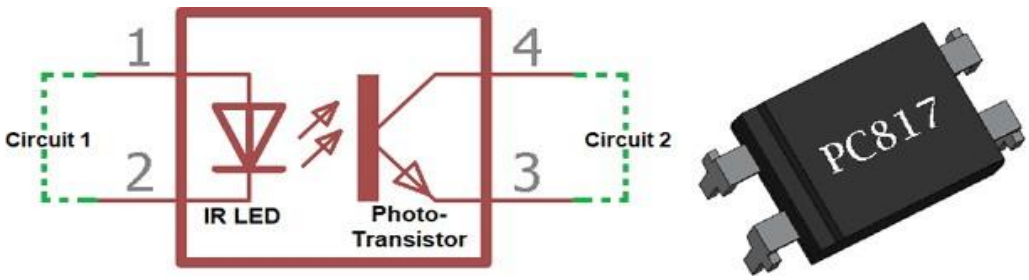
The 7-segment LED displays are available in two different configurations, namely; Common Anode and Common Cathode.

- In *common anode configuration*, the anodes of the 8 segments are connected commonly
- In *common cathode configuration*, the 8 LED segments share a common cathode line.

The following Figure illustrates the Common Anode and Cathode configurations.

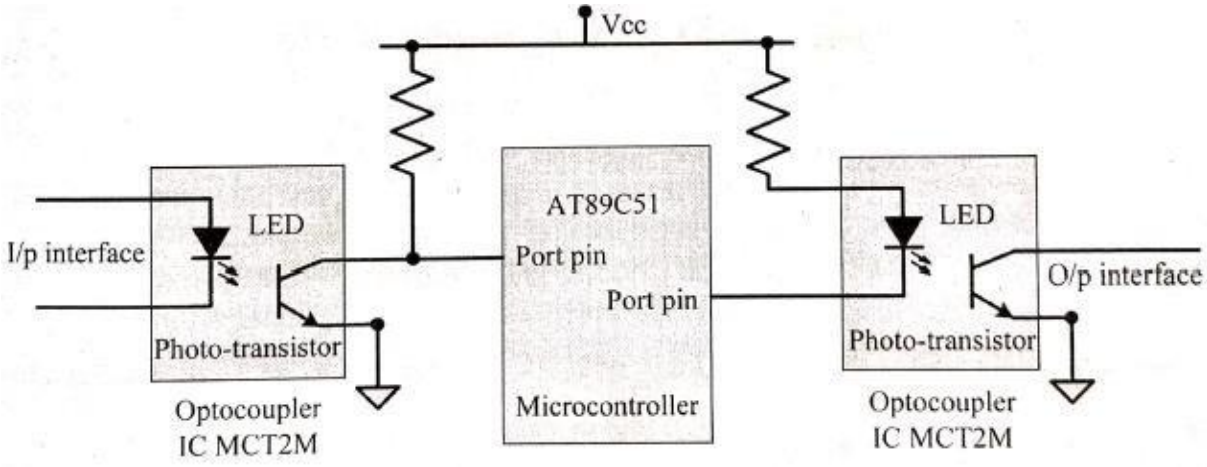


Optocoupler: *Optocoupler* is a solid state device to isolate two parts of a circuit. Optocoupler combines an LED and a photo-transistor in a single housing (package). The following Figure illustrates the functioning of an optocoupler device.



In electronic circuits, an optocoupler 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.

The following Figure illustrates the usage of optocoupler in input circuit and output circuit of an embedded system with a microcontroller as the system core.



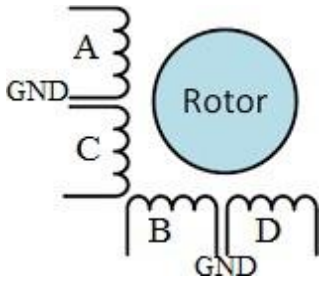
Optocoupler is available as ICs from different semiconductor manufacturers. The MCT2M IC from Fairchild semiconductor is an example for optocoupler IC.

Stepper Motor: A *stepper motor* is an electro-mechanical device which generates discrete displacement (motion) in response, to de electrical signals. It differs from the normal DC motor in its operation. The DC motor produces continuous rotation on applying DC voltage, whereas a stepper motor produces discrete rotation in response to the DC voltage applied to it.

Stepper motors are widely used in industrial embedded applications, consumer electronic products and robotics control systems. The paper feed mechanism of a printer/ fax makes use of stepper motors for its functioning.

Based on the coil winding arrangements, a two-phase stepper motor is classified into two. They are:

1. **Unipolar:** A unipolar stepper motor contains two windings per phase. The direction of rotation (clockwise or anticlockwise) of a stepper motor is controlled by changing the direction of current flow. Current in one direction flows through one coil and in the opposite direction flows through the other coil. It is easy to shift the direction of rotation by just switching the terminals to which the coils are connected. The following Figure illustrates the working of a two-phase unipolar stepper motor.



The coils are represented as A, B, C and D. Coils A and C carry current in opposite directions for phase 1 (only one of them will be carrying current at a time). Similarly, B and D carry current in opposite directions for phase 2 (only one of them will be carrying current at a time).

2. **Bipolar:** A bipolar stepper motor contains single winding per phase. For reversing the motor rotation the current flow through the windings is reversed dynamically. It requires complex circuitry for current flow reversal.

The stepping of stepper motor can be implemented in different ways by changing the sequence of activation of the stator windings. The different stepping modes supported by stepper motor are explained below:

Full Step: In the full step mode both the phases are energized simultaneously. The coils A, B, C and D are energized in the order, as shown in the following Table.

Wave Step: In the wave step mode, only one phase is energized at a time and each coils of the phase is energized alternatively. The A, B, C and D are energized in the order, as shown in the following Table.

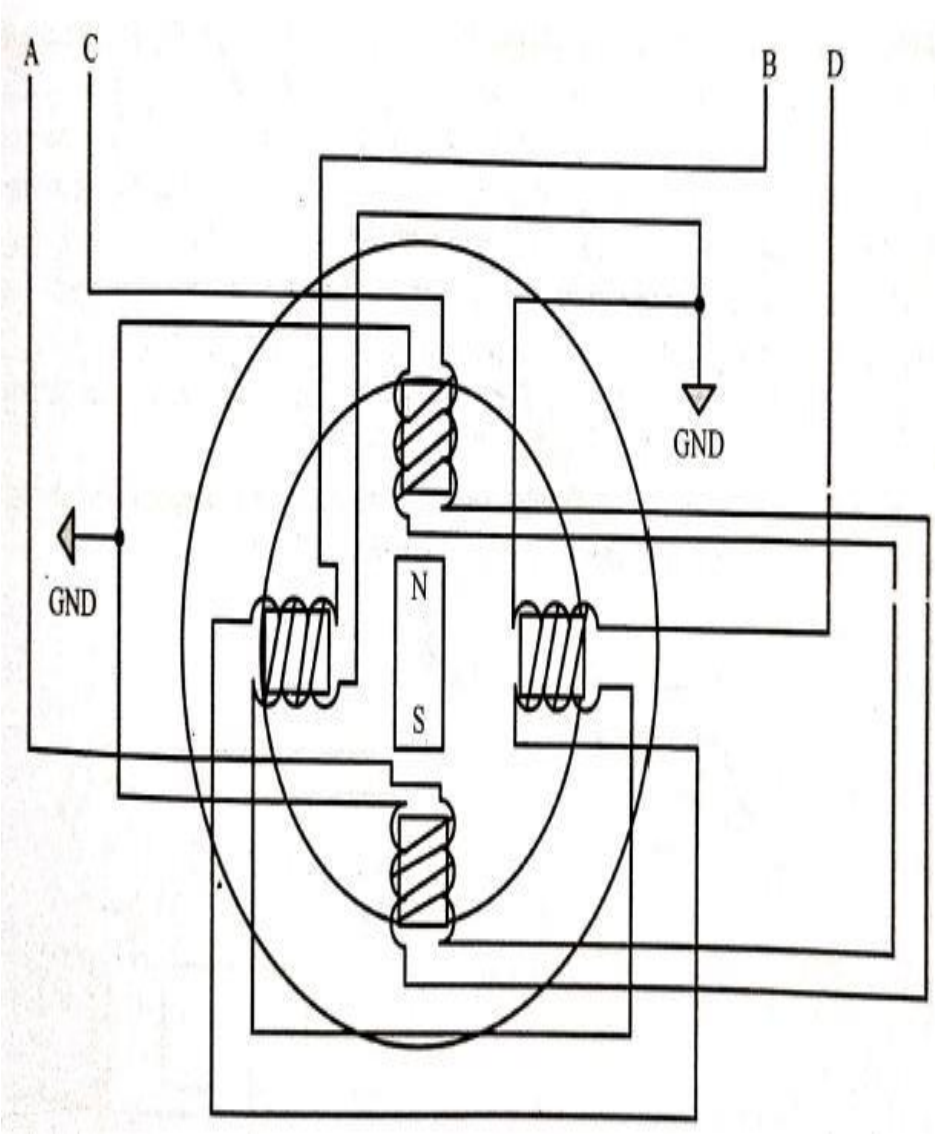
Step	Full Step				Wave Step			
	Coil A	Coil B	Coil C	Coil D	Coil A	Coil B	Coil C	Coil D
1	H	H	L	L	H	L	L	L
2	L	H	H	L	L	H	L	L
3	L	L	H	H	L	L	H	L
4	H	L	L	H	L	L	L	H

Half Step: It uses the combination of wave and full step. It has the highest torque and stability. The coil energizing sequence for half step is given in the Table below.

Step	Coil A	Coil B	Coil C	Coil D
1	H	L	L	L
2	H	H	L	L
3	L	H	L	L
4	L	H	H	L
5	L	L	H	L
6	L	L	H	H
7	L	L	L	H
8	H	L	L	H

The rotation of the stepper motor can be reversed by reversing the order in which the coil is energized.

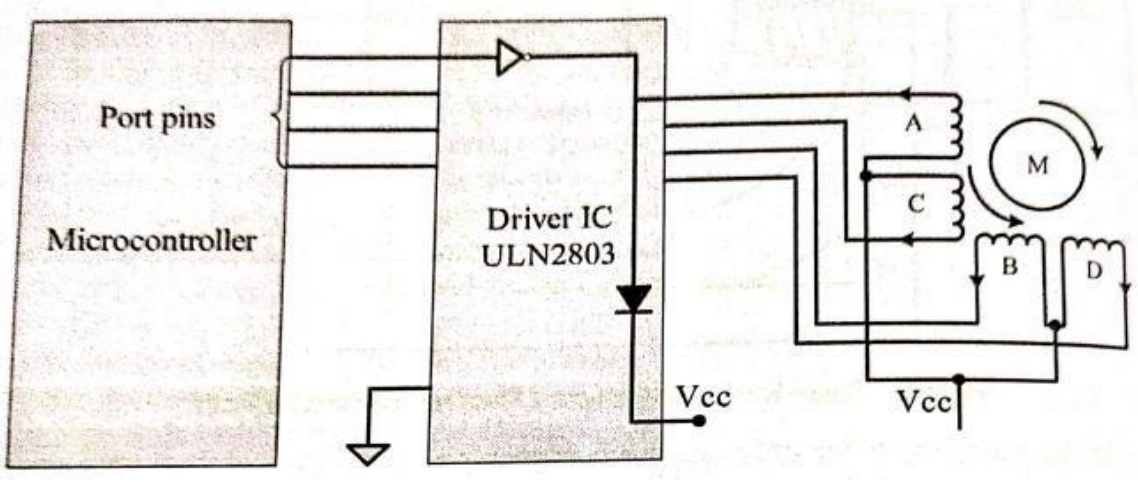
The following Figure shows the stator winding details of Stepper motor:



Two-phase unipolar stepper motors are the popular choice for embedded applications. The current requirement for stepper motor is little high and hence the port pins of a microcontroller/ processor may not be able to drive the directly. Also the supply voltage required to operate stepper motor varies normally in the range 5V to 24V. Depending on the current and voltage requirements, special driving circuits are required to interface the stepper motor with microcontroller/ processors.

ULN2803 is an octal peripheral driver array available from Texas Instruments and ST microelectronics for driving a 5V stepper motor. Simple driving circuit can also be built using transistors.

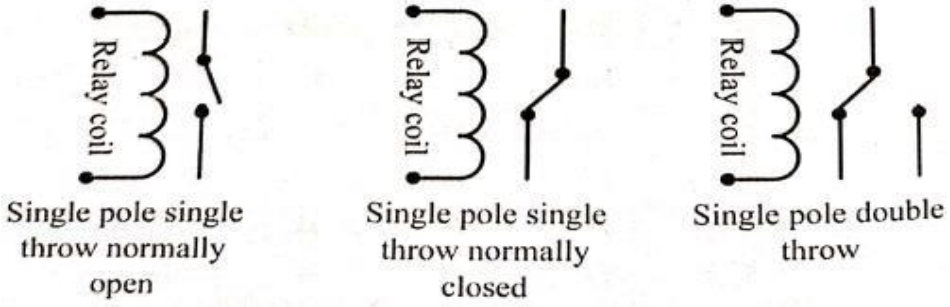
The following circuit diagram illustrates the interfacing of a stepper motor through a driver circuit connected to the port pins of a microcontroller/ processor.


Port pins
Microcontroller
Driver IC
ULN2803
A
C
B
D
M
Vcc
Vcc

Relay: Relay is an electro-mechanical device. In embedded application, the 'Relay' unit acts as dynamic path selectors for signals and power. The 'Relay' unit contains a relay coil made up of insulated wire on a metal core and a metal armature with one or more contacts.

'Relay' works on electromagnetic principle. When a voltage is applied to the relay coil, current flows through the coil, which in turn 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.

'Relays' are available in different configurations. The following Figure illustrates the widely used relay configurations for embedded applications.

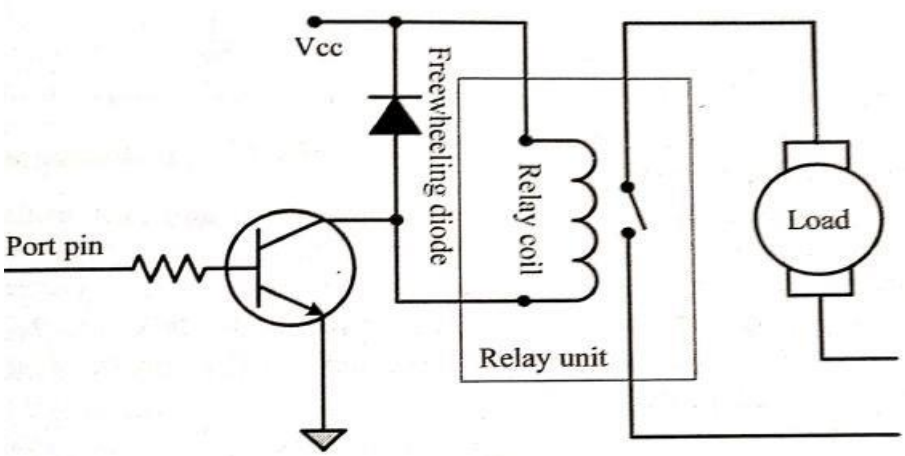


The Single Pole Single Throw configuration has only one path for information flow. The path is either open or dosed in normal condition.

- For normally open Single Pole Single Throw relay, the circuit is normally open and it becomes closed when the relay is energized.
- For normally closed Single Pole Single Throw configuration, the circuit is normally closed and it becomes open when the relay is energized.

For Single Pole Double Throw Relay, there are two paths for information flow and they are selected by energizing or de-energizing the relay.

The Relay is normally controlled using a relay driver circuit connected to the port pin of the processor/ controller. A transistor is used for building the relay driver circuit. The following Figure illustrates the same.



A free-wheeling diode is used for free-wheeling the voltage produced in the opposite direction when the relay coil is de-energized. The freewheeling diode is essential for protecting the relay and the transistor. .

Piezo Buzzer: *Piezo buzzer* is a piezoelectric device for generating audio indications in embedded application.

- A piezoelectric buzzer contains a piezoelectric diaphragm which produces audible sound in response to the voltage applied to it.

Piezoelectric buzzers are available in two types. 'Self• driving' and 'External driving'.

- The '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 piezoelectric buzzer.

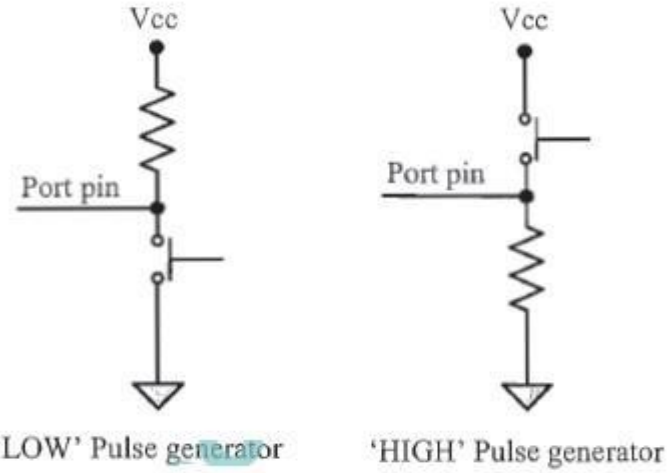
A piezo buzzer can be directly interfaced to the port pin of the processor/ control. Depending on the driving current requirements, the piezo buzzer can also be interfaced using a transistor based driver circuit as in the case of a 'Relay'.

Push Button Switch: *Push button switch* is an input device. Push button switch comes in two configurations, namely 'Push to Make' and 'Push to Break'.

- In the 'Push to Make' configuration, the switch is normally in the open state and it makes a circuit contact when it is pushed or pressed.
- In the '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' (for Push to Make type) or 'open' (For Push to Break type) state as long as it is kept in the pushed state and it breaks/ makes the circuit connection when it is released.
- Push button is used for generating a momentary pulse. In embedded application push button is generally used as reset and start switch and pulse generator. The Push button is normally connected to the port pin of the host processor/ controller.

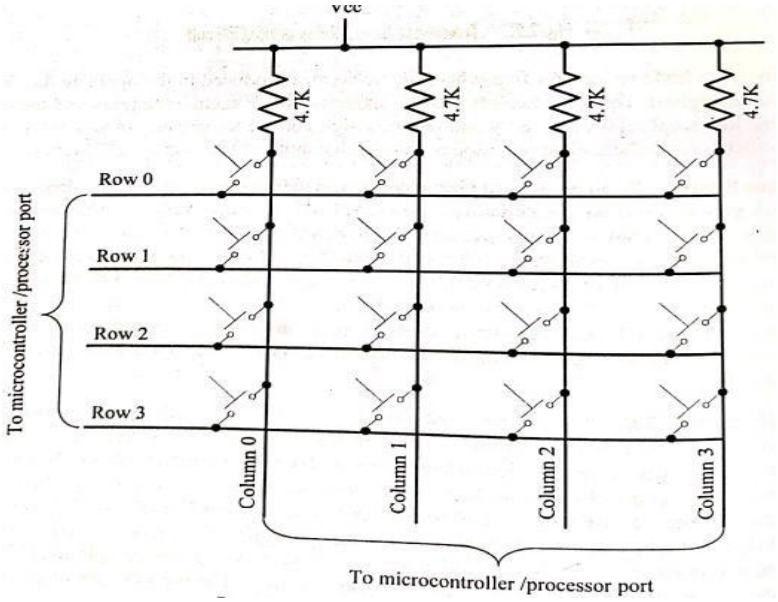
Depending on the way in which the push button interfaced to the controller, it can generate either a 'HIGH' pulse or a 'LOW' pulse.

The following Figure Illustrates how the push button can be used for generating 'LOW' and 'HIGH' pulses.



Keyboard: *Keyboard* is an input device 'HIGH' Pulse generator for user interfacing.

- If the number of keys required is very limited, push button switches can be used and they can be directly interfaced to the port pins for reading.
- However, there may be situations demanding a large number of keys for user input (e.g. PDA device with alpha-numeric keypad for user data entry).
    - In such situations it may not be possible to interface each keys to a port pin due to the limitation in the number of general purpose port pins available for the processor/ controller in use and moreover it is wastage of port pins.
    - Matrix keyboard is an optimum solution for handling large key requirement. It greatly reduces the number of interface connections.
- For example, for interfacing 16 keys, in the direct interfacing technique, 16 port pins are required, whereas in the matrix keyboard only 8 lines are required. The 16 keys are arranged in a 4 column x 4 Row matrix. The following Figure illustrates the connection o keys in a matrix keyboard.



In a matrix keyboard, the keys are arranged in matrix fashion. For detecting a key press, the keyboard uses the scanning technique, where each row of the matrix is pulled low and the columns are read. After reading the status of each columns corresponding to a row, the row is pulled high and the next row is pulled low and the status of the columns are read.

This process is repeated until the scanning for all rows are completed. When a row is pulled low and if a key connected to the row is pressed, reading the column to which the key is connected will give logic 0. Since keys are mechanical devices, proper key de-bouncing technique should be applied.

Programmable Peripheral Interface (PPI): *Programmable Peripheral Interface* devices are used for extending the I/O capabilities of processors/ controllers. Most of the processors/ controllers provide very limited number of I/O and data ports and at times it may require more number of I/O ports than the one supported by the controller/ processor.

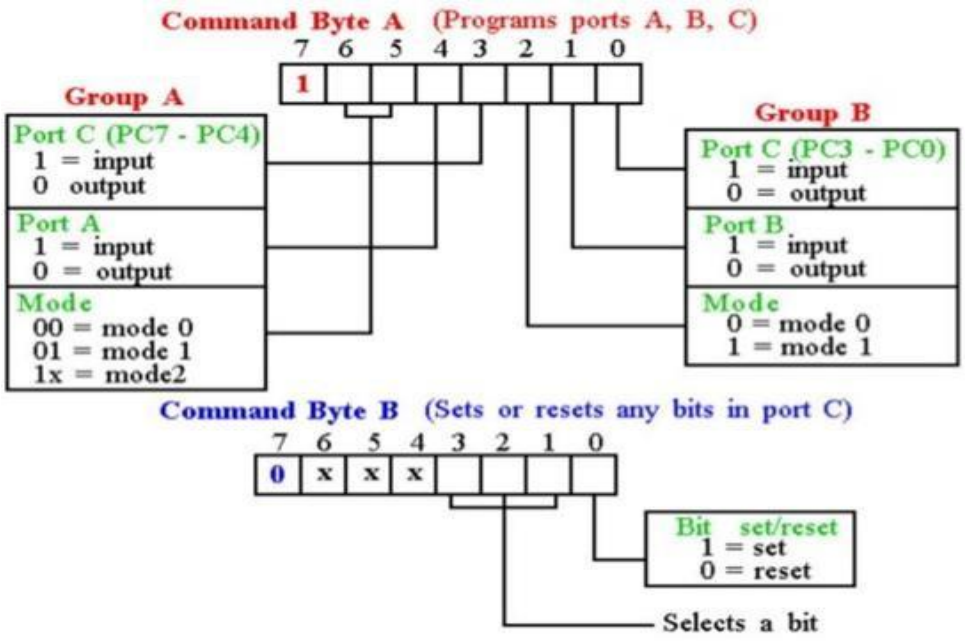
A programmable peripheral interface device expands the I/O capabilities of the processor/ controller. 8255A is a popular PPI device for 8-bit processors/ controllers.

8255A supports 24 I/O pins, and these I/O pins can be grouped as either three 8-bit parallel ports (Port A, Port B and Port C) or two 8-bit parallel ports (Port A and Port B) with Port C in any one of the following configurations:

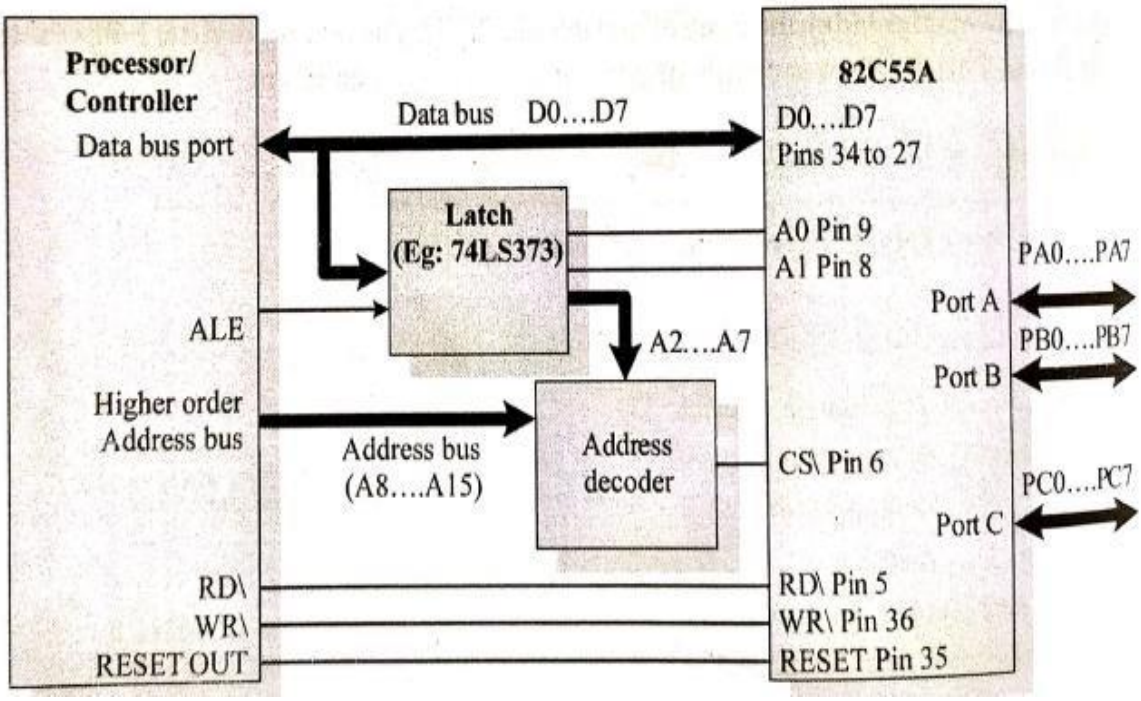
(1) As 8 individual I/O pins

(2) Two 4-bit ports; namely Port C_{UPPER} (Cu) and Port C_{LOWER} (CL).

This is configured by manipulating the control register of 8255A. The *control register* holds the configuration for Port A, Port B and Port C. The bit details of control register is given in the following Figure:



The following Figure illustrates the generic interfacing of a 8255A device with an 8-bit processor/ controller with 16-bit address bus (Lower order Address bus is multiplexed with data bus).



COMMUNICATION INTERFACE:

Communication Interface is essential for communicating with various subsystems of the embedded system and with the external world. For an embedded product, the communication interface can be viewed in two different perspectives –

- Device/ Board level communication interface (Onboard 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 components within an embedded product is referred as *Device/ Board Level Communication Interface* (*Onboard Communication Interface*).
  - Serial interfaces like I2C, SPI, DART, 1-Wire, etc., and parallel bus interface are examples of 'Onboard Communication Interface'.
- Product level communication interface (External Communication Interface)
  - Some embedded systems are self-contained units and they don't require any interaction and data transfer with other sub-systems or external world. On the other hand, certain embedded systems may be a part of a large distributed system and they require interaction and data transfer between various devices and sub-modules. The '*Product level communication interface*' (*External Communication Interface*) is responsible for data transfer between the embedded system and other devices or modules.

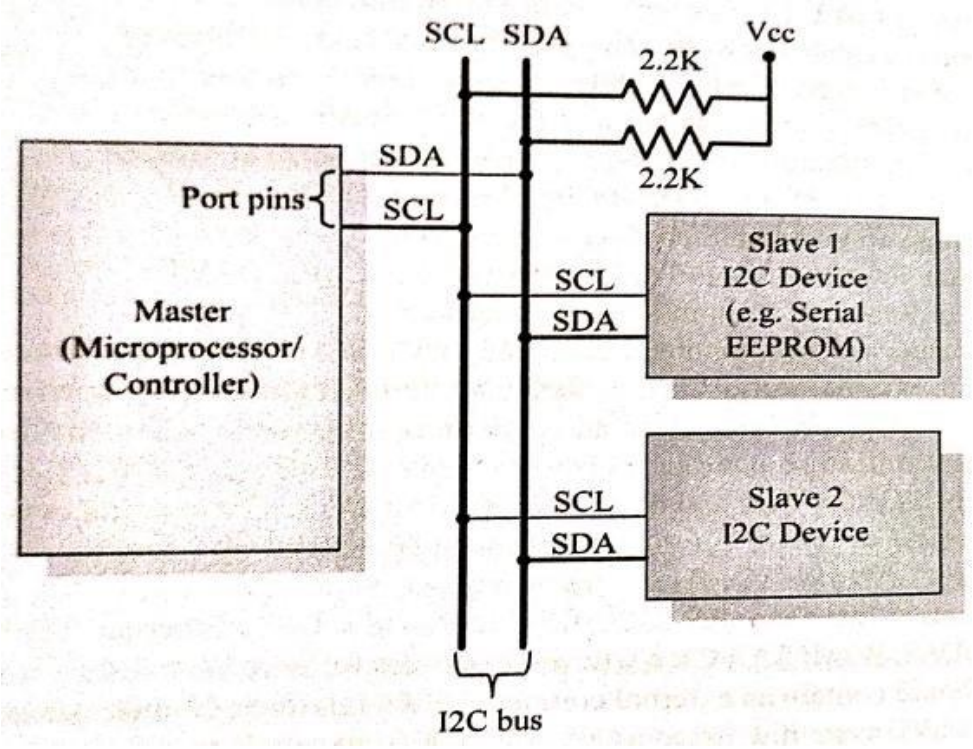
- The external communication interface can be either a wired medium or a wireless media and it can be a serial or a. parallel interface.
- Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS/ 3G/ 4GLTE, etc. are examples for wireless communication interface.
- RS-232C/ RS-422/ RS-485, USB, Ethernet IEEE 1394 port, Parallel port, CF-II interface, SDIO, PCMCIA/ PCIex, etc., are examples for wired interfaces.

Onboard Communication Interfaces:

Onboard Communication Interface refers to the different communication channels/ buses for interconnecting the various integrated circuits and other peripherals within the embedded system.

Inter Integrated Circuit (I2C) Bus: The *Inter Integrated Circuit Bus* (I2C-Pronounced 'I square C') is a synchronous bidirectional half duplex (one-directional communication at a given point of time) two wire serial interface bus.

- The concept of I2C bus was developed by 'Philips Semiconductors' in the early 1980s. The original intention of I2C was to provide an easy way of connection between a microprocessor/ microcontroller system and the peripheral chips in television sets.
- The I2C bus comprise of two bus lines, namely; *Serial Clock-SCL* and *Serial Data-SDA*.
  - *SCL* line is responsible for generating synchronization clock pulses.
  - *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.
- Devices connected to the I2C bus can act 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 synchronization clock pulses.
  - *'Slave' devices* wait for the commands from the master and respond upon receiving the commands.
- 'Master' and 'Slave' devices can act as either transmitter or receiver; regardless whether a master is acting as transmitter or receiver, the synchronization clock signal is generated by the 'Master' device only.
- I2C supports multi-masters on the same bus.
- The following Figure shows bus interface diagram, which illustrates the connection of master and slave devices on the I2C bus.



The address to various I2C devices in an embedded device is assigned and hardwired at the time of designing the embedded hardware.

The sequence of operations for communicating with an I2C slave device is listed below:

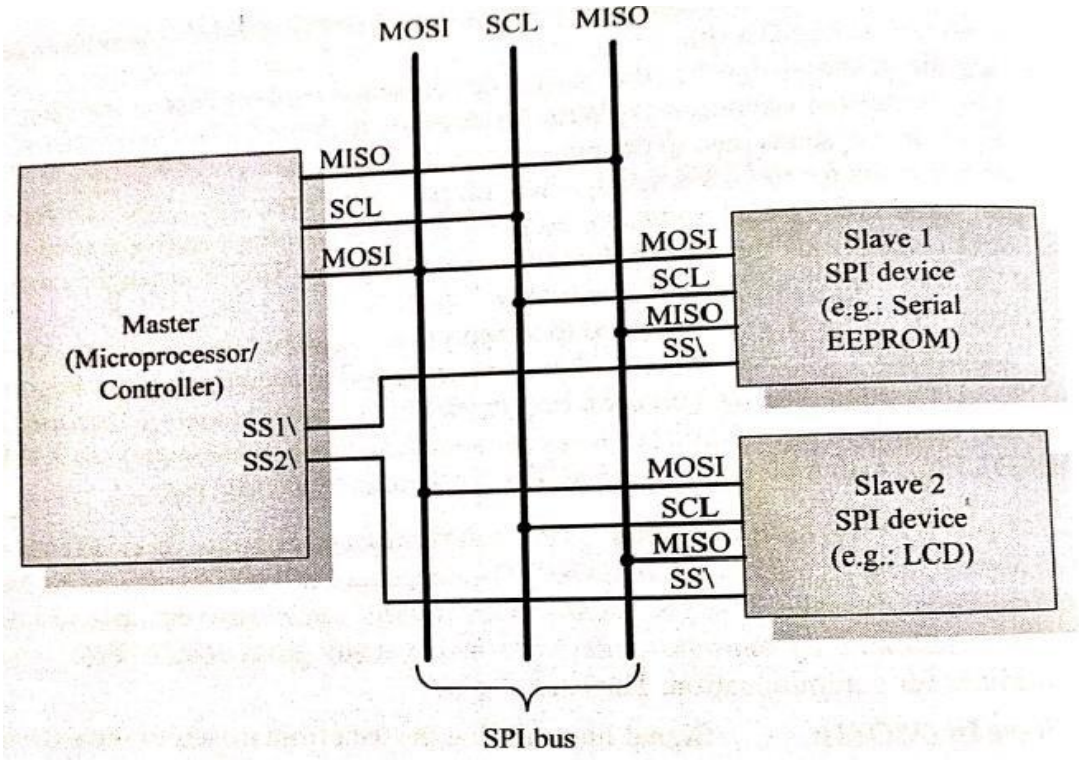
1. The master device pulls the clock line (SCL) of the bus to 'HIGH '
2. The master device pulls the data line (SDA) 'LOW', when the SCL line is at logic 'HIGH' (This is the 'Start' condition for data transfer)
3. The master device sends the address (7-bit or 10-bit wide) of the 'slave' device to which it wants to communicate, over the SDA line. Clock pulses are generated at the SCL line for synchronizing the bit reception by the slave device. The MSB of the data is always transmitted first. The data in the bus is valid during the 'HIGH' period of the clock signal
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 acknowledgement bit from the slave device whose address is sent on the bus along with the Read/ Write operation command. Slave devices connected to the bus compares the address received with the address assigned to them
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. Upon receiving the acknowledge bit, the master device sends the 8-bit data to the slave device over SDA line, if the requested operation is 'Write to device'. If the requested operation is 'Read from device', the slave device sends data to the master over the SDA line

8. The master device waits for the acknowledgement bit from the device upon byte transfer complete for a write operation and sends an acknowledge 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' (Indicating the 'STOP' condition).

Serial Peripheral Interface (SPI): *Serial Peripheral Interface Bus (SPI)* is asynchronous bi-directional full duplex four-wire serial interface bus. The concept of SPI was introduced by Motorola.

- SPI is a single master multi-slave system. It is possible to have a system where more than one SPI device can be master, provided the condition only one master device is active at any given point of time, is satisfied.
- SPI 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 (SO/ SDO).
    - Serial Clock (SCL): Signal line carrying the clock signals
    - Slave Select (SS): Signal line for slave device select. It is an active low signal.

The bus interface diagram is shown in the following Figure, illustrates the connection of master and slave devices on the SPI bus.



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'. The data out line (MISO) of all the slave devices when not selected floats at high impedance state.

SPI works on the principle of 'Shift Register'. The master and slave devices contain a special shift register for the data to transmit or receive. The size of the shift register is device dependent. Normally it is a multiple of 8.

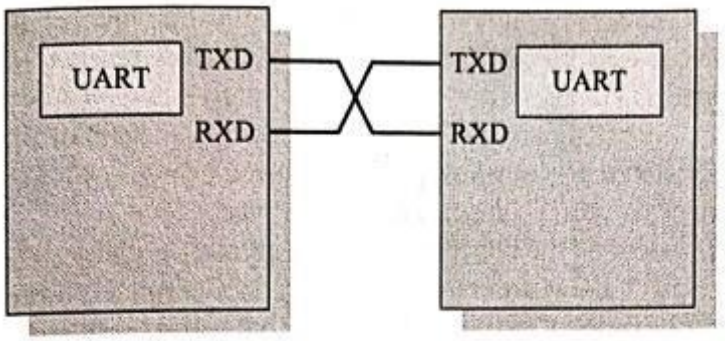
During transmission from the master to slave, the data in the master's shift register is shifted out to the MOSI pin and it enters the shift register of the slave device through the MOSI pin of the slave device. At the same time, the shifted out data bit from the slave device's shift register enters the shift register of the master device through MISO pin. In summary, the shift registers of 'master' and 'slave' devices form a circular buffer.

When compared to I2C, SPI bus is most suitable for applications requiring transfer of data in 'streams'. The only limitation is SPI doesn't support an acknowledgement mechanism.

Universal Asynchronous Receiver Transmitter (UART): *Universal Asynchronous Receiver Transmitter (UART)* based data transmission is an asynchronous form of serial data transmission.

- UART based serial data transmission doesn't require a clock signal to synchronize the transmitting end arid receiving end for transmission. Instead it relies upon the pre-defined agreement between the transmitting device and receiving device.
- The serial communication settings (Baud rate, number of bits per byte, parity, number 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 least significant bit of the data byte follows the 'start' bit.
- 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 baud rate settings. If the baud rate is '*x*' bits per second, the time slot available for one bit is *1/x* seconds.
- The receiver unit polls the receiver line at exactly half of the time slot available for the bit.
- If parity is enabled for communication, the UART of the transmitting device adds a parity bit (bit value is 1 for odd number of 1s in the transmitted bit stream and 0 for even number of 1s).
- 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 UART of the receiving device discards the 'Start', 'Stop' and 'Parity' bit from the received bit stream and converts the received serial bit data to a word.

For proper communication, the 'Transmit line' of the sending device should be connected to the 'Receive line' of the receiving device. The following Figure illustrates the same.



TXD: Transmitter line
RXD: Receiver line

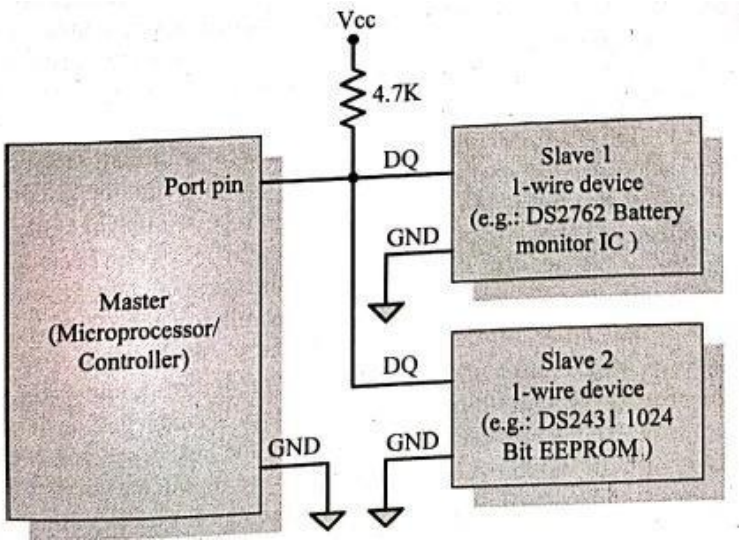
In addition to the serial data transmission function, UART provides hardware handshaking signal support for controlling the serial data flow.

UART chips are available from different semiconductor manufacturers. National Semiconductor's 8250 UART chip is considered as the standard setting UART. It was used in the original IBM PC.

1-Wire Interface: 1-wire interface is an asynchronous half-duplex communication protocol developed by Maxim Dallas Semiconductor. It is also known as *Dallas 1-Wire® protocol*. It makes use of only a single signal line (wire) called DQ for communication and follows the master-slave communication model.

- One of the key feature of 1-wire bus is that it allows power to be sent along the signal wire as well. The 1-wire slave devices incorporate internal capacitor (typically of the order of 800 pF) to power the device from the signal line.
- The 1-wire interface supports a single master and one or more slave devices on the bus.

The bus interface diagram shown in the following Figure illustrates the connection of master and slave devices on the 1-wire bus.



Every 1-wire device contains a globally unique 64-bit identification number stored within it. This unique identification number can be used for addressing individual devices present on the bus in case there are multiple slave devices connected to the 1-wire bus.

- 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 is listed below:

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 (Net Address Command followed by the 64-bit address of the device). This 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.

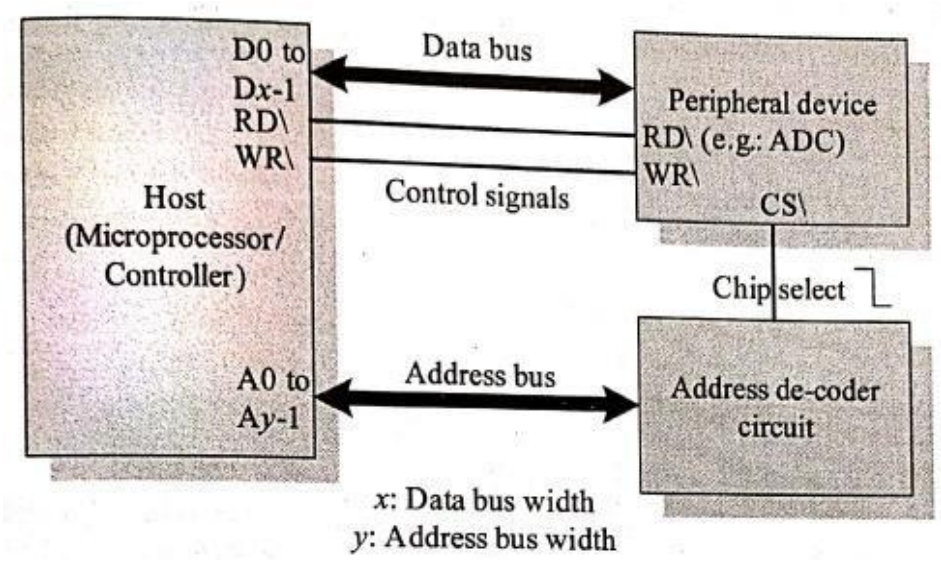
All communication over the 1-wire bus is master initiated. The communication over the 1-wire bus is divided into timeslots of 60 microseconds for regular speed mode of operation (16.3Kbps).

Parallel Interface: The on-board parallel interface 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 is controlled by the control signal interface between the device and the host.
- The 'Control Signals' for communication includes 'Read/ Write' signal and device select signal. The device normally contains a device select line and the device becomes active only when this line is asserted by the host processor.
- The direction of data transfer (Host to Device or Device to Host) can be controlled through the control signal lines for 'Read' and 'Write'. Only the host processor has control over the 'Read' and 'Write' control signals.
- The device is normally 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 thereby 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). Strict timing characteristics are followed for parallel communication.

- As mentioned earlier, parallel communication is host processor initiated. If a device wants to initiate the communication, it can inform the same to the processor through interrupts. For this, 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 4-bit, 8-bit, 16-bit, 32-bit or 64-bit, etc. The bus width supported by the device should be same as that of the host processor.
- Parallel data communication offers the highest speed for data transfer.

The bus interface diagram shown in the following Figure, illustrates the interfacing of devices through parallel interface.



External Communication Interfaces:

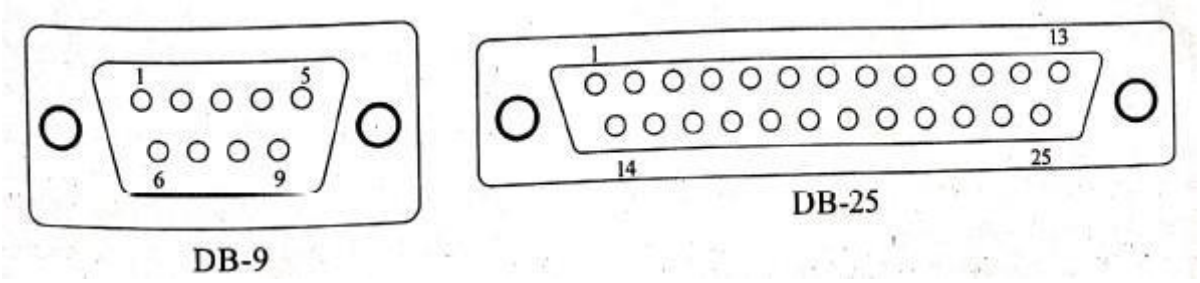
The *External Communication Interface* refers to the different communication channels/ buses used by the embedded system to communicate with the external world.

RS-232 C & RS-485: RS-232 C (Recommended Standard number 232, revision C) from the Electronic Industry Association is a legacy, full duplex, wired, asynchronous serial communication interface.

- The RS-232 interface is developed by the Electronics Industries Association (EIA) during the early 1960s. RS-232 extends the UART communication signals for external data communication.
- UART uses the standard TTL/ CMOS logic (Logic 'High' corresponds to bit value 1 and Logic 'Low' corresponds to bit value 0) for bit transmission; whereas RS-232 follows the EIA standard for bit transmission.

- As per the EIA standard, a logic '0' is represented with voltage between +3 and +25V and a logic' 1' is represented with voltage between -3 and -25V. In EIA standard, logic '0' is known as 'Space' and logic '1' as 'Mark'.

- The RS-232 interface defines various handshaking and control signals for communication apart from the 'Transmit' and. 'Receive' signal lines for data communication.
- RS-232 supports two different types of connectors:
  - DB-9: 9-Pin connector and
  - DB-25: 25-Pin connector.
- The following Figure illustrates the connector details for DB-9 and DB-25.



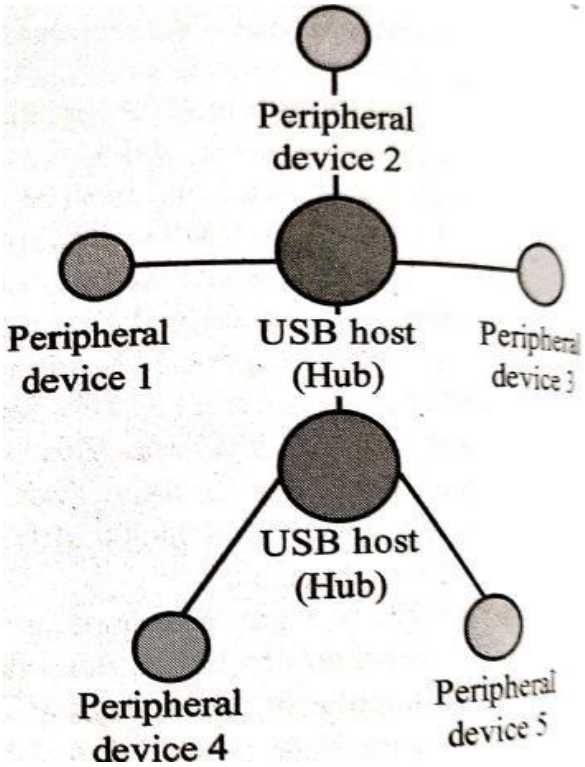
RS-232 is a point-to-point communication interface and the device involved in RS-232 communication are called '*Data Terminal Equipment (DTE)*' and '*Data Communication Equipment (DCE)*'.

- 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.
- RS-232 supports only point-to-point communication and not suitable for multi-drop communication. It uses single ended data transfer technique for signal transmission and thereby more susceptible to noise and it greatly reduces the operating distance.
- RS-422 is another serial interface standard from EIA for differential data communication. It supports data rates up to l00Kbps and distance up to 400 ft. RS-422 supports multi-drop communication with one transmitter device and receiver devices up to 10.
- RS-485 is the enhanced version of RS-422 and it supports multi-drop communication with up to 32 transmitting devices (drivers) and 32 receiving devices on the bus. The communication between devices in the bus uses the 'addressing' mechanism to identify slave devices.

Universal Serial Bus (USB): *Universal Serial Bus* is a wired high speed serial bus for data communication.

- The first version of USB (*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.
- The USB communication system follows a star topology with a USB host at the centre and one or more USB peripheral devices/ USB hosts connected to it.
- A USB 2.0 host can support connections up to 127, including slave peripheral devices and other USB hosts.

The following Figure illustrates the star topology for USB device connection.



- USB transmits data in packet format. Each data packet has a standard format. The USB communication is a host initiated one. 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.
- There are different standards for implementing the USB Host Control interface; namely Open Host Control Interface (OHCI) and Universal Host Control Interface (UHCI).
- The physical connection between a USB peripheral device and master device is established with a USB cable. The USB cable in USB 2.0 supports communication distance of up to 5 meters.
- The USB 2.0 standard uses two different types of connector at the ends of the USB cable for connecting the USB peripheral device and host device.
    - 'Type A' connector is used for upstream connection (connection with host) and Type B connector is used for downstream connection (connection with slave device).

- The USB connector present in desktop PCs or laptops are examples for 'Type A' USB connector.
- Both Type A and Type B connectors contain 4 pins for communication.

- The Pin details for the connectors are listed in the table given 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

USB uses differential signals for data transmission. It improves the noise immunity. USB interface has the ability to supply power to the connecting devices. Two connection lines (Ground and Power) of the USB interface are dedicated for carrying power. It can supply power up to 500 rnA at 5 V.

USB 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. Bulk transfer supports error checking and correction.
  - Transferring data to a printer is an example for bulk transfer.
- *Isochronous data transfer* is used for real-time data communication. In Isochronous transfer, data is transmitted as streams in real-time. Isochronous transfer doesn't support error checking and retransmission of data in case of any transmission loss.
  - All streaming devices like audio devices and medical equipment for data collection make use of the isochronous transfer.
- *Interrupt transfer* is used for transferring small amount of data. Interrupt transfer mechanism makes use of polling technique to see whether the USB device has any data to send. The frequency of polling is determined by the USB device and it varies from 1 to 255 milliseconds.
  - Devices like Mouse and Keyboard, which transmits fewer amounts of data, uses Interrupt transfer.

IEEE 1394 (Firewire): *IEEE 1394* is a wired isochronous high speed serial communication bus. It is also known as *High Performance Serial Bus (HPSB)*.

- The research on 1394 was started by Apple Inc. in 1985 and the standard for this was coined by IEEE.
- The implementation of it is available from various players with different names.
  - Apple Inc's implementation of 1394 protocol is popularly known as *Firewire*.

- *i.LINK* is the 1394 implementation from Sony Corporation
- *Lynx* is the implementation from Texas Instruments.

- 1394 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 up to 15 feet for interconnection.
- 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.
- 1394 is a popular communication interface for connecting embedded devices like Digital Camera, Camcorder, Scanners to desktop computers for data transfer and storage.
- Unlike USB interface (except USB OTG), IEEE 1394 doesn't require a host for communicating between devices.
  - For example, you can directly connect a scanner with a printer for printing.
- The data- rate supported by 1394 is far higher than the one supported by USB2.0 interface.
- The 1394 hardware implementation is much costlier than USB implementation.

Infrared (IrDA): *Infrared* is a serial, half duplex, line of sight based wireless technology for data communication between devices.

- IrDA is in use from the olden days of communication and you may be very familiar with it.
  - The remote control of your TV, VCD player, etc., works on Infrared data communication principle.
- Infrared communication technique uses infrared waves of the electromagnetic spectrum for transmitting the data.
- IrDA supports point-point and point-to-multipoint communication, provided all devices involved in the communication are within the line of sight.
- The typical communication range for IrDA lies in the range l0 cm to 1 m. The range can be increased by increasing the transmitting power of the IR device.
- IR supports data rates ranging from 9600bits/second to 16Mbps.
- Depending on the speed of data transmission IR is classified into Serial IR (SIR), Medium IR (MIR), Fast IR (FIR), Very Fast IR (VFIR) and Ultra Fast IR (UFIR).
  - SIR supports transmission rates ranging from 9600bps to 115.2kbps.
  - MIR supports data rates of 0.576Mbps and 1.152Mbps.
  - FIR supports data rates up to 4Mbps.
  - VFIR is designed to support high data rates up to l6Mbps.
  - The UFIR supports up to 96Mbps.

- IrDA communication involves a transmitter unit for transmitting the data over IR and a receiver for receiving the data. Infrared Light Emitting Diode (LED) is the IR source for transmitter and at the receiving end a photodiode acts as the receiver.
- Both transmitter and receiver unit will be present in each device supporting IrDA communication for bidirectional data transfer. Such IR units are known as '*Transceiver*'.
- Certain devices like a TV require control always require unidirectional communication and so they contain either the transmitter or receiver unit (The remote control unit contains the transmitter unit and TV contains the receiver unit).

Bluetooth (BT): *Bluetooth* is a low cost, low power, short range wireless technology for data and voice communication.

- Bluetooth was first proposed by 'Ericsson' in 1994.
- Bluetooth operates at 2.4GHz of the Radio Frequency spectrum and uses the Frequency Hopping Spread Spectrum (FHSS) technique for communication. Literally it supports a data rate of up to 1Mbps and a range of approximately 30 to 100 feet (version dependent) for data communication.
- Like IrDA, Bluetooth communication 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 Bluetooth communication. The physical link works on the wireless principle making use of RF waves for communication. Bluetooth enabled devices essentially contain a Bluetooth wireless radio for the transmission and reception of data.
    - The protocol part is responsible for defining the rules of communication. The rules governing the Bluetooth communication is implemented in the '*Bluetooth protocol stack*'.
- Each Bluetooth device will have a 48-bit unique identification number. Bluetooth communication follows packet based data transfer.
- Bluetooth supports point-to-point (device to device) and point-to-multipoint (device to multiple device broadcasting) wireless communication.
- The point-to-point communication follows the master slave relationship. A Bluetooth device can function as either master or slave.
    - When a network is formed with one Bluetooth device as master and more than one device as slaves, it is called a *Piconet*. A Piconet supports a maximum of seven slave devices.

Wi-Fi: *Wi-Fi* or *Wireless Fidelity* is the popular wireless communication technique for networked communication of devices.

- Wi-Fi follows the IEEE 802.11 standard. Wi-Fi is intended for network communication and supports Internet Protocol (IP) based communication. It is essential to have device identities in a multi-point communication to address specific devices for data communication.
- In an IP based communication each device is identified by an IP address, which is unique to each device on the network.
- Wi-Fi based communications require an intermediate agent called Wi-Fi router/ Wireless Access point to manage the communications.
    - The Wi-Fi router is responsible for restricting the access to a network, assigning IP address to devices on the network, routing data packets to the intended devices on the network.
- Wi-Fi enabled devices contain a wireless adaptor for transmitting and receiving data in the form of radio signals through an antenna. The hardware part of it is known as Wi-Fi Radio.
- Wi-Fi operates at 2.4GHz or 5GHz of radio spectrum and they co-exist with other ISM band devices like Bluetooth.

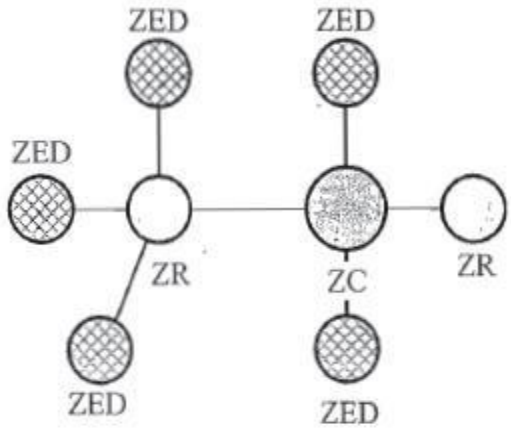
The following Figure illustrates the typical interfacing of devices in a Wi-Fi network.

ZigBee: *ZigBee* is a low power, low cost, wireless network communication protocol based on the IEEE 802.15.4-2006 standard.

- ZigBee is targeted for low power, low data rate and secure applications for Wireless Personal Area Networking (WPAN).
- The ZigBee specifications support a robust mesh network containing multiple nodes. This networking strategy makes the network reliable by permitting messages to travel through a number of different paths to get from one node to another.
- ZigBee operates worldwide 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.

- ZigBee Supports an operating distance of up to 100 meters and a data rate of 20 to 250Kbps.
- In the ZigBee terminology, each ZigBee device falls under any one of the following ZigBee device category:
    - *ZigBee Coordinator (ZC)/ Network Coordinator:* The ZigBee coordinator acts as the root of the ZigBee network. The ZC is responsible for initiating the ZigBee network and it has the capability to store information about the network.
    - *ZigBee Router (ZR)/ Full Function Device (FFD):* Responsible for passing information from device to another device or to another ZR.
    - *ZigBee End Device (ZED)/ 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.

The following Figure gives an overview of ZC, ZED and ZR in a ZigBee network:



General Packet Radio Service (GPRS), 3G, 4G, LTE: General Packet Radio Service is a communication technique for transferring data over a mobile communication 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 re-constructed by combining the received data packets.
- GPRS supports a theoretical maximum transfer rate of 171.2kbps.
- In GPRS communication, the radio channel is concurrently shared between several users instead of dedicating a radio channel to a cell phone user. The GPRS communication divides the channel into 8 timeslots and transmits data over the available channel.
- GPRS supports Internet Protocol (IP), Point to Point Protocol (PPP) and X.25 protocols for communication.
- GPRS is mainly used by mobile enabled embedded devices for data communication. The device should support the necessary GPRS hardware like GPRS modem and GPRS radio.

- To accomplish GPRS based communication, the carrier network also should have support for GPRS communication. GPRS is an old technology and it is being replaced by new generation data communication techniques like EDGE, High Speed Downlink Packet Access (HSDPA), etc. which offers higher bandwidths for communication.

EMBEDDED FIRMWARE:

Embedded firmware 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 un-avoidable part of an embedded system.

There are various methods available for developing the embedded firmware. They are listed below:

1. Write the program in high level languages like Embedded C/ C++ using an Integrated Development Environment
    - The IDE will contain a editor, compiler, linker, debugger, simulator, etc.
    - IDEs are different for different family of processors/ controllers.
        - For example, Keil microvision3 IDE is used for all family member of 8051 microcontroller, since it contains the generic 8051 compiler C51.
2. Write the program in Assembly language using the instructions supported by your application's target processor/ controller.

The instruction set for each family of processor/ controller is different and the program written in either of the methods given above 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'.
    - If you are following the Assembly language based programming technique, you can use the utilities supplied by the processor/ controller vendors to convert the source code into 'HEX File'.
    - Also third party tools are available, which may be of free of cost, for this conversion.
- For a beginner in the embedded software field, it is strongly recommended to use the high level language based development technique. The reasons for this being:
    - Writing codes in a high level language is easy, the code written in high level language is highly portable which means you can use the same code to run on different processor/ controller with little or less modification. The only thing you need to do is re-compile the

program with the required processor's IDE, after replacing the include files for that particular processor.

- Also the programs written in high level languages are not developer dependent. Any skilled programmer can trace out the functionalities of the program by just having a look at the program. It will be much easier if the source code 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.

- The embedded software development process in assembly language is tedious and time consuming. The developer needs to know about all the instruction sets of the processor/ controller or at least s/he should carry an instruction set reference manual with her/ him. A programmer using assembly language technique writes the program according to his/ her view and taste. Often he/ she may be writing a method or functionality which can be achieved through a single instruction as an experienced person's point of view, by two or three instructions in his/ her own style. So the program will be highly dependent on the developer. It is very difficult for a second person to understand the code written in Assembly even if it is well documented.

OTHER SYSTEM COMPONRNTS:

The *other system components* refer to the components/ circuits/ ICs which are necessary for the proper functioning of the embedded system. Some of these circuits may be essential for the proper functioning of the processor/ controller and firmware execution.

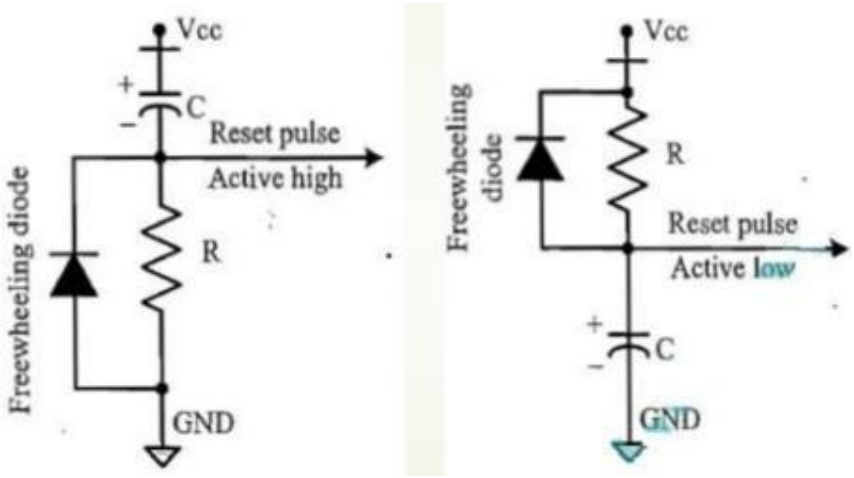
- Watchdog timer, Reset IC (or passive circuit), brown-out protection IC (or passive circuit) etc., are examples of circuits/ ICs which are essential for the proper functioning of the processors/ controllers. Some of the controllers or SoCs, integrate these components within a single IC and doesn't require such components externally connected to the chip for proper functioning.
- Depending on the system requirement, the embedded system may include other integrated circuits for performing specific functions, level translator ICs for interfacing circuits with different logic levels, etc.

Reset Circuit: The *reset circuit* 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 (Normally from vector address 0x0000 for conventional processors/ controllers.

- The reset signal can be either active high (The processor undergoes reset when the reset pin of the processor is at logic high) or active low (The processor undergoes reset when the reset pin of the processor is at logic 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 stabilize 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. Select the reset IC based on the type of reset signal and logic level (CMOS/ TTL) supported by the processor/ controller in use.
- Some microprocessors /controllers contain built-in internal reset circuitry and they don't require external reset circuitry.

The following Figure illustrates 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.

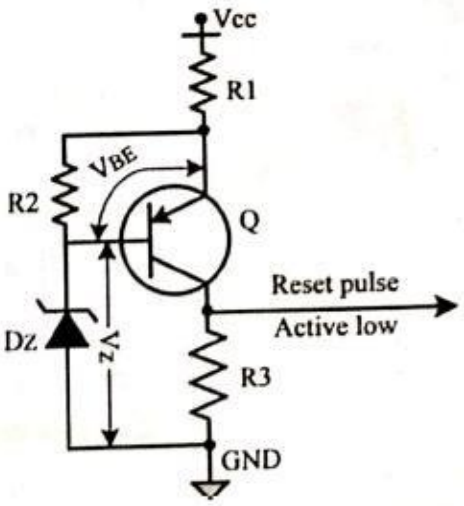


Brown-out Protection Circuit: *Brown-out protection circuit* prevents the processor/ controller from unexpected program execution behavior when the supply voltage to the processor/ controller falls below a specified voltage.

- It is essential for battery powered devices since there are greater chances for the battery voltage to drop below the required threshold. The processor behavior may not be predictable if the supply voltage falls below the recommended operating voltage. It may lead to situations like data corruption.
- A brown-out protection circuit holds the processor/ controller in reset state, when operating voltage falls below the threshold, until it rises above the threshold voltage.
- Certain processors/ controllers support built in brown-out protection circuit which monitors the supply voltage internally.

- If the processor/ controller don't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs.

The following Figure illustrates a brown-out circuit implementation using Zener diode and transistor for processor/ controller with active low Reset logic.



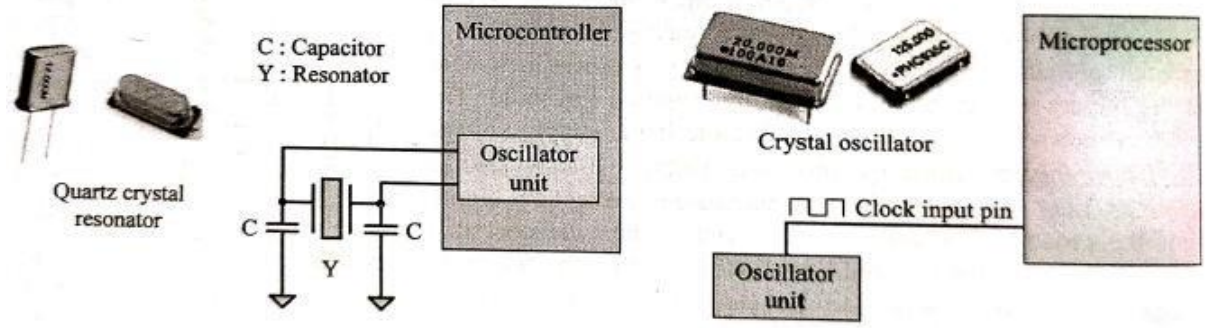
The Zener diode, Dz, and transistor, Q, forms the heart of this circuit. The transistor conducts always when the supply voltage V_{CC} is greater than that of the sum of V_{BE} and V_Z (Zener voltage). The transistor stops conducting when the supply voltage falls below the sum of V_{BE} and V_Z. Select the Zener diode with required voltage for setting the low threshold value for V_{CC}. The values of Rl, R2, and R3 can be selected based on the electrical characteristics of the transistor in use.

Oscillator Unit: A microprocessor/ microcontroller is a digital device made up of digital combinational and sequential circuits. The instruction execution of a microprocessor/ controller occurs in synchronization with a clock signal. 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. Quartz crystals and ceramic resonators are equivalent in operation, however they possess physical difference.
- Certain devices may not contain built-in oscillator unit and require the clock pulses to be generated and supplied externally.
- The speed of operation of a processor is primarily dependent on the clock frequency. However we cannot increase the clock frequency blindly for increasing the speed of execution. The logical circuits lying inside the processor always have an upper threshold value for the maximum clock at which the system can run, beyond which the system becomes unstable and non functional.
- The total system power consumption is directly 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 following Figure illustrates the usage of quartz crystal/ ceramic resonator and external oscillator chip for clock generation.



Real-Time Clock (RTC): *Real-Time Clock* is a system component responsible for keeping track of time. RTC holds information like current time (In hours, minutes and seconds) in 12-hour/ 24-hour format, date, month, year, day of the week, etc. and supplies timing reference to the system.

- RTC is intended to function even in the absence of power. RTCs are available in the form of Integrated Circuits from different semiconductor manufacturers like Maxim/Dallas, ST Microelectronics etc.
- The RTC chip contains a microchip for holding the time and date related information and backup battery cell for functioning in the absence of power, in a single IC package. The 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 synchronizing the operations of the OS kernel. The RTC 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 updating, managing software timers etc when an RTC 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 alarm interrupt).

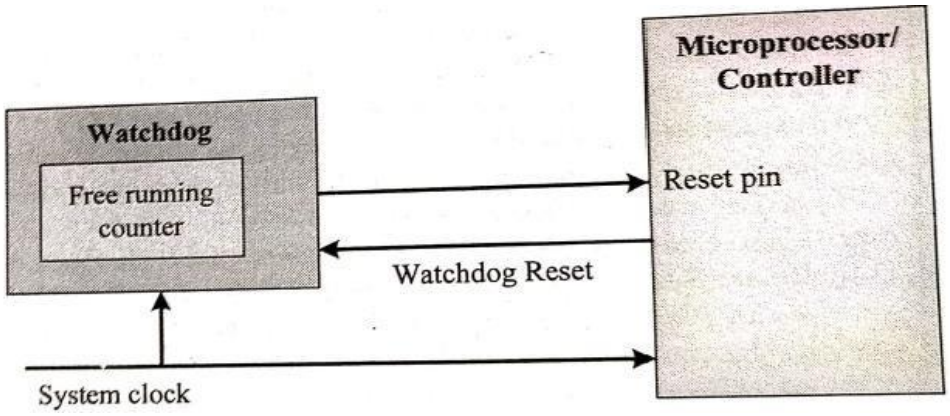
Watchdog Timer: In desktop Windows systems, if we feel our application is behaving in an abnormally or if the system hangs up, we have the *'Ctrl + Alt + Del'* to come out of the situation. What it happens to embedded system?

- We have a watchdog to monitor the firmware execution and reset the system processor/ microcontroller when the program execution hangs up. A *watchdog timer*, or simply a *watchdog*,

is a hardware timer for monitoring the firmware execution. Depending on the internal implementation, the watchdog timer 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 watchdog, or the highest count value for an up counting watchdog.

- If the watchdog counter is in the enabled state, the firmware can write a zero (for up counting watchdog implementation) to it before starting the execution of a piece of code and the watchdog 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 watchdog, you can reset the count by writing a 0 (for an up counting watchdog timer) to the watchdog timer register.
- If the processor/ controller doesn't contain a built in watchdog timer, the same can be implemented using an external watchdog timer IC circuit. The external watchdog timer uses hardware logic for enabling/ disabling, resetting the watchdog count, etc., instead of the firmware based 'writing' to the status and watchdog timer register. The microprocessor supervisor IC DS 1232 integrates a hardware watchdog timer in it.

The following Figure illustrates the implementation of the external watchdog timer based microprocessor based supervisor circuit for a small embedded system.



****

****

MODULE – 4

EMBEDDED SYSTEM DESIGN COMPONENTS

CHARACTERISTICS AND QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

No matter whether it is an embedded or a non-embedded system, there will be a set of *characteristics* describing the system. The non-functional aspects that need to be addressed in embedded system design are commonly referred as *quality attributes*. Whenever you design an embedded system, the design should take into consideration of both the functional and non-functional aspects.

CHARACTERISTICS OF AN EMBEDDED SYSTEM:

Unlike general purpose computing systems, embedded systems possess certain specific characteristics and these characteristics are unique to each embedded system.

Some of the important characteristics of an embedded system are:

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small size and weight
6. Power concerns

Application and Domain Specific:

- If you closely observe any embedded system, you will find that each embedded system is having certain functions to perform.
- Embedded systems are developed in such a manner to do only intended functions. They cannot be used for any other purpose. It is the major criterion which distinguishes an embedded system from a general purpose system.
    - For example, you cannot replace the embedded control unit of your microwave oven with your air conditioners embedded control unit, because the embedded control units of microwave oven and air conditioner are specifically designed to perform certain specific tasks.
- Also you cannot replace an embedded control unit developed for a particular domain say telecom with another control unit designed to serve another domain like consumer electronics.

MICROCONTROLLER AND EMBEDDED SYSTEMS

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.
    - Any changes happening in the Real world (which is called an *Event*) are captured by the sensors or input devices in Real Time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.
- The event may be a periodic one or an unpredicted one. If the event is an unpredicted one, then such system should be designed in such a way that it should be scheduled to capture the events without missing them.
- Embedded systems produce changes in output in response to the changes in the input. So they are generally referred as *Reactive Systems*.
- *Real Time System* operation means the timing behavior of the system should be deterministic; meaning the system should respond to requests or tasks in a know amount of time.
- A Real Time system should not miss any deadlines for tasks or operations.
- It is not necessary that all embedded systems should be Real Time in operations.
    - Embedded applications or systems which are mission critical, like flight control systems, Antilock Brake Systems (ABS), etc. are examples of Real Time systems.
- The design of an embedded Real Time system should take the worst case scenario into consideration.

Operates in Harsh Environment:

- It is not necessary that all embedded systems should be deployed in controlled environments.
- The environment in which the embedded system deployed 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. The design should take care of the operating conditions of the area where the system is going to implement.
    - For example, if the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.
- Here we cannot go for a compromise in cost. Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.
- Power supply 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.

Distributed:

- The term *distributed* means that, embedded systems may be a part of larger systems.
- Many numbers of distributed embedded systems form a single large embedded control unit.
    - An automatic vending machine is a typical example for this. The vending machine contains a card reader (for pre-paid vending systems), a vending unit, etc. Each of them are independent embedded units but they work together to perform the overall vending function.
    - Another example is the Automatic Teller Machine (ATM). An ATM contains a card reader embedded unit, responsible for reading and validating the user's AIM card, transaction unit for performing transactions, a currency counter for dispatching/ vending currency to the authorized person and a printer unit for printing the transaction details. We can visualize these as independent embedded systems. But they work together to achieve a common goal.
    - Another typical example of a distributed embedded system is the Supervisory Control And Data Acquisition (SCADA) system used in Control & Instrumentation applications, which contains physically distributed individual embedded control units connected to a supervisory module.

Small Size and Weight:

- Product aesthetics is an important factor in choosing a product.
    - For example, when you plan to buy a new mobile phone, you may make a comparative study on the pros and cons of the products available in the market. Definitely the product aesthetics (size, weight, shape, style, etc. will be one of the deciding factors to choose a product.
- People believe in the phrase "Small is beautiful". Moreover it is convenient to handle a compact device than a bulky product.
- In embedded domain also compactness is a significant deciding factor. Most of the application demands small size and low weight products.

Power Concerns:

- Power management is another important factor that needs to be considered in designing embedded systems.
- Embedded systems should be designed in such a way as to minimize the heat dissipation by the system.

- The production of high amount of heat demands cooling requirements like cooling fans which in turn occupies additional space and make a system bulky.
- Nowadays ultra low power components are available in the market. Select the design according to the low power components like low dropout regulators, and controllers/ processors with power saving modes.
- Also power management is a critical constraint in battery operated applications. The more the power consumption the less is the battery life.

QUALITY ATTRIBUTES OF AN EMBEDDED SYSTEM:

Quality attribute are non-functional requirements that need to be documented properly in any system design. If the quality attributes are more concrete and measurable, it will give a positive impact on the system development process and the end product.

The various quality attributes that needs to be addressed in any embedded system development are broadly classified into two, namely '*Operational Quality Attributes*' and '*Non-Operational Quality Attributes*'.

Operational Quality Attributes:

The *operational quality attributes* represent the relevant quality attributes related to the embedded system when it is in the operational mode or 'online' mode.

The important quality attributes coming under this category are listed below:

1. ***Response:*** is a measure of quickness of the system. It gives an idea about how fast your system is tracking the changes in input variables.
    - Most of the embedded systems demand fast response which should be almost Real Time.
        - For example, an embedded system deployed in flight control application should respond in a Real Time manner. Any response delay in the system will create potential damages to the safety of the flight as well as the passengers.
    - It is not necessary that all embedded systems should be Real Time in response.
        - For example, the response time requirement for an electronic toy is not at all time critical. There is no specific deadline that this system should respond wit in this particular timeline.

2. ***Throughput:*** deals with the efficiency of a system. *Throughput* is defined as the rate of production or operation of a defined process over a stated period of time.
    - The rates can be expressed in terms of units of products, batches produced, or any other meaningful measurements.

- In case of a Card Reader, throughput means how many transactions the Reader can perform in a minute or in an hour or in a day.

- Throughput is generally measured in terms of 'Benchmark'.
- A '*Benchmark*' is a reference point by which something can be measured.
- Benchmark can be a set of performance criteria that a product is expected to meet or a standard product that can be used for comparing other products of the same product line.

3. ***Reliability:*** 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.
   - *Mean Time Between Failures* (MTBF) and *Mean Time To Repair* (MTTR) are the terms used in defining system reliability.
     - *MTBF* gives the frequency of failures in hours/ weeks/ months.
     - *MTTR* specifies how long the system is allowed to be out of order following a failure.
   - For an embedded system with critical application need, it should be of order of minutes.

4. ***Maintainability:*** 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.
   - *Reliability* and *Maintainability* are considered as two complementary disciplines.
   - A more *reliable system* means a system with less corrective maintainability requirements and vice versa. As the reliability of the system increases, the chances of failure and non-functioning reduces, thereby the need for maintainability is also reduced.
   - Maintainability is closely related to the system availability. Maintainability can be broadly classified into two categories, namely, '*Scheduled* or *Periodic Maintenance* (*preventive maintenance*)' and '*Maintenance to unexpected failures* (*corrective maintenance*)'.
   - Some embedded products may use consumable components or may contain components which are subject to wear and tear and they should be replaced on a periodic basis. The period may be based on the total hours of the system usage or the total output the system delivered.
     - A printer is a typical example for illustrating the two types of maintainability. 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' number of printouts, to get quality prints. This is an example for '*Scheduled or Periodic maintenance*'.
     - If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem. This is an example of '*Maintenance to unexpected failure*'.

- In both of the maintenances (scheduled and repair), the-printer needs to be brought offline and during this time it will not be available for the user.
- In any embedded system design, the ideal value for availability is expressed as

$$Ai = MTBF / (MTBF + MTTR)$$

Where, Ai – Availability in the ideal conditions.

5. ***Security:*** aspect covers '*Confidentiality*', '*Integrity*', and '*Availability*' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section).
   - *Confidentiality* deals with the protection of data and application from unauthorized disclosure.
   - *Integrity* deals with the protection of data and application from unauthorized modifications.
   - *Availability* deals with protection of data and application from unauthorized users.
     - A very good example of the '*Security*' aspect in an embedded product is a Personal Digital Assistant (PDA). The PDA can be either a shared resource (e.g. PDAs used in LAB setups) or an individual one.
   - If it is a shared one, there should be some mechanism in the form of user name and password to access into a particular person's profile – An example of' Availability.
   - Also all data and applications present in the PDA need not be accessible to all users. Some of them are specifically accessible to administrators only. For achieving this, Administrator and user level s of security should be implemented – An example of Confidentiality.
   - Some data present in the PDA may be visible to all users but there may not be necessary permissions to alter the data by the users. That is Read Only access is allocated to all users – An example of Integrity.

6. ***Safety:*** '*Safety*' and '*Security*' are confusing terms. Sometimes you may feel both of them as a single attribute. But they represent two unique aspects in quality attributes.
   - *Safety* deals with the possible damages that can happen to
     - the operators,
     - public and the environment;
     - due to
       - the breakdown of an embedded system,
       - 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 a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level.
- Some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

Non-Operational Quality Attributes:

The quality attributes that needs to be addressed for the product 'not ' on-the basis of operational aspects are grouped under this category.

The important quality attributes coming under this category are listed below:

1. ***Testability & Debug-ability:*** deals with how easily one can test his/ her design, application; and by which means he/ she can test it.
   - For an embedded product, *testability* is applicable to both the embedded hardware and firmware.
     - *Embedded hardware testing* ensures that the peripherals and the total hardware functions in the desired manner, whereas *firmware testing* ensures that the firmware is functioning in the expected way.
   - *Debug-ability* is a means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system.
   - Debug-ability has two aspects in the embedded system development context, namely, hardware level debugging and firmware level debugging.
     - *Hardware debugging* is used for figuring out the issues created by hardware problems whereas *firmware debugging* is employed to figure out the probable errors that appear as a result of flaws in the firmware.

2. ***Evolvability:*** is a term which is closely related to Biology.
   - *Evolvability* is referred as the non-heritable variation. For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product (including firmware and hardware) can be modified to take advantage of new firmware or hardware technologies.

3. ***Portability:*** is a measure of 'system independence'.
   - An embedded product is said to be *portable* if the product is capable of functioning 'as such' in various en environments, target processors/ controllers and embedded operating systems.
   - The ease with which an embedded product can be ported on to a new platform is a direct measure of the rework required.

- A standard embedded product should always be flexible and portable.
- In embedded products, the term '*porting*' represents the migration of the embedded firmware written for one target processor (i.e., Intel x86) to a different target processor (say ARM Cortex M3 processor).
- If the firmware is written in a high level language like 'C' with little target processor-specific functions (operating system extensions or compiler specific utilities), it is very easy to port the firmware for the new processor by replacing those 'target processor-specific functions' with the ones for the new target processor and re-compiling the program for the new target processor• specific settings. Re-compiling the program or the new target processor generates the new target processor-specific machine codes.
- If the firmware is written in Assembly Language for a particular family of processor (say x86 family), it will be difficult to translate the assembly language instructions to the new target processor specific language and so the portability is poor.
- If you look into various programming languages for application development for desktop applications, you will see that certain applications developed on certain languages run only on specific operating systems and some of them run independent of the desktop operating systems.
    - For example, applications developed using Microsoft technologies (e.g. Microsoft Visual C++ using Visual studio) is capable of running only on Microsoft platforms and will not function on other operating systems; whereas applications developed using 'Java' from Sun Microsystems works on any operating system that supports Java standards.

4. ***Time to Prototype and Market:*** is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial products).
    - The commercial embedded product market is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product. There may be multiple players in the embedded industry who develop products of the same category (like mobile phone, portable media players, etc.). If you come up with a new design and if it takes long time to develop and market it, the competitor product may take advantage of it with their product.
    - Also, embedded technology is one where rapid technology change is happening. If you start your design by making use of a new technology and if it takes long time to develop and market the product, by the time you market the product, the technology might have superseded with a new technology.

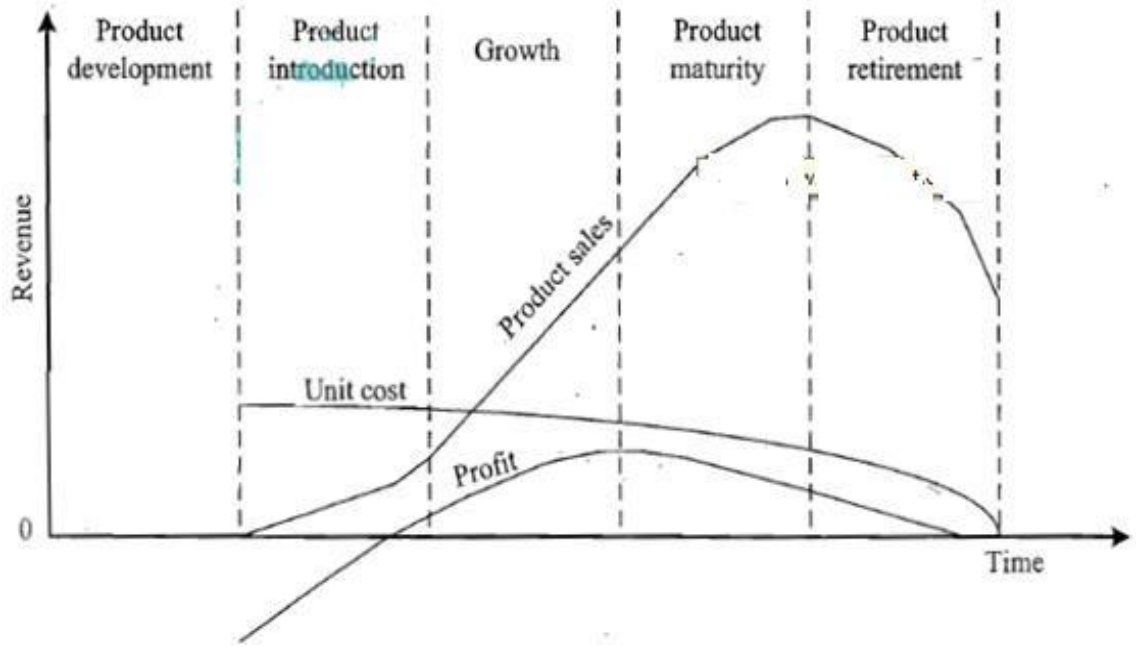
- *Product prototyping* helps a lot in reducing time-to-market. Whenever you have a product idea, you may not be certain about the feasibility of the idea.
- *Prototyping* is an informal kind of rapid product development in which the important features of the product under consideration are developed.
- The *time to prototype* is also another critical factor. If the prototype is developed faster, the actual estimated development time can be brought down significantly. In order to shorten the time to prototype, make use of all possible options like the use of off-the-shelf components, re-usable assets, etc.

5. ***Per Unit and Total Cost:*** is a factor which is closely monitored by both end user (those who buy the product) and product manufacturer (those who build the product).
   - Cost is a highly sensitive factor for commercial products. Any failure to position the cost of a commercial product at a nominal rate, may lead to the failure of the product in the market. Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product.
   - From a designer/ product development company perspective the ultimate aim of a product is to generate marginal profit. So the budget and total system cost should be properly balanced to provide a marginal profit.

The Product Life Cycle (PLC): Every embedded product has a product life cycle which starts with the design and development phase.

- The product idea generation; prototyping, Roadmap definition, actual product design and development are the activities carried out during this phase.
- During the *design and development phase* there is only investment and no returns.
- Once the product is ready to sell, it is introduced to the market. This stage is known as the *Product Introduction stage.*
- During the initial period the sales' and revenue will be low. There won't be much competition and the product sales and revenue increases with time. In the *growth phase*, the product grabs high market share.
- During the *maturity phase*, the growth and sales will be steady and the revenue reaches its peak.
- The *Product retirement/ Decline phase* starts with the drop in sales volume; market share and revenue. The decline happens due to various reasons like competition from similar product with enhanced features or technology changes, etc. At some point of the decline stage, the manufacturer announces discontinuing of the product.

- The different stages of the embedded products life cycle-revenue, unit cost and profit in each stage-are represented in the following Product Life-cycle graph.



EMBEDDED SYSTEMS – APPLICATION- AND DOMAIN- SPECIFIC

Embedded systems are *application and domain specific*, meaning; they are specifically built for certain applications in certain domains like consumer electronics, telecom, automotive, industrial control, etc.

It is possible to replace a general purpose computing system with another system which is closely matching with the existing system, whereas it is not the case with embedded systems.

Embedded systems are highly specialized in functioning and are dedicated for a specific application. Hence it is not possible to replace an embedded system developed for a specific application in a specific domain with another embedded system designed for some other application in some other domain.

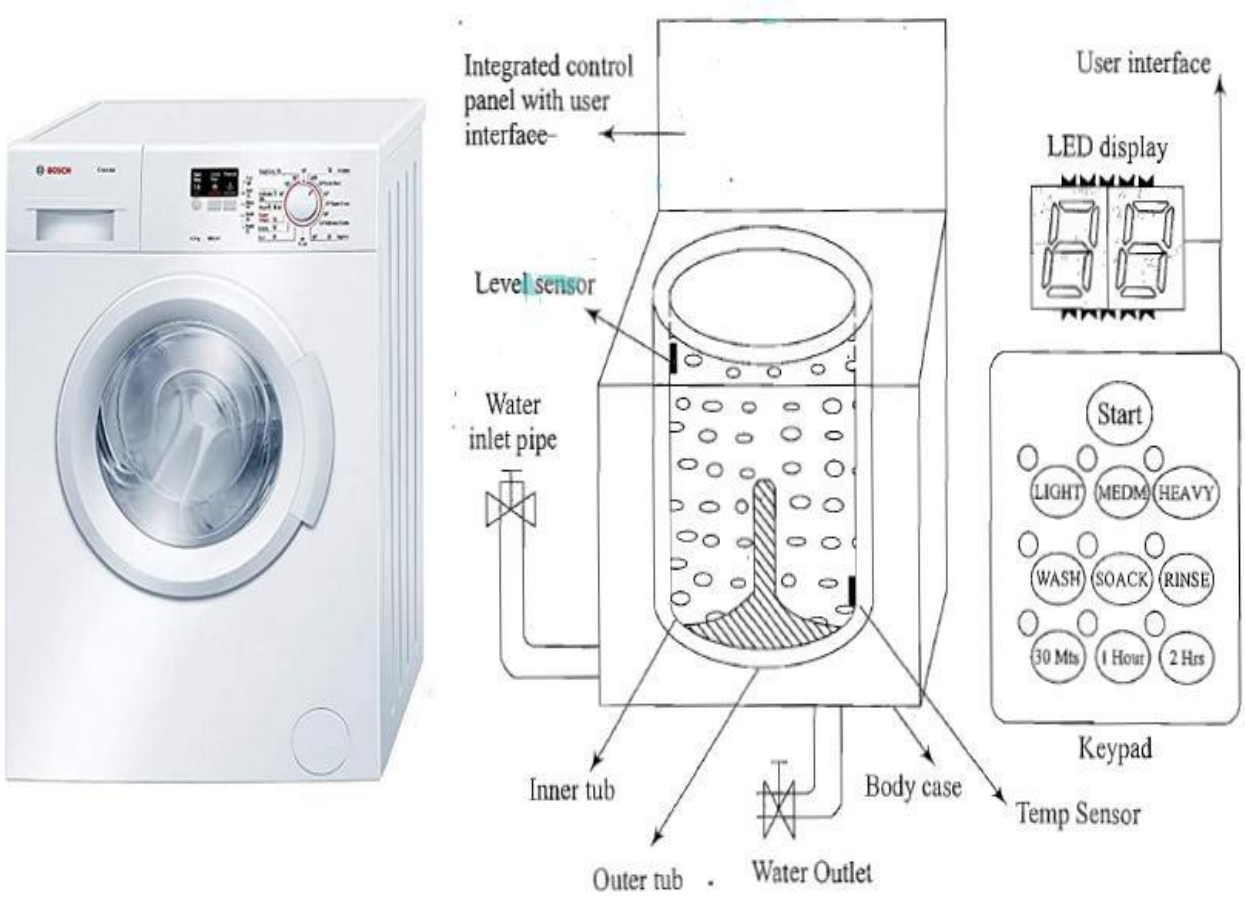
WASHING MACHINE – APPLICATION-SPECIFIC EMBEDDED STSREM:

Washing machine 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. One can see all these components in a washing machine. Some of them are visible and some of them may be invisible.
- The actuator part of the washing machine consists of a motorized agitator, tumble tub, water drawing pump and inlet valve to control the flow of water into the unit.
- The sensor part consists of the water temperature sensor, level sensor, etc.
- The control part contains a micro• processor/ controller 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. User feedback is reflected through the display unit and LEDs connected to the control board.

The functional block diagram of washing machine is shown in the following Figure.



Washing machine comes in two models, namely, top loading and front loading machines.

- In top loading models the agitator of the 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 up 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 the water over and over again. This is the first phase of washing.
- In the second phase of washing, water is pumped out from the tub and the 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*'.
- If you look into the keyboard panel of your washing machine you can see three buttons: Wash, Spin and Rinse. You can use these buttons to configure the washing stages.
- As you can see from the picture, the inner tub of the machine contains a number of holes and

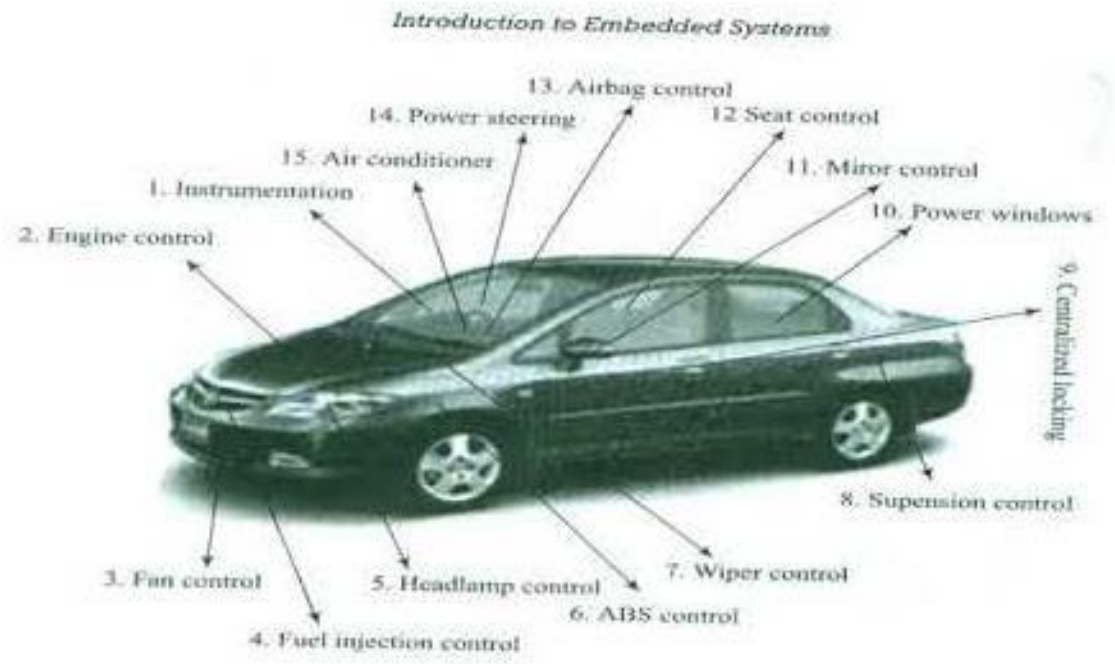
during the spin cycle the inner tub spins, and forces the water out through these holes to the stationary outer tub from which it is drained off through the outlet pipe.

It is to be noted that the design of washing machines may vary from manufacturer to manufacturer, but the general principle underlying in the working of the washing machine remains the same.

- The basic controls consist 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 into 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; clothe selector: Light, Medium, Heavy duty and washing time setting, etc.
- The output interface consists of LED/ LCD displays, status indication LEDs, etc. connected to the I/O bus of the controller.
- The other types of I/O interfaces which are invisible to the end user are different kinds of sensor interfaces: water temperature sensor, water level sensor, etc., and actuator interface including motor control for agitator and tub movement control, inlet water flow control, etc.

AUTOMATIVE – DOMAIN-SPECIFIC EXAMPLES EMBEDDED STSREM:

The major application domains of embedded systems are consumer, industrial, automotive, telecom, etc., of which telecom and automotive industry holds a big market share. The following Figure gives an overview of the various types of electronic control units employed in automotive applications.



Inner Working of Automotive Embedded Systems:

- Automotive embedded systems are the one where electronics take control over the mechanical systems. The presence of automotive embedded system in a vehicle varies from simple mirror and wiper controls to complex air bag controller and antilock brake systems (ABS).
- Automotive embedded systems are normally built around microcontrollers or DSPs or a hybrid of the two and are generally known as Electronic Control Units (ECUs). The number of embedded controllers in an ordinary vehicle varies from 20 to 40 whereas a luxury veh1cle like Mercedes S and BMW 7 may contain over 100 embedded controllers.
- The first embedded system used in automotive application was the microprocessor based fuel injection system introduced by Volkswagen 1600 in 1968.

The various types of electronic control units (ECUs) used in the automotive embedded industry can be broadly classified into two-*High-speed embedded control units* and *Low-speed embedded control units*.

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.

Low-speed Electronic Control Units (LECUs): are deployed in applications where response time is not so critical. They generally are built around low cost microprocessors/ microcontrollers and digital signal processors. 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., are examples of LECUs.

Automotive Communication Buses:

Automotive applications make use of serial buses for communication, which greatly reduces the amount of wiring required inside a vehicle. The different types of serial interface buses deployed in automotive embedded applications are –

1. ***Controller Area Network (CAN):*** The CAN bus was originally proposed by Robert Bosch, pioneer in the Automotive embedded solution providers.
   - CAN supports medium speed (ISO 11519-class B with data rates up to 125 Kbps) and high speed (ISO 11898 class C with data rates up to 1Mbps) data transfer.
   - CAN 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):*** LIN bus is a single master multiple slave (up to 16 independent slave nodes) communication interface.
   - LIN is a low speed, single wire communication interface with support for data rates up to 20 Kbps and is used for sensor/ actuator interfacing.
   - LIN bus follows the master communication triggering technique to eliminate the possible bus arbitration problem that can occur by the simultaneous talking of different slave nodes connected to a single interface bus.
   - LIN bus 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: MOST is targeted for automotive audio/ video equipment interfacing, used primarily in European cars.
   - A MOST bus is a multimedia fibre-optic point-to-point network implemented in a star, ring or daisy• chained topology over optical fibre cables.
   - The MOST bus specifications define the physical (electrical and optical parameters) layer as well as the application layer, network layer, and media access control.
   - MOST bus is an optical fibre cable connected between the Electrical Optical Converter (EOC) and Optical Electrical Converter (OEC), which would translate into the optical cable MOST bus.

Key Players of the Automotive Embedded Market:

The key players of the automotive embedded market can be visualized in three verticals namely, silicon providers, tools and platform providers and solution providers.

1. ***Silicon Providers:*** are responsible for providing the necessary chips which are used in the control application development.
   - The chip may be a standard product like microcontroller or DSP or ADC/ DAC chips.
   - Some applications may require specific chips and they are manufactured as Application Specific Integrated Chip (ASIC).
   - The leading silicon providers in the automotive industry are:
     a) **Analog Devices (www.analog.com):** Provider of world class digital signal processing chips, precision analog microcontrollers, programmable inclinometer/accelerometer, LED drivers, etc. for automotive signal processing applications, driver assistance systems, audio system, GPS/Navigation system, etc.
     b) **Xilinx (www.xilinx.com):** Supplier of high performance FPGAs, CPLDs and automotive specific IP cores for GPS navigation systems, driver information systems, distance

control, collision avoidance, rear1seat entertainment, adaptive cruise control, voice recognition, etc.

c) **Atmel (www.atmel.com):** Supplier of cost-effective high-density Flash controllers and memories. Atmel provides a series of high performance microcontrollers, namely, ARM®1 and 80C51. A wide range of Application Specific Standard Products (ASSPs) for chassis, body electronics, security, safety and car infotainment and automotive networking products for CAN, LIN and FlexRay are also supplied by Atmel.

d) **Maxim/Dallas (www.maxim-ic.com):** Supplier of world class analog, digital and mixed signal products (Microcontrollers, ADC/ DAC, amplifiers, comparators, regulators, etc), RF components, etc. for all kinds of automotive solutions.

e) **NXP semiconductor (www.nxp.com):** Supplier of 8/ 16/ 32 Flash microcontrollers.

f) **Renesas (www.renesas .com):** Provider of high speed microcontrollers, battery control systems, power train solutions, chassis and safety solutions, body control solutions, instrument cluster solutions, etc.

g) **Texas Instruments (www.ti.com):** Supplier of microcontrollers, digital signal processors and automotive communication control chips for Local Inter Connect (LIN bus products.

h) **Fujitsu (www.fmal.fujitsu.com):** Fujitsu is global leader in graphics display controllers (GDCs), including instrument clusters, in-dash navigation, heads-up displays and rear-seat entertainment. It also offers automotive controller for HD video in vehicle networks.

i) **Infineon (www.infineon.com):** Supplier of high performance microcontrollers and customized application specific chips.

j) **Freescale Semiconductor (www.freescale.com):** Solution provider for Advanced Driver Assistance Systems (ADAS), Body Electronics, Chassis and Safety, Instrument cluster, Power train and Hybrid systems.

k) **Microchip (www.microchip.com):** Supplier of robust automotive grade microcontroller, analog and memory products, CAN/ LIN transceivers, etc.

2. ***Tool and Platform Providers:*** 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.
   - Tools fall into two categories, namely embedded software application development tools and embedded hardware development tools.
   - Some of the leading suppliers of tools and platforms in automotive embedded applications are listed below:

a) **ENEA (www.enea.com):** Enea Embedded Technology is the developer of the OSE Real-Time operating system. The OSE RTOS supports both CPU and DSP and has also been specially developed to support multi-core and fault-tolerant system development. .

b) **The Math Works (www.mathworks.com):** It is the world's leading developer and supplier of technical software. It offers a wide range of tools, consultancy and training for numeric computation, visualization, modeling and simulation across many different industries. MathWork's breakthrough product is MATLAB – a high-level programming language and environment for technical computation and numerical analysis. Together MATLAB, SIMULINK, Stateflow and Real-Time Workshop provide top quality tools for data analysis, test & measurement, application development and deployment, image processing and development of dynamic and reactive systems for DSP and control applications.

c) **Keil Software (www.keil.com):** The Integrated Development Environment Keil Microvision from Keil software is a powerful embedded software design tool for 8051 & C166 family of microcontrollers.

d) **Lauterbach (http://www.lauterbach.com/):** It is the world's number one supplier of debug tools, providing support for processors from multiple silicon vendors in the automotive market.

e) Atego Modeling Tools (http://www.atego.com): It is the leading supplier of collaborative modeling tools for requirement analysis, specification, design and development of complex applications.

f) **Microsoft (www.microsoft.com):** It is a platform provider for automotive embedded applications. Microsoft's Windows Embedded Automotive is an extensible technology platform for automakers and suppliers to deliver in-car experiences that keep drivers connected and informed.

3. ***Solution Providers:*** Solution providers supply Original Equipment Manufacturer (OEM) and complete solution for automotive applications making use of the chips, platforms and different development tools.

   - The major players of this domain are listed below:

     a) **Bosch Automotive (www.boschindia.com):** Bosch is providing complete automotive solution ranging from body electronics, diesel engine control, gasoline engine control, power train systems, safety systems, in-car navigation systems and infotainment systems.

     b) **DENSO Automotive (www.globaldensoproducts.com):** Denso is an OEM and solution provider for engine management, climate control, body electronics, driving control & safety, hybrid vehicles, embedded infotainment and communications.

c) **Infosys Technologies (www.infosys.com):** Infosys is a solution provider for automotive embedded hardware and software. Infosys provides the competitive edge in integrating technology change through cost effective solutions.

d) **Delphi (www.delphi.com):** Delphi is the complete solution provider for engine control, safety, infotainment, etc., and OEM for spark plugs, bearings, etc.

HARDWARE SOFTWARE CO-DESIGN AND PROGRAM MODELING

In the traditional embedded system development approach, the hardware software partitioning is done at an early stage and engineers from the software group take care of the software architecture development and implementation, whereas engineers from the hardware group are responsible for building the hardware required for the product.

There is less interaction between the two teams and the development happens either serially or in parallel. Once the hardware and software are ready, the integration is performed.

The increasing competition in the commercial market and need for reduced 'time-to-market' the product calls for a novel approach for embedded system design in which the hardware and software are co-developed instead of independently developing both.

FUNDAMENTAL ISSUES IN HARDWARE SOFTWARE CO-DESIGN:

The hardware software co-design is a problem statement and when we try to solve this problem statement in real life we may come across multiple issues in the design. The following section illustrates some of the fundamental issues in hardware software co-design.

Selecting the Model: In hardware software co-design, 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. Most often designers switch between varieties of models from the requirements specification to the implementation aspect of the system design. The reason being, the objective varies with each phase.
    - For example, at the specification stage, only the functionality of the system is in focus and not the implementation information. When the design moves to the implementation aspect, the information about the system component is is revealed and the designer has to switch to a model capable of capturing the system's structure.

Selecting the Architecture: A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'

- The *architecture* specifies how a system is going to implement in terms of the number 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.
- Some of them fall into Application Specific Architecture Class (like Controller Architecture), while others fall into either General Purpose Architecture Class (CISC, RISC, etc.) or Parallel Processing Class (like VLIW, SIMD, MIMD, etc.).
    - The ***Datapath Architecture*** is best suited for implementing the data flow graph model where the output is generated as a result of a set of predefined computations on the input data. A datapath represents a channel between the input and output; and in datapath architecture the datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units. Ports connect the datapath to multiple buses.
    - The ***Finite State Machine Datapath (FSMD) architecture*** combines the controller architecture with datapath architecture. It implements a controller with datapath. The controller generates the control input, whereas the datapath processes the data. The datapath contains two types of I/O ports, out of which one acts as the control port for receiving/ sending the control signals from/ to the controller unit and the second I/O port interfaces the datapath with external world for data input and data output.
    - The ***Complex Instruction Set Computing (CISC) architecture*** uses an instruction set representing complex operations. It is possible for a CISC instruction set to perform a large complex operation with a single instruction. The use of a single complex instruction in place of multiple simple instructions greatly reduces the program memory access and program memory size requirement. However it requires additional silicon for implementing microcode decoder for decoding the CISC instruction. The datapath for the CISC processor is complex.
    - The ***Reduced Instruction Set Computing (RISC) architecture*** reuses instruction set representing simple operations and it requires the execution of multiple RISC instructions to perform a complex operation. The data path of RISC architecture contains a large register file for storing the operands and output. RISC instruction set is designed to operate on registers. RISC architecture supports extensive pipelining.

- The ***Very Long Instruction Word (VLIW) architecture*** implements multiple functional units (ALUs, multipliers, etc.) in the datapath. The VLIW instruction packages one standard instruction per functional unit of the datapath.
- ***Parallel Processing architecture*** implements multiple concurrent Processing Elements (PEs) and each processing element 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 architecture.
    - In SIMD architecture, a single instruction is executed in parallel with the help of the Processing Element. The scheduling-of the instruction execution and controlling of each PE is performed through a single controller. The SIMD architecture forms the basis of reconfigurable processor.
    - On the other hand, the processing elements of the MIMD architecture execute different instructions at a given point of time. The MIMD architecture forms the basis of multiprocessor systems. The PEs in a multiprocessor system communicates through mechanisms like shared memory and message passing.

Selecting the Language: A programming language captures a 'Computational Model' and maps it into architecture. There is no hard and fast rule to specify which language should be used for capturing this model. 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. On the other hand, a single language can be used for capturing a variety of models. Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model. The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily.

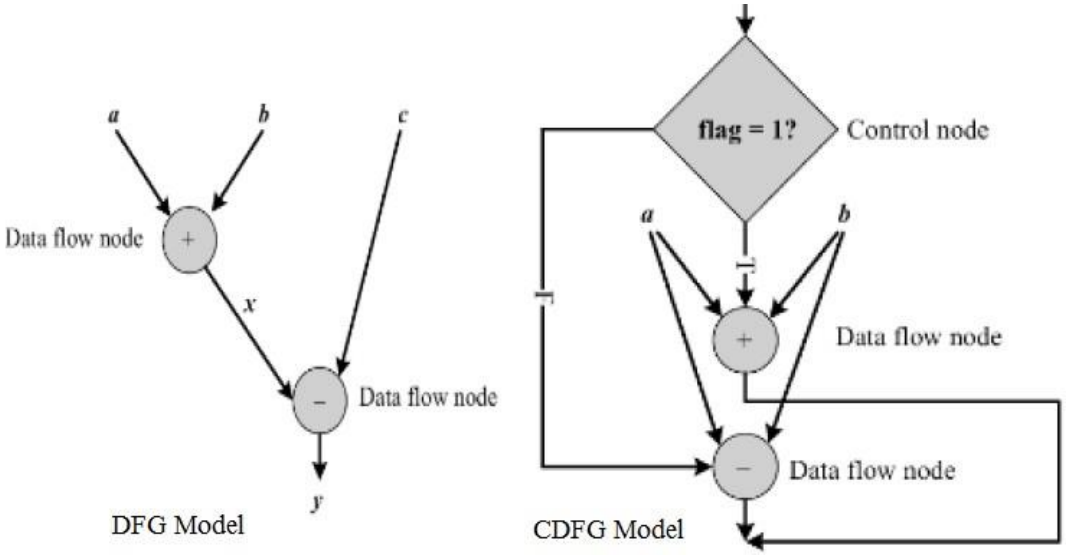
Partitioning System Requirements into Hardware and Software: It may be possible to implement the system requirements in either hardware or software (firmware). It is a tough decision making task to figure out which one to opt. Various hardware software trade-offs are used for making a decision on the hardware-software partitioning.

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.

Data Flow Graph/ Diagram (DFG) Model: The *DFG* model translates the data processing requirements into a data flow graph.

- The *Data Flow Graph model* is a data driven model in which the program execution is determined by data. This model emphasizes on the data and operations on the data transform the input data to output data.
- Indeed Data Flow Graph 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 in DFG notation.
- Embedded applications which are computational intensive and data driven are modeled using the DFG model.
- Suppose one of the functions in an application contains the computational requirement $x = a + b$; and $y = x - c$. The following Figure illustrates the implementation of a DFG model for implementing these requirements.



- In a DFG model, a data path is the data flow path from input to output.
- A DFG model is said to be *acyclic DFG* (*ADFG*), if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s).
- Feedback inputs (Output is fed back to Input), events, etc. are examples for non-acyclic inputs.
- A DFG model translates the program as a single sequential process execution.

Control Data Flow Graph/ Diagram (CDFG) Model: In a DFG model, the execution is controlled by data and it doesn't involve any control operations (conditionals).

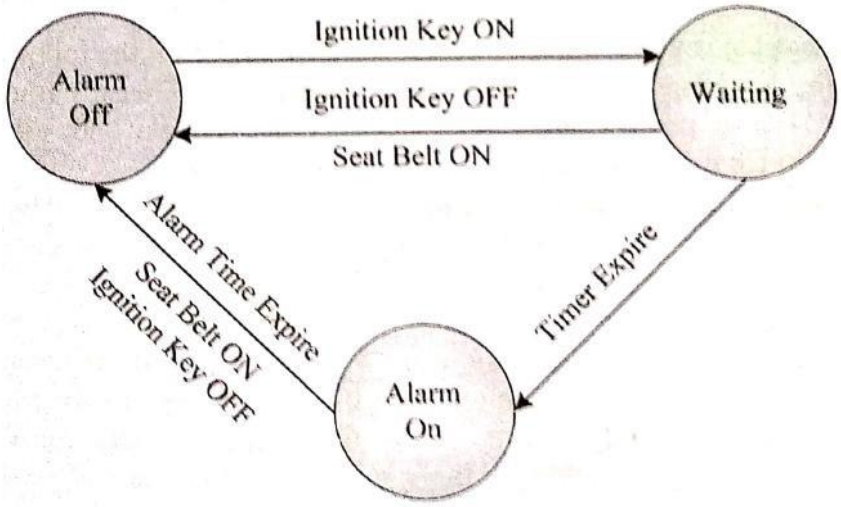
- The *Control DFG (CDFG) model* is used for modeling applications involving conditional program execution. CDFG models contains both data operations and control operations.

- The CDFG uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers. CDFG contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes. Let us have a look at the implementation of the CDFG for the following requirement.
- If flag = 1, x = a + b; else y = a – b; this requirement contains a decision making process. The CDFG model for the same is given in the above Figure.
- 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 modeled to a concurrent process model. The decision on which process is to be executed is determined by the control node.
    - A real world example for modeling the embedded application using CDFG is the capturing and saving of the image to a format set by the user in a digital still camera where everything is data driven starting from the Analog Front End which converts the CCD sensor generated analog signal to Digital Signal and the task which stores the data from ADC to a frame buffer for the use of a media processor which performs various operations like, auto correction, white balance adjusting, etc. The decision on, in which format the image is stored (formats like JPEG, TIFF, BMP, etc.) is controlled by the camera settings, configured by the user.

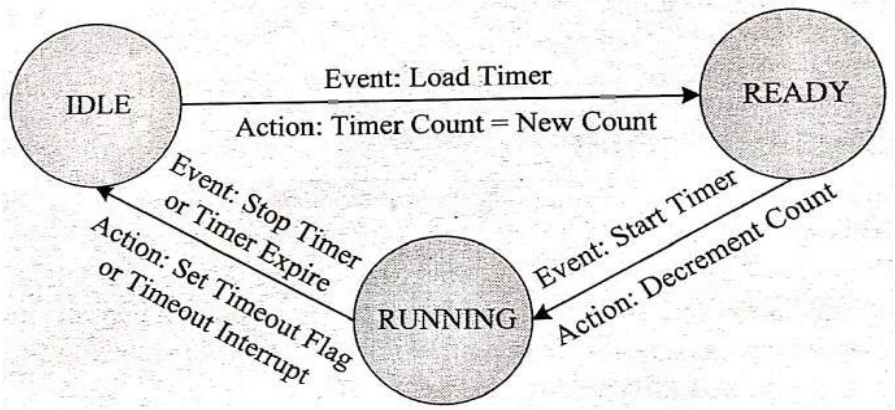
State Machine Model: The *State Machine model* is used for modeling reactive or event-driven embedded systems whose processing behavior is dependent on state transitions. Embedded systems used in the control and industrial applications are typical examples for event driven systems.

- The State Machine model describes the system behavior with '*States*', '*Events*', '*Actions*' and '*Transitions*'.
    - *State* is a representation of a current situation.
    - An *event* is an input to the *state*. The *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. In other words the system is described using a finite number of possible states.
    - As an example let us consider the design of an embedded system for driver/ passenger '*Seat Belt Warning*' in an automotive using the FSM model. The system requirements are captured as.
    - 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.

- The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/ passenger fasten 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*'.
- Using the FSM, the system requirements can be modeled as given in following Figure.



- The '*Ignition Key ON*' event triggers the 10 second timer and transitions the state to '*Waiting*'.
- If a '*Seat Belt ON*' or '*Ignition Key OFF*' event occurs during the wait state, the state transitions into '*Alarm Off*'.
- 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.
- The '*Alarm On*' state continues until a '*Seat Belt ON*' or '*Ignition Key OFF*' event or '*Alarm Time Expire*' event, whichever occurs first. The occurrence of any of these events transitions the state to '*Alarm Off*'.
- The wait state is implemented using a timer. The timer also has certain set of states and events for state transitions. Using the FSM model, the timer can be modeled as shown in the following Figure.

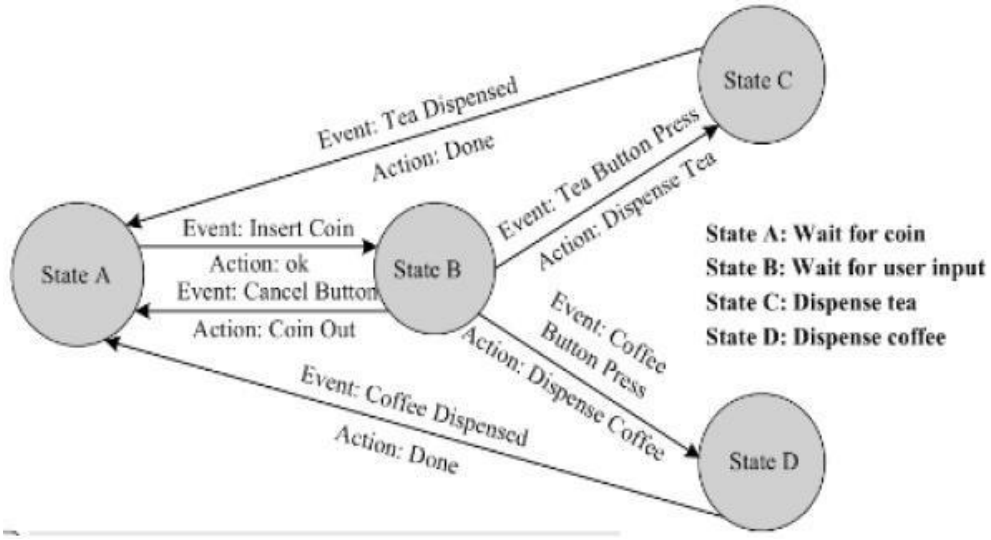


- As seen from the FSM, the timer state can be either '*IDLE*' or '*READY*' or '*RUNNING*'.
- During the normal condition when the timer is not running, it is said to be in the '*IDLE*' state.
- The timer is said to be in the '*READY*' state when the timer is loaded with the count corresponding to the 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*' even occurs. The timer state changes to '*IDLE*' from '*RUNNING*' on receiving a '*Stop Timer*' or '*Timer Expire*' event.

Example 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 press '*Cancel*' to cancel the order and take back the coin.

The FSM representation for the above requirement is given in the following Figure.

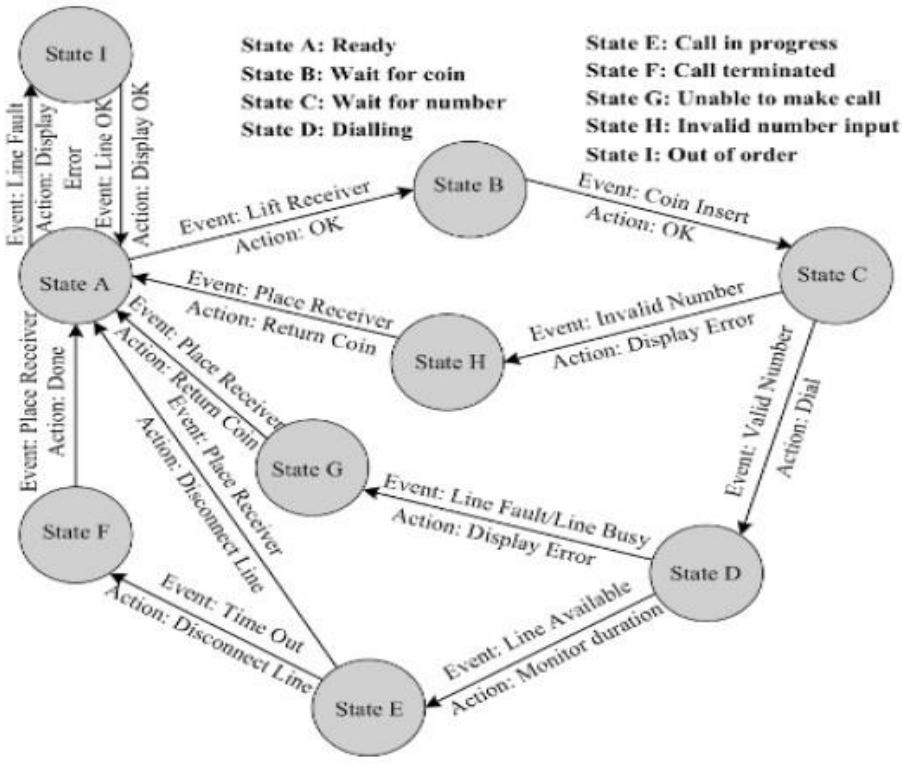


- The FSM representation contains four states namely; '*Wait for coin*' '*Wait for User Input*', '*Dispense Tea*' and '*Dispense Coffee*'.
- The event '*Insert Coin*' (5 rupee coin insertion), transitions the state to '*Wait for User Input*'. The system stays in this state until a user input is received from the buttons '*Cancel*', '*Tea*' or '*Coffee*'.
- 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*' or *'Dispense Coffee*' respectively.
- Once the coffee/ tea vending is over, the respective states transitions back to the '*Wait for Coin*' state.

Example 2: Design a coin operated public telephone unit based on FSM model for the following requirements.

1. The calling process is initiated by lifting the receiver (off-hook) of the telephone unit
2. After lifting the phone 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 the 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.

The FSM model shown in the following Figure is a simple representation.



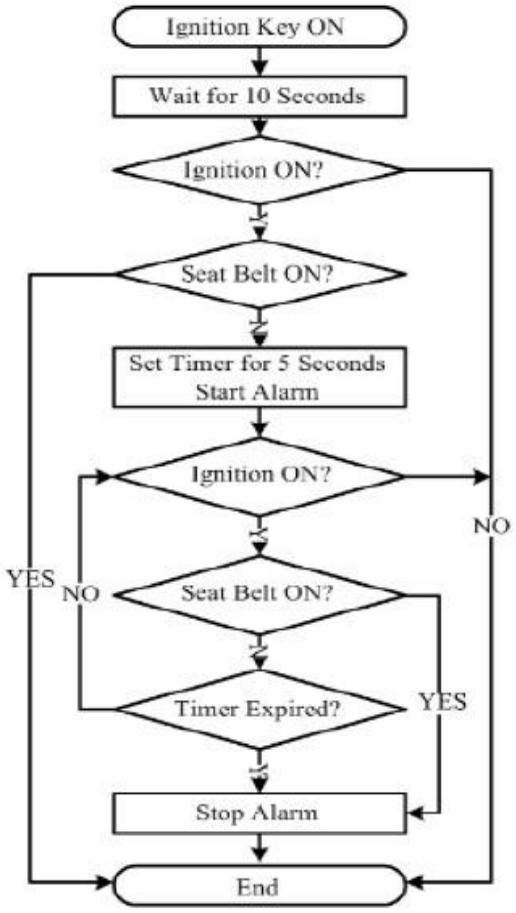
- Most of the time state machine model translates the requirements into sequence driven program and it is difficult to implement concurrent processing with FSM. This limitation is addressed by the *Hierarchical/ Concurrent Finite State Machine model (HCFSM)*.
- The HCFSM is an extension of the FSM for supporting concurrency and hierarchy.
- HCFSM extends the conventional state diagrams by the AND, OR decomposition of States together with inter level transitions and a broadcast mechanism for communicating between concurrent processes.

- HCFSM uses statecharts for capturing the states, transitions, events and actions.
- The Harel Statechart, UML State diagram, etc. are examples for popular statecharts used for the HCFSM modeling of embedded systems.

Sequential Program Model: In the sequential programming Model, the functions or processing requirements are executed in sequence. It is same as the conventional procedural programming.

- Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
- FSMs are good choice for sequential program modeling.
- Another important tool used for modeling sequential program is Flow Charts.
- The FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow.
- The execution of functions in a sequential program model for the '*Seat Belt Warning*' system is illustrated below.

```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn ()
{   wait_10sec ():
    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 () == ON));
            stop_alarm ():
        }
    }
}
```

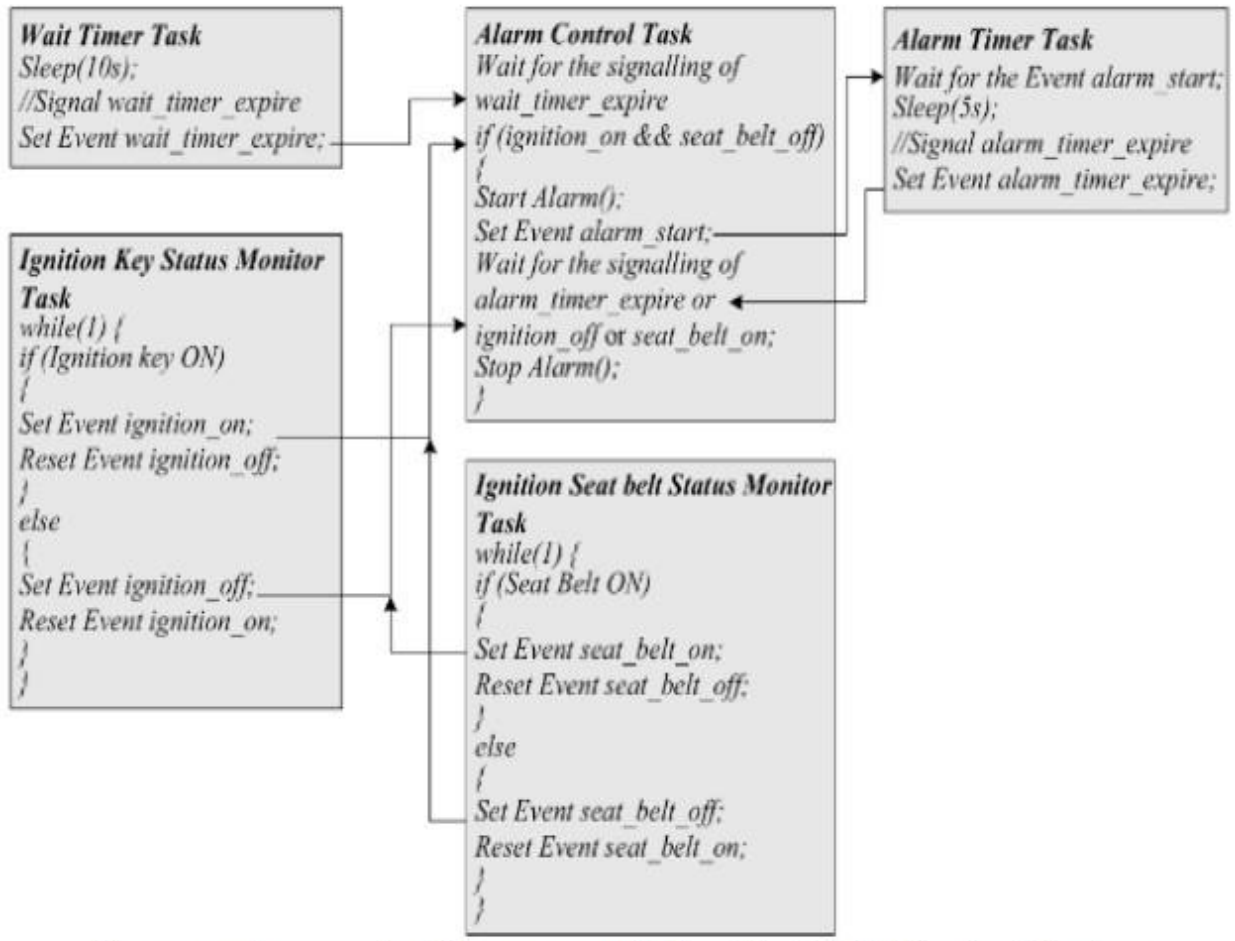


Concurrent/ Communicating Process Model: The concurrent or communicating process model models concurrently executing tasks/ processes.

- It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution.
- Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilization, when the task involves I/O waiting, sleeping for specified duration etc.
- If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively, when the subtask under execution goes to a wait or sleep mode, by switching the task execution.
- However, concurrent processing model requires additional overheads in task scheduling, task synchronization and communication.
- As an example for the concurrent processing model let us examine how we can implement the '*Seat Belt Warning*' system in concurrent processing model. We can split the tasks into:
    1. Timer task 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)
- We have five tasks here and we cannot execute them randomly or sequentially. We need to synchronize their execution through some mechanism.
- We need to start the alarm only after the expiration of the 10 seconds wait timer and that too only if the seat belt is OFF and the ignition key is ON. Hence the alarm: control task is executed only when the wait timer is expired and if the ignition key is in the ON state and seat belt is in the OFF state.
- One way of implementing a concurrent model for the '*Seat Belt Warning*' system is illustrated in the following Figure.

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*

Tasks for "Seat Belt Warning" System



Concurrent processing Program model for "Seat Belt Warning" System

Object-Oriented Model: The object-oriented model is an object based model for modeling system requirements. It disseminates a complex software requirement into simple well defined pieces called *objects*.

- Object-oriented model brings re-usability, maintainability and productivity in system design.
- In the object-oriented modeling, object is an entity used for representing or modeling a particular piece of the system. Each object is characterized by a set of unique behavior and state. A class is an abstract description of a set of objects and it can be considered as a '*blueprint*' of an object.
- A class represents the state of an object through member variables and object behavior 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, whereas public variables and functions are accessible within the class as well as outside the class.
    - The protected variables and functions are protected from external access.

- However classes derived from a parent class can also access the protected member functions and variables.
- The concept of object and class brings abstraction, hiding and protection.

EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT

The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements mentioned in the requirements for the particular embedded product. Firmware is considered as the master brain of the embedded system. Embedded firmware is stored at a permanent memory (ROM).

Designing embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details I/O port details, configuration and register details of various hardware chips used and some programming language (low level assembly language or a high level languages like C/C++/JAVA).

Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modeling skills like UML or flow chart based representation. The UML diagrams or flow chart gives a diagrammatic representation of the decision items to be taken and the tasks to be performed.

EMBEDDED FIRMWARE DESIGN APPROACHES:

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

- Two basic approaches are used for embedded firmware design. They are '*Conventional Procedural Based Firmware Design*' (also known as '*Super Loop Model*') and '*Embedded Operating System (OS) Based Design*'.

The Super Loop Based Approach:

The *Super Loop based firmware development approach* is adopted for applications that are not time critical and where the response time is not so important.

- Super loop approach is very similar to a conventional procedural programming where the code is executed task by task. The task, listed at the top of the program code, is executed first and the tasks, just below the top are executed after completing the first task. This is a true procedural one.
- In a multiple task based system, each task is executed in serial in this approach. The firmware execution flow for this will be –
    1. Configure the common parameters and perform initialization for various hardware components memory, registers, etc.

2. Start the first task and execute it
3. Execute the second task
4. Execute the next task
5. :
6. :
7. Execute the last defined task
8. Jump back to the first task and follow the same flow.

- From the firmware execution sequence, it is obvious that the order in which the tasks to be executed are fixed and they are hard coded in the code itself. Also the operation is an infinite loop based approach.
- We can visualize the operational sequence listed above in terms of a '*C*' program code as –

```
void main ()
{
        configurations ();
        initializations ();
        while (1)
        {
                Task 1 ();
                Task 2 ():
                :
                :
                Task n ();
        }
}
```

- From the above '*C*' code you can see that the tasks *1* to *n* are performed one after another and when the last task (n^{th} task) is executed, the firmware execution is again re-directed to Task *1* and it is repeated forever in the loop. This repetition is achieved by using an infinite loop. This approach is also referred 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.
  - A hardware reset brings the program execution back to the main loop.
  - An interrupt request suspends the task execution temporarily and performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted.

- The 'Super loop based design' doesn't require an operating system, since there is no need for scheduling which task is to be executed and assigning priority to 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. Hence the code for performing these tasks will be residing in the code memory without an operating system image.
- Super loop based design is deployed in low cost embedded products and products where response time is not time critical. Some embedded products demand this type of approach.
    - For example, reading/ writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authenticating the operation, reading/writing, etc.
    - A typical example of a 'Super loop based' product is an electronic video game toy containing keypad and display unit. 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. Even if the application misses a key press, it won't create any critical issues; rather it will be treated as a bug in the firmware.
- The 'Super loop based design' is simple and straight forward without any OS related overheads.
- The major drawback of this approach is that any failure in any part of a task will affect the total system. If the program hangs up at some point while executing a task, it may remain there forever and ultimately the product stops functioning.
- Another major drawback of 'Super loop based design' approach is the lack of real timeliness. If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases. This brings the probability of missing out some events.
    - For example in a system with Keypads, according to the 'Super loop design', there will be a task for monitoring the keypad connected I/O lines and this need not be the task running while you press the keys. In order to identify the key press, you may have to press the keys for a sufficiently long time, till the keypad status monitoring task is executed internally by the firmware. This will really lead to the lack of real timeliness.
- There are corrective measures for this also. The best advised option in use interrupts for external events requiring real time attention.

The Embedded Operating System (OS) Based Approach:

The *Operating System (OS) based approach* contains operating systems, which can be either a General Purpose Operating System (GPOS) or a Real Time Operating System (RTOS) to host the user written application firmware.

- The *General Purpose OS (GPOS) based design* is very similar to a conventional PC based application development, where the device contains an operating system (Windows/ Unix/ Linux, etc. for Desktop PCs) and you will be creating and running user applications on top of it.
    - Example of a GPOS used in embedded product development is Microsoft® Windows XP Embedded 8.1, which offers customization to use industry devices like Personal Digital Assistants (PDAs), Hand held devices/ Portable devices, Point of Sale (PoS) Terminals, Patient Monitoring Systems, etc.
- OS based applications also require 'Driver software' for different hardware present on the board to communicate with them.
- The *Real Time Operating System (RTOS) based design* is employed in embedded products demanding Real-time response.
- RTOS respond in a timely and predictable manner to events. Real Time operating system contains a Real Time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc. A Real Time Operating System (RTOS) allows flexible scheduling of system resources like the CPU and memory and offers some way to communicate between tasks.
    - '*Windows Embedded Compact*', '*pSOS*', '*VxWorks*', '*ThreadX*', '*MicroC/OS-III*', '*Embedded Linux*', '*Symbian*', etc., are examples of RTOS employed in embedded product development.

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES:

Embedded firmware can be –

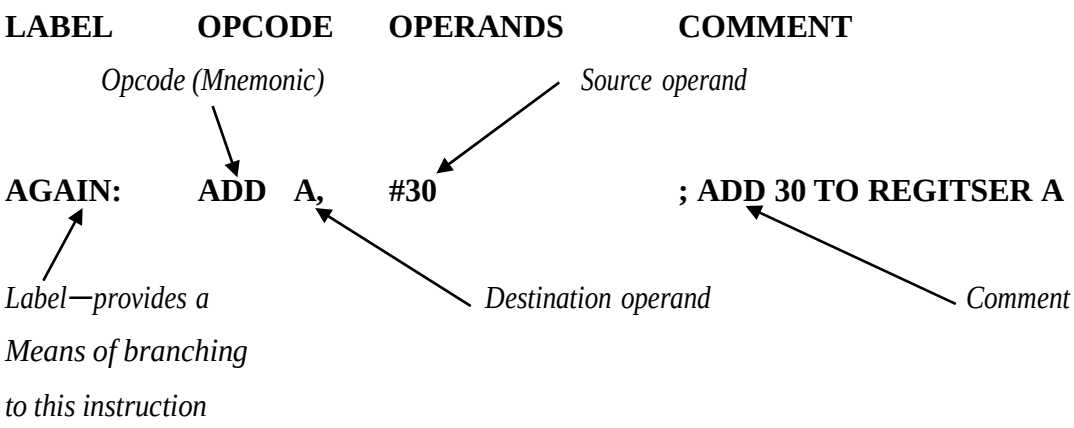
- ✓ a target processor/ controller specific language (Assembly language or Low level language) or
- ✓ a target processor/ controller independent language (C, C++, JAVA, etc.) commonly known as High Level Language or
- ✓ a combination of Assembly and High level Language.

Assembly Language Based Development:

'*Assembly language*' is the human readable notation of '*machine language*', whereas '*machine language*' is a processor understandable language.

- Processors deal only with binaries (*1*s and *0*s). Machine language is a binary representation. Machine language is made readable by using specific symbols called '*mnemonics*'. Hence machine language can be considered as an interface between processor and programmer.
- Assembly language and machine languages are processor/ controller dependent and an assembly program written for one processor/ controller family will not work with others.

- *Assembly language programming* is the task of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler.
- The general format of an assembly language instruction is an Opcode followed by Operands.
- The *Opcode* tells the processor/ controller what to do and the Operands provide the data and information required to perform the action specified by the opcode.
    - Example: *MOV A, #30* – This instruction mnemonic moves decimal value 30 to the 8051 Accumulator register. Here *MOV A* is the Opcode and 30 is the operand. The same instruction when written in machine language: *01110100 00011110* – where the first 8-bit binary value *01110100*, represents the opcode *MOV A* and the second 8-bit binary value *00011110* represents the operand 30.
    - The mnemonic *INC A* is an example for instruction holding operand implicitly in the Opcode. The machine language representation of the same is *00000100*.
- 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 + Operands). Each line of an assembly language program is split into four fields as given below:



- *LABEL* is an optional field. A '*LABEL*' is an identifier used extensively in programs to reduce the reliance on programmers or remembering where data or code is located.
- LABEL is commonly used for representing a memory location, address of a program, sub-routine, code portion, etc. Labels are used for representing subroutine names and jump locations in Assembly language programming.
- The maximum length of a label differs between assemblers.
- Assemblers insist strict formats for labeling. Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from *0* to *9* and special character _ (underscore).
- 'LABEL' is not a mandatory field; it is optional.

- The sample code given below using 8051 Assembly language illustrates the structured assembly language programming. Each Assembly instruction should be written in a separate line.

```
DELAY:   MOV   R0, #255        ; Load Register R0 with 255.
         DJNE  R1, DELAY       ; Decrement R1 and loop till R1 = 0.
         RET                   ; Return to calling program.
```

- The Assembly program contains a main routine and it may or may not contain subroutines. The example given above is a subroutine, which can be invoked by a main program by the Assembly instruction: *LCALL DELAY*
  - Executing this instruction transfers the program flow to the memory address referenced by the 'LCALL DELAY'.
- It is a good practice to provide comments to your subroutines by indicating the purpose of that subroutine/ instruction.
- While assembling the code a ';' informs the assembler that the rest of the part coming in the line after the ';' symbol is comments and simply ignore it.
- In the above example the label DELAY represents the reference to the start of the subroutine DELAY. The required address is calculated by the assembler at the time of assembling the program and it replaces the label.
- Label can also be directly replaced by putting the desired address first and then writing the Assembly code, as given below:

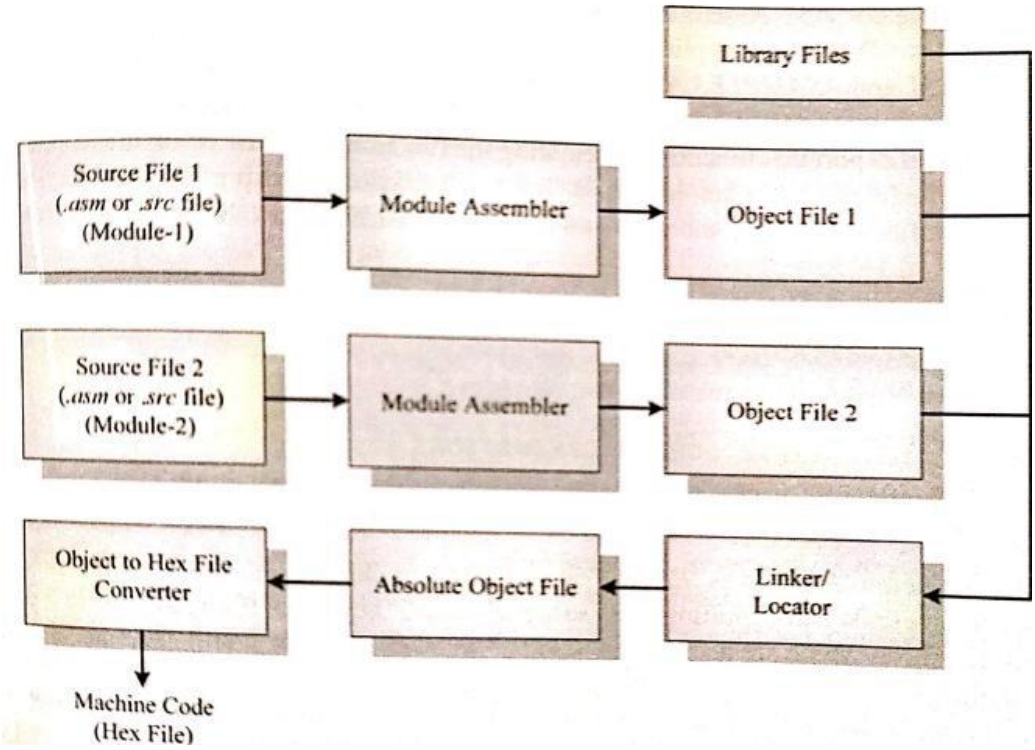
```
ORG 0100H
         MOV   R0, #255        ; Load Register R0 with 255.
         DJNE  R1, DELAY       ; Decrement R1 and loop till R1 = 0.
         RET                   ; Return to calling program.
```

- The statement *ORG 0100H* in the above example is not an assembly language instruction; it is an *assembler directive* instruction. It tells the assembler that the Instructions from here onward should be placed at location starting from *0100H*. The Assembler directive instructions are known as '*pseudo• ops*'. They are used for –
  1. Determining the start address of the program (e.g. *ORG 0000H*)
  2. Determining the entry address of the program (e.g. *ORG 0100H*)
  3. Reserving memory for data variables, arrays and structures (e.g. *NUM1 EQU 70H*)
  4. Initializing variable values (e.g. *VALUE DATA 12H*)
     - The *EQU directive* is used for allocating memory to a variable and *DATA directive* is used for initializing a variable with data. No machine codes are generated for the 'pseudo-ops'.

- The Assembly language program written in assembly code is saved as *.asm* (Assembly file) file or *.src* (source) file. Any text editor like 'notepad' or 'WordPad' from Microsoft® or the text editor provide by an Integrated Development (IDE) tool can be used for writing the assembly instructions.
- When a program is too complex or too big; the entire code can be divided into sub-modules and each module can be re-usable (called as *Modular Programming*). Modular programs are usually easy to code, debug and alter.

Source File to Object File Translation: Translation of assembly code to machine code is performed by assembler. The assemblers for different target machines are different. Assemblers from multiple vendors are available in market. A51 Macro assembler from Keil software is a popular assembler for the 8051 family microcontroller.

- The various steps involved in the conversion of a program written in assembly language to corresponding binary file/ machine language is illustrated in the following Figure.



- Each *source module* is written in Assembly and is stored as .src file or .asm file. Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions. On successful assembling of each .src/ .asm file a corresponding object file is created with extension '.obj'.
- The *object file* does not contain the absolute address of where the generated code needs to be placed on the program memory and hence it is called *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.
- Importing a variable or function from a module (taking a variable or function from any one of other modules) is done by declaring that variable or function as *EXTREN*) in the module where it is going to be accessed.

Library File Creation and Usage: Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time. When the linker processes a library, only those object modules in the library that are necessary to create the program are used. Library files are generated with extension '.*lib*'.

- Library is some kind of source code hiding technique. If you don't want to reveal the source code behind the various functions you have written in your program and at the same time you want them to be distributed to application developers for making use of them in their applications, you can supply them as library files and give them the details of the public functions available from the library (function name, function input/output, etc). For using a library file in a project, add the library to the project.
    - '*LIB51*' from K eil Software is an example for a library creator and it is used for creating library files for A51 Assembler/ C51 Compiler for 8051 specific controller.

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

- *Linker* is a program which combines the target program with the code of other programs (modules) and library routines.
- During the process of linking, the absolute object module is created. The *object module* contains the target code and information about other programs and library routines that are required to call during the program execution.
- An absolute object file or module does not contain any re-locatable code or data. All code and data reside at fixed memory locations. The absolute object file is used for creating hex files for dumping into the code memory of the processor/ controller.

- '*BL51*' from Keil Software is an example for a Linker & Locater for A51 Assembler/ C51 Compiler for 8051 specific controller.

Object to Hex File Converter: This is the final stage in the conversion of Assembly language (mnemonics) to machine understandable language (machine code).

- *Hex File* is the representation of the machine code and the hex file is dumped into the code memory of the processor/ controller.
- The hex file representation varies depending on the target processor/ controller make.
    - For Intel processors/ controllers the target hex file format will be '*Intel HEX*' and for Motorola, the hex file should be in '*Motorola HEX*' format.
- *HEX files* are ASCII files that contain a hexadecimal representation of target application. Hex file is created from the final 'Absolute Object File' using the Object to Hex File Converter utility.
- '*QH51*' from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/ C51 Compiler for 8051 specific controller.

Advantages of Assembly Language Based Development: Assembly Language based development was (is) the most common technique adopted from the beginning of embedded technology development. Thorough understanding of the processor architecture memory organization, register sets and mnemonics is very essential for Assembly Language based development. The major advantages of Assembly Language based development listed below.

- ***Efficient Code Memory and Data Memory Usage (Memory Optimization):*** Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations. This lead less utilization of code memory and efficient utilization of data memory.
- ***High Performance:*** Optimized code not only improves the code memory usage, but also improves the total system performance. Through effective assembly coding, optimum performance can be achieved for a target application.
- ***Low Level Hardware Access:*** Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers, and low level interrupt routines, etc., are making use of direct assembly coding; since low level device specific operation support is not commonly available with most of the high-level language cross compilers.
- ***Code Reverse Engineering:*** *Reverse engineering* is the process of understanding the technology behind a product by extracting the information from a finished product. Reverse engineering is performed by 'hackers' to reveal the technology behind 'Proprietary Products'.

Drawbacks of Assembly Language Based Development: Every technology has its own pros and cons. From certain technology aspects assembly language development is the most efficient technique. But it is having the following technical limitations also.

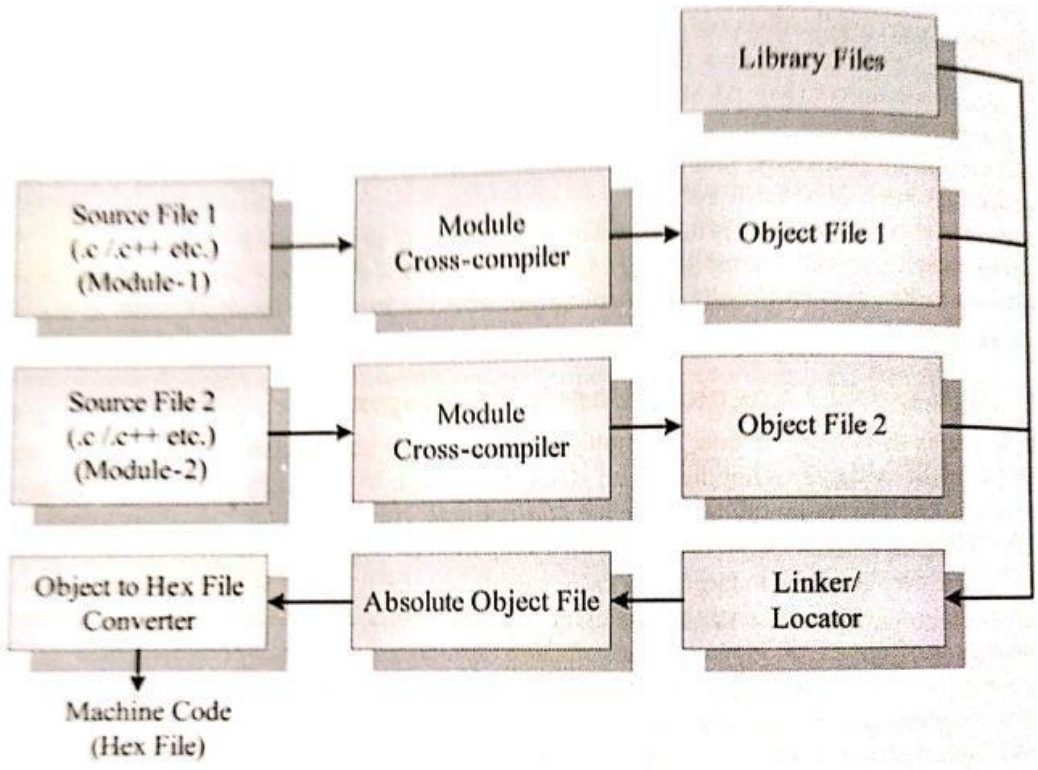
- ***High Development Time:*** Assembly language is much harder to program than high level languages. The developer must pay attention to more details and must have thorough knowledge of the architecture, memory organization and register details of the target processor in use. Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.
- ***Developer Dependency:*** There is no common written rule for developing assembly language based applications, whereas all high level languages instruct certain set of rules for application development. In assembly language programming, the developers will have the freedom to choose the different memory location and registers. Also the programming approach varies from developer to developer depending on his/ her taste.
    - For example moving data from a memory location to accumulator can be achieved through different approaches.
    - If the approach done by a developer is not documented properly at the development stage, he/ she may not be able to recollect why this approach is followed at a later stage or when a new developer is instructed to analyze this code, he/ she also may not be able to understand. Hence upgrading an assembly program on a later stage is very difficult.
- ***Non-Portable:*** Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Inte x86 family of processors) and cannot be re-used for another target processors/ controllers (Say ARM Cortex M family of processors). If the target processor/ controller changes, a complete re-writing of the application using the assembly instructions for the new target processor/ controller is required.

High Level Language Based Development:

Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture. Also applications developed in Assembly language are non-portable.

- Any high level language (like C, C++ or Java) with a supported cross compiler (for converting the application developed in high level language to target processor specific assembly code) for the target processor can be used for embedded firmware development.
- The most commonly used high level language for embedded firmware application development is 'C'.

- '*C*' is the well defined, easy to use high level language with extensive cross platform development tool support. Nowadays Cross-compilers for C++ is also emerging out and embedded developers are making use of C++ for embedded application development.
- The various steps involved in high level language based embedded firmware development is same as that of assembly language based development, except that the conversion of source file written in high level language to object file is done by a cross-compiler, whereas in Assembly language based development, it is carried out by an assembler.
- The various steps involved in the conversion of a program written in high level language to corresponding binary file/ machine language is illustrated in the following Figure.



- The program written in any of the high level language is saved with the corresponding language extension (.c for C, .cpp for C++ etc). Any text editor like 'notepad' or 'WordPad ' from Microsoft® or the text editor provided by an Integrated Development (IDE) tool supporting the high level language can be used for writing the program.
- Most of the high level languages support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language.
- Translation of high level source code to executable object code is done by a cross-compiler. The cross-compilers for different high level languages for the same target processor are different.
- C51 is a popular. Cross-compiler available for '*C*' language for the 8051 family of micro controller. Conversion of each module's source code to corresponding object file is performed by the cross compiler.

- Rest of the steps 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 of High Level Language Based Development:

- ***Reduced Development Time:*** Developer requires less or little knowledge on the internal hardware details and architecture of the target processor/ controller. Bare minimal knowledge of the memory organization and register details of the target processor in use and syntax of the high level language are the only pre-requisites for high level language based firmware development.
    - All other things will be taken care of by the cross-compiler used for the high level language. Thus the ramp up time required by the developer in understanding the target hardware and target machine's assembly instructions is waived off by the cross compiler and it reduces the development time by significant reduction in developer effort.
    - High level language based development also refines the scope of embedded firmware development from a team of specialized architects to anyone knowing the syntax of the language and willing to put little effort on understanding the minimal hardware details.
    - With high level language, each task can be accomplished by lesser number of lines of code compared to the target processor/ controller specific Assembly language based development.
- ***Developer Independency:*** The syntax used by most of the 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.
    - High level languages always instruct certain set of rules for writing the code and commenting the piece of cadet lf the developer strictly adheres to the rules, the firmware will be 100% developer independent.
- ***Portability:*** Target applications written in high level languages are converted to target processor / controller understandable format (machine codes) by cross-compiler.
- An application written in high level language for a particular target processor can easily be converted to another target processor/ controller specific application, with little or less effort by simply re-compiling/ little code modification followed by re-compiling the application for the required processor/ controller.

Limitations of High Level Language Based Development:

- Some cross-compilers available for high level languages may not be so efficient in generating optimized target processor specific instructions. Target images created by such compilers may be messy and non-optimized in terms of performance as well as code size.

- The investment required for high level language based development tools (Integrated Development Environment incorporating cross-compiler) is high compared to Assembly Language based firmware development tools.

Mixing Assembly and High Level language:

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

Mixing Assembly with High Level Language (e.g. Assembly Language with 'C'): Assembly routines are mixed with '*C*' in situations where the entire program is written in '*C*' and the cross-compiler do not have a built in support for implementing certain features like Interrupt Service Routine functions (ISR) or if the programmer wants to take advantage of the speed and optimized code offered by machine code generated by hand written assembly rather than cross compiler generated machine code.

When accessing certain low level hardware, the timing specifications may be very critical and a cross-compiler generated binary may not be able to offer the required time specifications accurately. Writing the hardware/ peripheral access routine in processor/ controller specific Assembly language and invoking it from '*C*' is the most advised method to handle such situations.

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

The following steps give an idea how *C51* cross-compiler performs the mixing of Assembly code with '*C*':

1. Write a simple function in *C* that passes parameters and returns values the way you want your assembly routine to.
2. Use the SRC directive (*#PRAGMA SRC* at the top of the file) so that the *C* compiler generates an *.SRC* file instead of an *.OBJ* file.
3. Compile the *C* file. Since the *SRC* directive is specified, the *.SRC* file is generated. The *.SRC* file contains the assembly code generated for the *C* code you wrote.
4. Rename the *.SRC* file to *.A51* file.
5. Edit the *.A51* file and insert the assembly code you want to execute in the body of the assembly function shell included in the *.A51* file.

Mixing High level language with Assembly (e.g. 'C' with Assembly Language): Mixing the code written in a high level language like '*C*' and Assembly language is useful in the following scenarios:

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 optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the Assembly, etc. But some 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*'.

Inline Assembly: *Inline assembly* is another technique for inserting target processor/ controller specific Assembly instructions at any location of a source code written in high level language '*C*'. This avoids the delay in calling an assembly routine from a '*C*' code. Special keywords are used to indicate the start and end of Assembly instructions. *C51* uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

'C' versus 'Embedded C': '*C*' is a well structured, well defined and standardized general purpose programming language with extensive bit manipulation support. '*C*' offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like pay roll systems, banking applications, etc). The conventional '*C*' language follows *ANSI* standard and it incorporates various library files for different operating systems. A platform (operating system) specific application, known as, compiler is used for the conversion of programs written in '*C*' to the target processor (on which the OS is running) specific binary files. Hence it is a platform specific development.

Embedded C can be considered as a subset of conventional '*C*' language. *Embedded C* supports all '*C*' instructions and incorporates a few target processor specific functions/ instructions. 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 specific functions/ instructions depends upon the processor/ controller as well as supported cross-compiler for the particular *Embedded C* language. A software program called '*Cross-compiler*' is used for the conversion of programs written in *Embedded C* to target processor/ controller specific instructions (machine language).

C	Embedded C
C is a general purpose programming language, which can be used to design any type of desktop-based applications.	Embedded C is an extension of C language and used to develop microcontroller-based applications.
C language program is hardware independent.	Embedded C program is hardware dependent.
C language uses standard compilers to compile and execute the program.	Embedded C requires specific compilers that are able to generate particular hardware/ microcontroller-based output.
Readability, modifications, bug fixing, etc., are very easy in a C language program.	Readability, modifications, bug fixing, etc., are not easy in a Embedded C language program.

Compiler versus 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 specific target processor architecture (e.g. Intel x86/ Pentium). Here the operating system, the complier 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 (OS as well as target processor on which the OS is running). Compilers are generally termed as '*Native Compilers*'. A native compiler generates machine code for the same machine (processor) on which it is running.

Cross-compilers are software tools used in cross-platform development applications. In cross•platform development, the compiler running on a particular target processor/ OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS. Embedded system development is a typical example for cross•platform development where embedded firmware is developed on a machine with Intel/ AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. *8051*, *PIC*, *ARM*, etc.). Keil *C51* is an example for cross-compiler.

NOTE: The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications. Whenever you see the term 'Compiler' related to any embedded firmware application, please understand that it is referring the cross-compiler.

****

MODULE – 5

RTOS AND IDE FOR EMBEDDED SYSTEM DESIGN

IRTOS-BASED EMBEDDED SYSTEM DESIGN

The *super loop* based task execution model for firmware executes the tasks sequentially in order in which the tasks are listed within the loop. Here every task is repeated at regular intervals and the task execution is non-real time. Also, any response delay is acceptable and it will not create any operational issues or potential hazards.

But, certain applications demand time critical response to tasks/ events and delay in the response may be catastrophic. Examples: Flight control systems, Air bag control, Anti-lock Brake Systems (ABS) for vehicles, Nuclear monitoring devices, etc.

In embedded systems, the time critical response for tasks/ events may be addressed by –

- Assigning priority to tasks and execute the high priority task.
- Dynamically change the priorities of tasks, if required on a need basis.

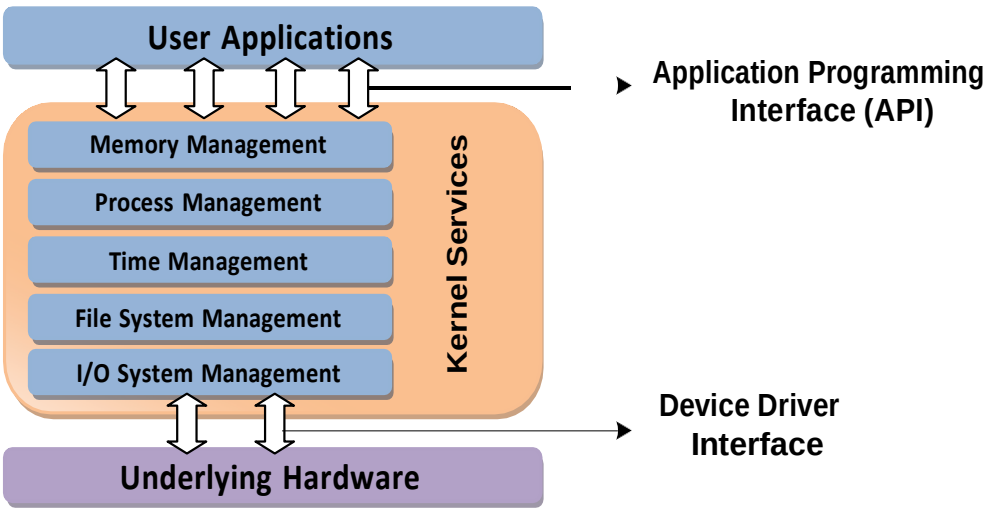
The introduction of operating system based firmware execution in embedded devices can address these needs to a greater extent.

OPERATING SYSTEM (OS) BASICS:

The *Operating System (OS)* acts as a bridge between the user applications/ tasks and the underlying system resources through a set of system functionalities and services. The primary functions of operating systems are

- Make the system convenient to use
- Organize and manage the system resources efficiently and correctly.

The following Figure gives an insight into the basic components of an operating system and their interfaces with rest of the world.



The Kernel:

The *kernel* is the core of the operating system. It is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications.

- Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services like memory management, process management, time management, file system management, I/O system management.

Process Management: The process management deals with managing the process/ tasks. Process management includes –

- setting up a memory for the process
- loading process code into memory
- allocating system resources
- scheduling and managing the execution of the process
- setting up and managing Process Control Block (PCB)
- inter process communication and synchronization
- process termination/ deletion, etc.

Primary Memory Management: Primary memory refers to a volatile memory (RAM), where processes are loaded and variables and shared data are stored.

The Memory Management Unit (MMU) of the kernel is responsible for –

- Keeping a track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis.

File System Management: File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/ video files, etc. A file system management service 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
- Providing automatic allocation of file space based on the amount of free running space available
- Providing flexible naming conversion for the files.

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, direct

access to I/O devices is not allowed; access to them is establish through Application Programming Interface (API). The kernel maintains list of all the I/O devices of the system. The service „*Device Manager*" of the kernel is responsible for handling all I/O related operations. The Device Manager is responsible for –

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device.

Secondary Storage Management: The secondary storage management deals with managing the secondary storage memory devices (if any) connected to the system. Secondary memory is used as backup medium for programs and data, as main memory is volatile. In most of the systems secondary storage is kept in disks (hard disks). The secondary storage management service of kernel deals with –

- Disk storage allocation
- Disk scheduling
- Free disk space management

Protection Systems: Modern operating systems are designed in such way to support multiple users with different levels of access permissions. The *protection* deals with implementing the security policies to restrict the access of system resources and particular user by different application or processes and different user.

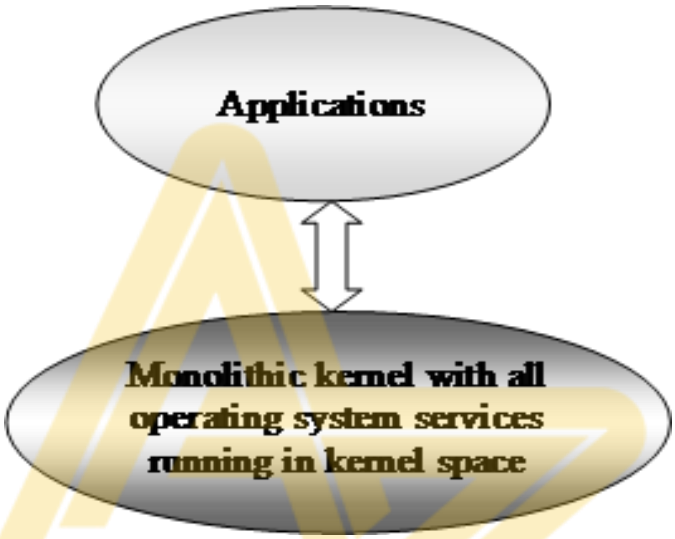
Interrupt Handler: Kernel provides interrupt handler mechanism for all external/ internal interrupt generated by the system.

The important services offered by the kernel of an OS:

- ***Kernel Space and User Space:*** The program code corresponding to the kernel applications/ services are kept in a contiguous area of primary (working) memory and is protected from the un-authorized access by user programs/ applications.
    - The memory space at which the kernel code is located is known as „*Kernel Space*". All user applications are loaded to a specific area of primary memory and this memory area is referred as „*User Space*".
    - The partitioning of memory into kernel and user space is purely Operating System dependent.
    - Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

Monolithic Kernel and Microkernel: Kernel forms the heart of OS. Different approaches are adopted for building an operating system kernel. Based on the kernel design, kernels can be classified into „*Monolithic*" and „*Micro*".

- ***Monolithic Kernel:*** In monolithic kernel architecture, all kernel services run in the kernel space. All kernel modules run within the same memory space under a single kernel thread.
- The major drawback of monolithic kernel 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.



- ***Microkernel:*** The microkernel design incorporates only essential set of operating system services into the kernel. The rest of the operating systems services are implemented in program known as „*Servers*" which runs in user space. The memory management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. The benefits of micro kernel based designs are –
    - *Robustness:* If a problem is encountered in any of the services, which runs as a server can be reconfigured and restarted without the restarting the entire OS. Here chances of corruption of kernel services are ideally zero.
    - *Configurability:* Any services, which runs as a server application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

TYPES OF OPERATING SYSTEMS:

Depending on the type of kernel and kernel services, purpose and type of computing system, Operating Systems are classified into different types.

General Purpose Operating System (GPOS):

The operating systems, which are deployed in general computing systems, are referred as *GPOS*. The GPOSs are often quite non-deterministic in behavior.

- Windows 10/8.x/XP/MS-DOS, etc., are examples of GPIOs.

Real Time Operating System (RTOS):

Real Time implies deterministic in timing behavior.

- RTOS services consumes only known and expected amounts of time regardless the number of services.
- RTOS implements policies and rules concerning time-critical allocation of a system‟s resources.
- RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
    - Windows Embedded Compact, QNX, VxWorks MicroC/OS-II, etc., are examples of RTOSs.

The Real-Time kernel: The kernel of a Real-Time OS is referred as Real-Time kernel. The Real-Time kernel is highly specialized and it contains only the minimal set of services required for running user applications/ tasks. The basic functions of a Real-Time kernel are listed below:

- Task/ Process management
- Task/ Process scheduling
- Task/ Process synchronization
- Error/ Exception handling
- Memory management
- Interrupt handling
- Time management.

- ***Task/ Process Management:*** Deals with setting up the memory space for the tasks, loading the task‟s code into the memory space, allocating system resources and 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. TCB usually 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: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.
- Task Priority: Task priority (E.g. 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)
- Other Parameters: Other relevant task parameters.

  - The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels based on the task management implementation.

- ***Task/ Process Scheduling:*** Deals with sharing the CPU among various tasks/ processes. A kernel application called „*Scheduler*" handles the task scheduling. Scheduler is an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

- ***Task/ Process Synchronization:*** Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

- ***Error/ Exception Handling:*** Deals with registering and handling the errors occurred/ exceptions rose 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 level.
    - *Deadlock* is an example for kernel level exception, whereas *timeout* is an example for a task level exception.
      - ✓ Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
      - ✓ Timeouts and retry are two techniques used together. The tasks retries an event/ message certain number of times; if no response is received after exhausting the limit, the feature might be aborted.
  - The OS kernel gives the information about the error in the form of a system call (API).

- ***Memory Management:*** The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.
    - In general, the memory allocation time increases depending on the size of the block of memory need to be allocated and the state of the allocated memory block. RTOS achieves predictable timing and deterministic behavior, by compromising the effectiveness of memory allocation.
    - RTOS generally uses „*block*" based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS. RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a „*Free buffer Queue*".
    - Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe mode* when an illegal memory access occurs.
    - The memory management function a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

- ***Interrupt Handling:*** Deals with the handling of various interrupts. 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. Usually the software interrupts fall under the *Synchronous Interrupt* category.
            - ✓ Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.
        - For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
        - Interrupts which occurs at any point of execution of any task, and are not in sync with the currently executing task are *Asynchronous interrupts*.
            - ✓ Timer overflow interrupts, serial data reception/ transmission interrupts etc., are examples for asynchronous interrupts.
        - For asynchronous interrupts, the interrupt handler is usually written as separate task (depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

- Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually. Most of the RTOS kernel implements „*Nested Interrupts*" architecture.

- ***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. Usually, the „Timer tick" varies in the microseconds range. The time parameters for tasks are expressed as the multiples of the „Timer tick".
  - The System time is updated based on the „Timer tick". If the System time register is 32 bits wide and the „Timer tick" interval is 1 microsecond, the System time register will reset in;

    $2^{32} * 10^{-6} / (24 * 60 * 60) = \sim 0.0497$ Days = 1.19 Hours
  - If the „Timer tick" interval is 1 millisecond, the System time register will reset in

    $2^{32} * 10^{-3} / (24 * 60 * 60) = 49.7$ Days = ~ 50 Days
  - The „Timer tick" interrupt is handled by the „*Timer Interrupt*" handler of kernel. The „Timer tick" interrupt can be utilized for implementing the following actions:
    - Save the current context (Context of the currently executing task)
    - Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register.
    - Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = „*count up*" and decrement registers with count direction setting = „*count down*")
    - Activate the periodic tasks, which are in the idle state
    - Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
    - Delete all the terminated tasks and their associated data structures (TCBs)
    - Load the context for the first task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the „*Timer Interrupt*" task.

- ***Hard Real-Time:*** A Real Time Operating Systems which strictly adheres 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 emphasize on the principle „A late answer is a wrong answer".
        - For example, Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems.
    - Most of the Hard Real Time Systems are automatic.

- ***Soft Real-Time:*** Real Time Operating Systems 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 if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
    - Soft real-time system emphasizes on the principle „A late answer is an acceptable answer, but it could have done bit faster".
    - Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

TASKS, PROCESSES 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 program or part of it in execution is also called a „*Process*".

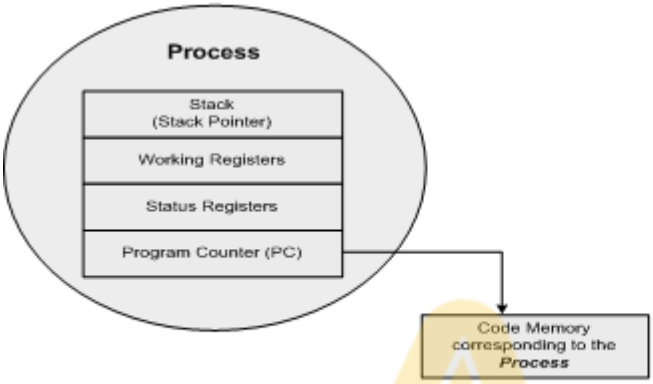
- The terms „*Task*", „*Job*" and „*Process*" refer to the same entity in the Operating System context and most often they are used interchangeably.

Process:

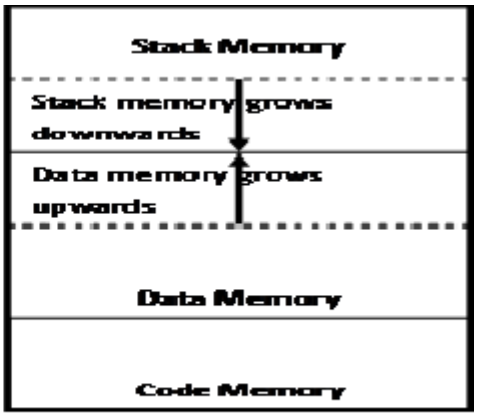
A „Process" is a program, or part of it, in execution. Process is also known as an instance of a program in execution. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.

- **Structure of a Processes:** The concept of „Process" leads to concurrent execution of tasks and thereby, efficient utilization 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, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualized as shown in the following Figure.



Structure of a Process

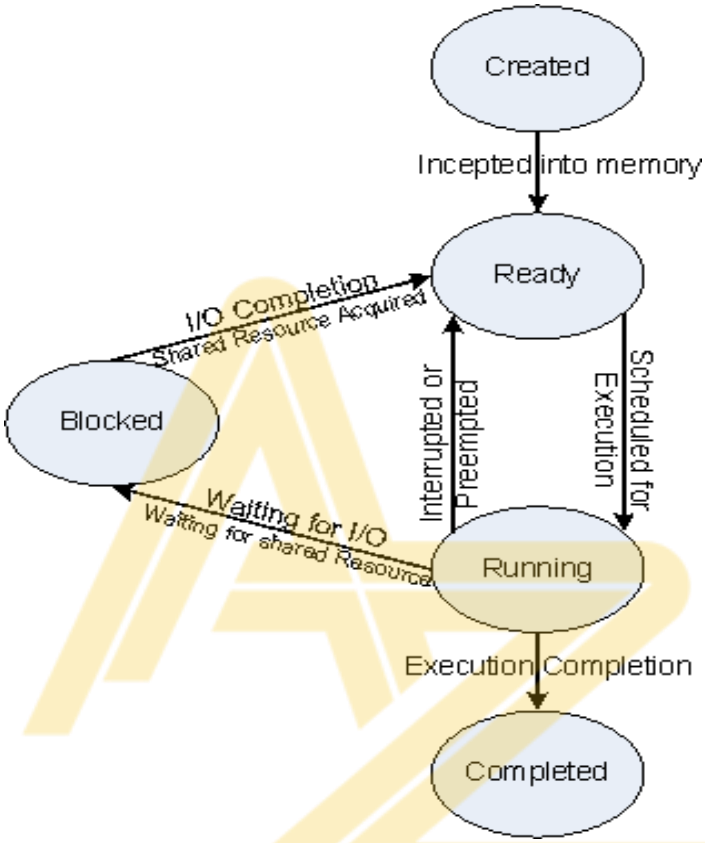


Memory Oraganization of a Process

- 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 process gets its turn, its registers and 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 (Figure, shown above).
  - The „*Stack*" memory holds all temporary data such as variables local to the process.
  - The „*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.

- ***Process States & 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.
- The transition of a process from one state to another is known as „*State transition*". The Process states and state transition representation are shown in the following Figure.



- *Created State:* The state at which a process is being created is referred as „Created State". The Operating System recognizes a process in the „Created State" but no resources are allocated to the process.
- *Ready State:* 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.
- *Running State:* The state where in the source code instructions corresponding to the process is being executed is called „Running State". Running state is the state at which the process execution happens.
- *Blocked State/ Wait State:* Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur (E.g. Waiting for user inputs such

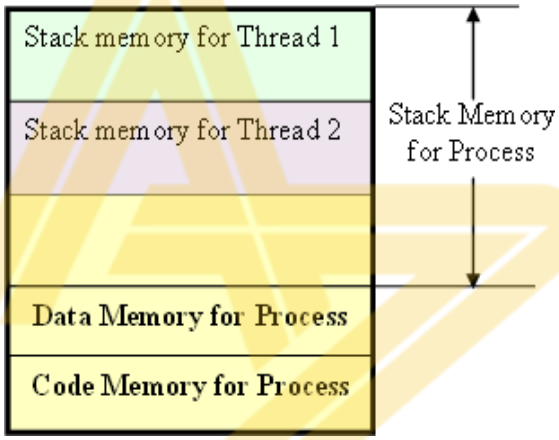
as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc.

- *Completed State:* A state where the process completes its execution.

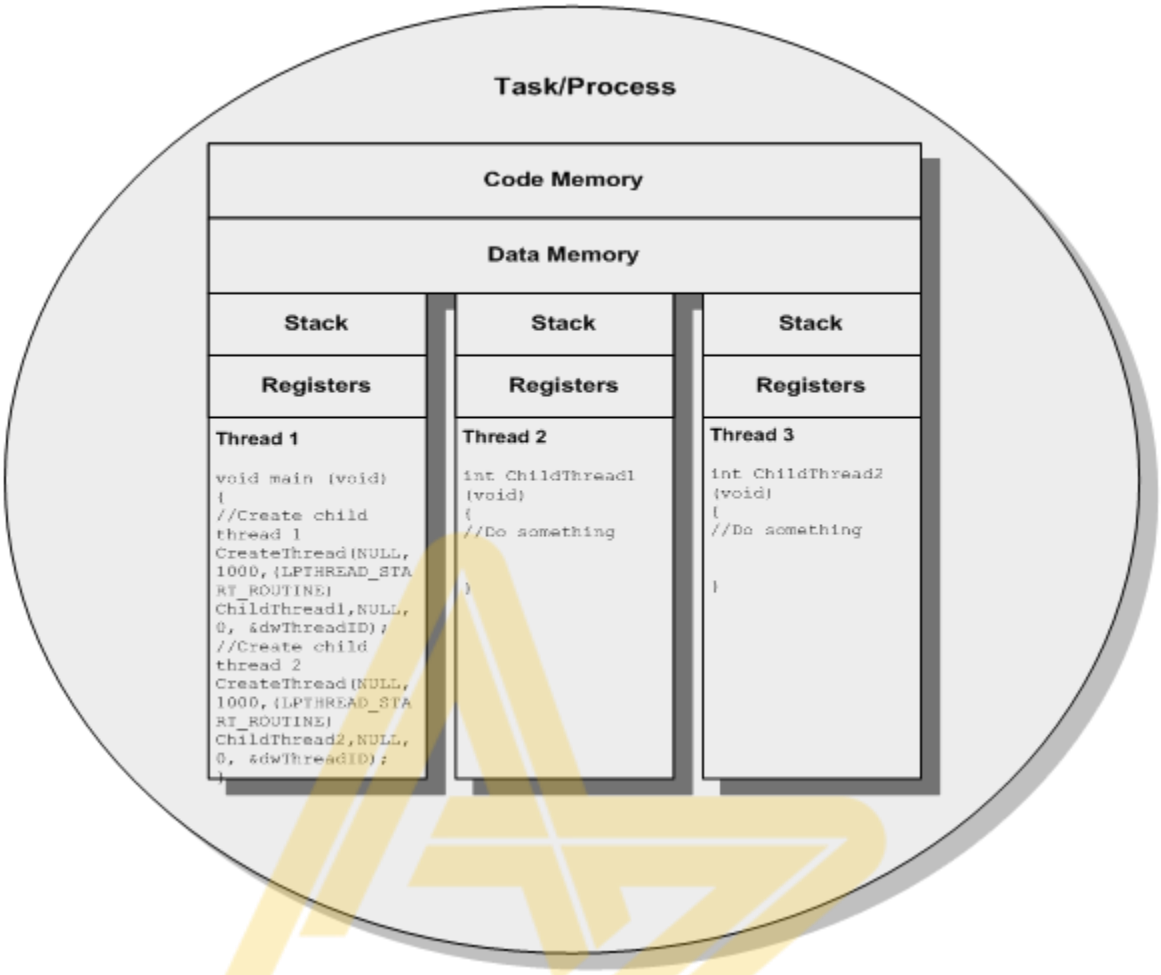
Threads:

A *thread* is the primitive that can execute code. A *thread* is a single sequential flow of control within a process. A *thread* is also known as lightweight process.

- A 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.
- 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 the following figure.



- ***The Concept of Multithreading:*** The process is split into multiple threads, which executes a portion of the process; there will be a main thread and rest of the threads will be created within the main thread.
  - The multithreaded architecture of a process can be visualized with the thread-process diagram, shown below.
  - Use of multiple threads to execute a process brings the following advantage:
    - *Better memory utilization:* Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
    - Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
    - Efficient CPU utilization. The CPU is engaged all time.



- ***Thread Standards:*** Thread standards deal with the different standards available for thread creation and management. These standards are utilized by the Operating Systems for thread creation and thread management. It is a set of thread class libraries. The commonly available thread class libraries are –
    - *POSIX Threads:* POSIX stands for Portable Operating System Interface. The POSIX.4 standard deals with the Real Time extensions and POSIX.4a standard deals with thread extensions. The POSIX standard library for thread creation and management is „*Pthreads*“. „*Pthreads*“ library defines the set of POSIX thread creation and management functions in „*C*“ language. (*Example 1 – Self study*).
    - *Win32 Threads:* Win32 threads are the threads supported by various flavors of Windows Operating Systems. The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions. Win32 threads are created with the API.
    - *Java Threads:* Java threads are the threads supported by Java programming Language. The java thread class „Thread“ is defined in the package „*java.lang*“. This package needs to be imported for using the thread creation functions supported by the Java thread class.

There are two ways of creating threads in Java: Either by extending the base „Thread" class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes.

- ***Thread Pre-emption:*** *Thread pre-emption* is the act of pre-empting the currently cunning thread (stopping temporarily). It is dependent on the Operating System. It is performed for sharing the CPU time among all the threads. The execution switching among threads are known as „*Thread context switching*". Threads falls into one of the following types:
  - *User Level Thread:* User level threads do not have kernel/ Operating System support and they exist only in the running process. A process may have multiple user level threads; but the OS threats it as single thread and will not switch the execution among the different threads of it. It is the responsibility of the process to schedule each thread as and when required. Hence, user level threads are non-preemptive at thread level from OS perspective.
  - *Kernel Level/ System Level Thread:* Kernel level threads are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by the OS.
    - The execution switching (thread context switching) of user level threads happen only when the currently executing user level thread is voluntarily blocked. Hence, no OS intervention and system calls are involved in the context switching of user level threads. This makes context switching of user level threads very fast.
    - Kernel level threads involve lots of kernel overhead and involve system calls for context switching. However, kernel threads maintain a clear layer of abstraction and allow threads to use system calls independently.
    - There are many ways for binding user level threads with kernel/ system level threads; which are explained below:
      - Many-to-One Model: Many user level threads are mapped to a single kernel thread. The kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU. Solaris Green threads and GNU Portable Threads are examples for this.

- One-to-One Model: Each user level thread is bonded to a kernel/ system level thread. Windows XP/NT/2000 and Linux threads are examples of One-to-One thread models.
- Many-to-Many Model: In this model many user level threads are allowed to be mapped to many kernel threads. Windows NT/2000 with *ThreadFiber* package is an example for this.

- ***Thread versus Process:***

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heap memory.	Process has its own code memory, data memory, and stack memory.
A thread cannot live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process; the first (main) thread calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory; each thread holds separate memory area for stack.
Threads are very inexpensive to create.	Processes are very expensive to create; involves many OS overhead.
Context switching is inexpensive and fast.	Context switching is complex and involves lots of OS overhead and comparatively slow.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resource allocated to it are reclaimed by the OS and all associated threads of the process also dies.

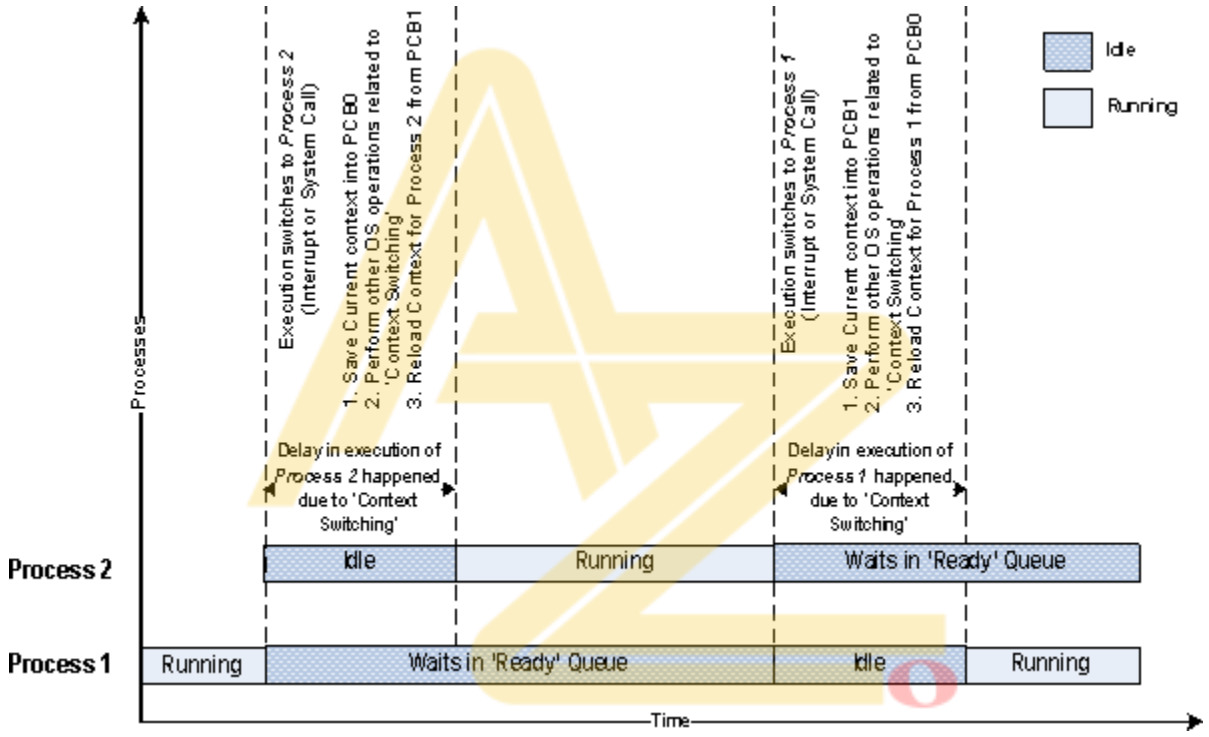
MULTIPROCESSING AND MULTITASKING:

The ability to execute multiple processes simultaneously is referred as *multiprocessing*. 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 System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uni-processor system, it is not possible to execute multiple processes simultaneously.

Multitasking refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process.

- Multitasking involves „*Context switching*" (see the following Figure), „*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 (details like Register details, Memory details, System Resource Usage details, Execution details, etc.) for the currently running processes 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*".



Types of Multitasking:

Depending on how the task/ process execution switching act is implemented, multitasking can is classified into –

- ***Co-operative Multitasking:*** Co-operative multitasking is the most primitive form of multitasking in which a task/ process gets a chance to execute only when the currently executing task/ process voluntarily relinquishes the CPU. In this method, any task/ process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

- ***Preemptive Multitasking:*** Preemptive multitasking ensures that every task/ process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/ process priority.
- ***Non-preemptive Multitasking:*** The process/ task, which is currently given the CPU time, is allowed to execute until it terminates (enters the „*Completed*" state) or enters the „*Blocked/ Wait*" state, waiting for an I/O. The co-operative and non-preemptive multitasking differs in their behavior when they are in the „Blocked/Wait" state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the „*Blocked/ Wait*" sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

TASK COMMUNICATION:

In a multitasking system, multiple tasks/ processes run concurrently (in pseudo parallelism) and each process may or may not interact between. Based on the degree of interaction, the processes/ tasks running on an OS are classified as –

- ***Co-operating Processes:*** In the co-operating interaction model, one process requires the inputs from other processes to complete its execution.
- ***Competing Processes:*** The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.
    - The co-operating processes exchanges information and communicate through the following methods:
        - *Co-operation through sharing:* Exchange data through some shared resources.
        - *Co-operation through Communication:* No data is shared between the processes. But they communicate for execution synchronization.

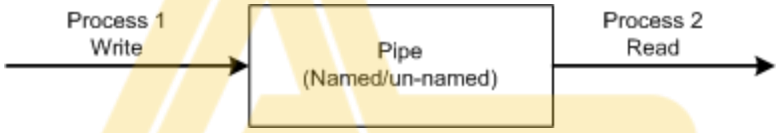
The mechanism through which tasks/ processes communicate each other is known as *Inter Process/ Task Communication (IPC)*. IPC is essential for process co-ordination. The various types of IPC mechanisms adopted by process are kernel (Operating System) dependent. They are explained below.

IPC Mechanism - Shared Memory:

Processes share some area of the memory to communicate among them (see the following Figure). Information to be communicated by the process is written to the shared memory area. Processes which require this information can read the same from the shared memory area.

- The implementation of shared memory is kernel dependent. Different mechanisms are adopted by different kernels for implementing this, a few among are s follows:

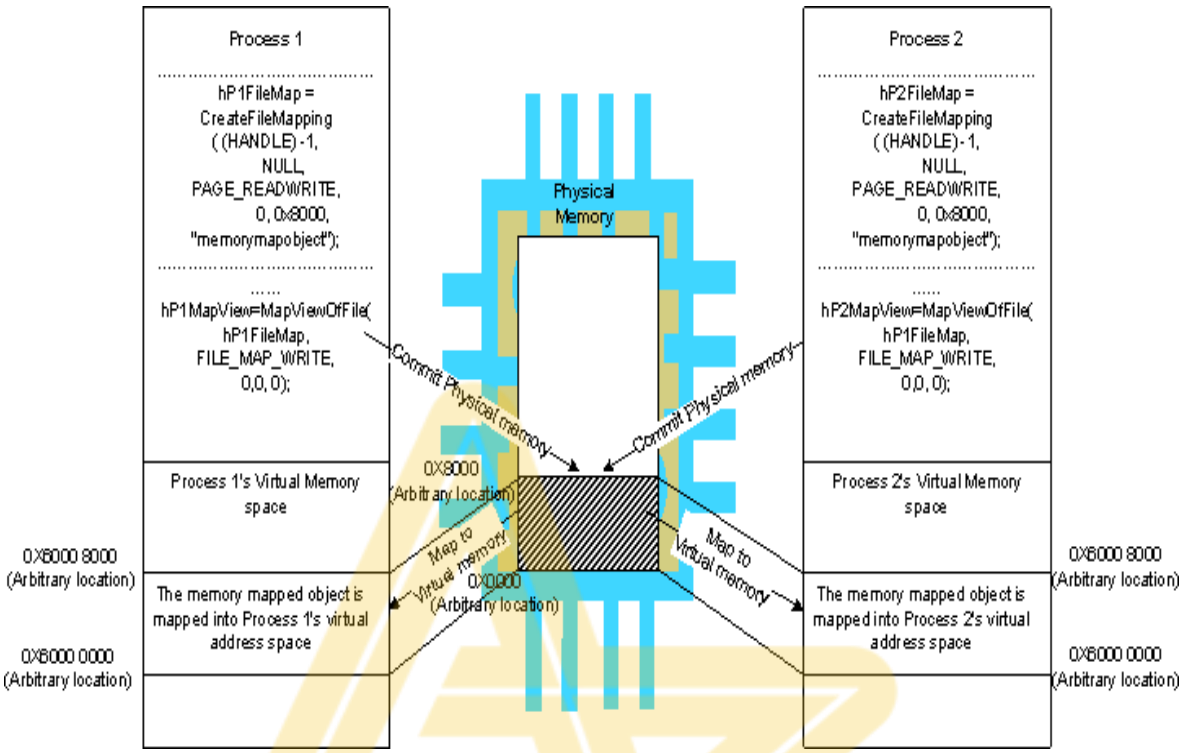
1. ***Pipes:*** „*Pipe*" is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as *pipe server* and a process which connects to a pipe is known as *pipe client*. A pipe can be considered as a medium for information flow and has two conceptual ends. It can be *unidirectional*, allowing information flow in one direction or *bidirectional* allowing bi-directional information flow. A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualized as



   - The implementation of „Pipes" is OS dependent. Microsoft® Windows Desktop Operating Systems support two types of „Pipes" for Inter Process Communication. Namely;
     - *Anonymous Pipes:* The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.
     - *Named Pipes:* Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server. A process which connects to the named pipe is known as pipe client. With named pipes, any process can act as both client and server allowing point-to-point communication. Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

2. ***Memory Mapped Objects:*** Memory mapped object is a shared memory technique adopted by certain Real Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously. In this approach, a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space. All read and write operation to this virtual address space by a process is directed to its committed physical area.

Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data. The concept of memory mapped object is shown bellow.



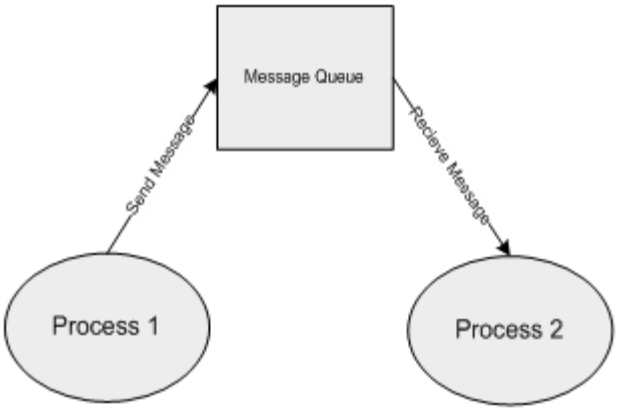
IPC Mechanism - Message Passing:

Message passing is a/ an synchronous/ asynchronous information exchange mechanism for Inter Process/ Thread Communication. The major difference between shared memory and message passing technique is

- Through shared memory lots of data can be shared whereas only limited amount of info/ data is passed through message passing.
- Message passing is relatively fast and free from the synchronization overheads compared to shared memory.

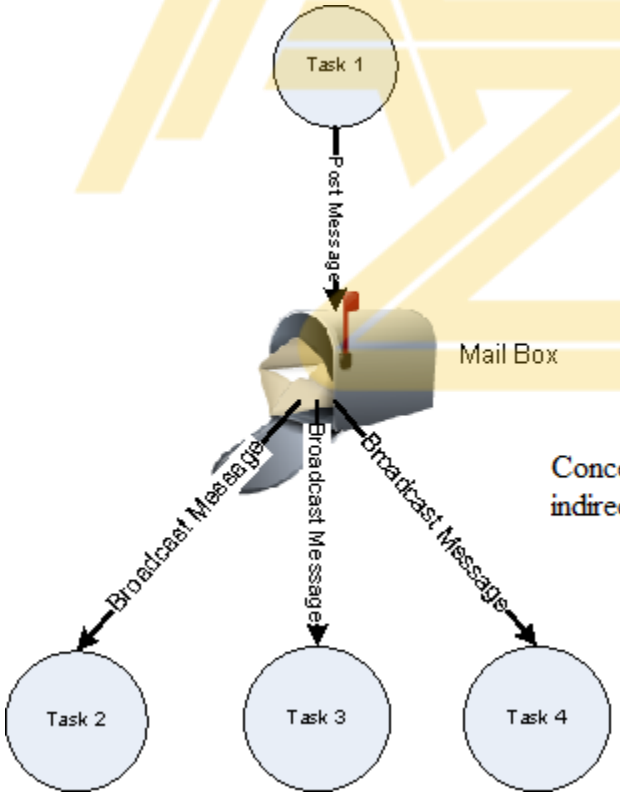
Based on the message passing operation between the processes, message passing is classified into –

1. ***Message Queues:*** Process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called „*Message queue*“, which stores the messages temporarily in a system defined memory object, to pass it to the desired process. Messages are sent and received through *send (Name of the process to which the message is to be sent, message*) and *receive* (*Name of the process from which the message is to be received, message*) methods. The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent.



Concept of message queue based indirect messaging for IPC

2. ***Mailbox:*** Mailbox is a special implementation of message queue. Usually used for one way communication, only a single message is exchanged through mailbox whereas „message queue" can be used for exchanging multiple messages. One task/process creates the mailbox and other tasks/process can subscribe to this mailbox for getting message notification. The implementation of the mailbox is OS kernel dependent. The MicroC/ OS-II RTOS implements mailbox as a mechanism for inter task communication

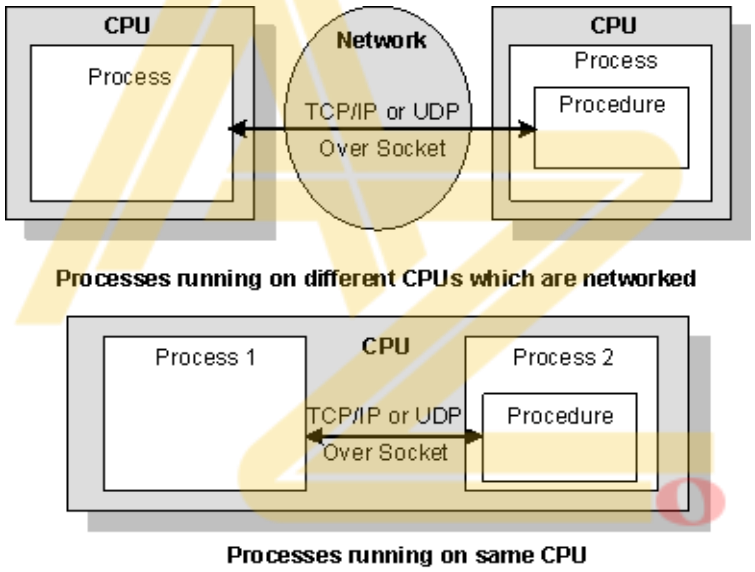


Concept of Mailbox based indirect messaging for IPC

3. ***Signalling:*** Signals are used for an asynchronous notification mechanism. The signal mainly used for the execution synchronization of tasks process/ tasks. Signals do not carry any data and are not queued. The implementation of signals is OS kernel dependent and VxWorks RTOS kernel implements „signals" for inter process communication.

IPC Mechanism - Remote Procedure Call (IPC) and Sockets: Remote Procedure Call is the Inter Process Communication (IPC) mechanism used by a process, to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In the object oriented language terminology, RPC is also known as *Remote Invocation* or *Remote Method Invocation* (*RMI*). The CPU/ process containing the procedure which needs to be invoked remotely is known as server. The CPU/ process which initiates an RPC request is known as client.

- In order to make the RPC communication compatible across all platforms, it should stick on to certain standard formats.
- Interface Definition Language (IDL) defines the interfaces for RPC. Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms.
- The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking).



Concept of Remote Procedure Call (RPC) for IPC

Sockets are used for RPC communication. ***Socket*** is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application. Sockets are of different types namely; Internet sockets (INET), UNIX sockets, etc.

- The *INET Socket* works on Internet Communication protocol. TCP/ IP, UDP, etc., are the communication protocols used by INET sockets.
- INET sockets are classified into:
    - *Stream Sockets:* are connection oriented and they use TCP to establish a reliable connection.
    - *Datagram Sockets:* rely on UDP for establishing a connection.

TASK SYNCHRONIZATION:

In a multitasking environment, multiple processes run concurrently and share the system resources. Also, each process may communicate with each other with different IPC mechanisms. Hence, there may be situations that; two processes try to access a shared memory area, where one process tries to write to the memory location when the other process is trying to read from the same memory location. This will lead to unexpected results.

The solution is, make each process aware of access of a shared resource. The act of making the processes aware of the access of shared resources by each process to avoid conflicts is known as "*Task/ Process Synchronization*".

Task/ Process Synchronization is essential for –

1. Avoiding conflicts in resource access (racing, deadlock, etc.) in multitasking environment.
2. Ensuring proper sequence of operation across processes.
3. Establish proper communication between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource is known as „Critical Section". In order to synchronize the access to shared resources, the access to the critical section should be exclusive.

Task Communication/ Synchronization Issues:

Various synchronization issues may arise in a multitasking environment, if processes are not synchronized properly in shared resource access, such as:

1. ***Racing:*** Look into the following piece of code:

```
#include <stdio.h>
//*****************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two processes Process A and Process B.
char Buffer [10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//*****************************************************************
// Process A
Void Process_A (void)
{
        int i;
        for (i =0; i<5; i++)
        {
                if (Buffer [i] > 0)
```

```
                counter++;
        }
}
//*****************************************************************
// Process B
Void Process_B (void)
{
        int j;
        for (j =5; j<10; j++)
        {
                if (Buffer[j] > 0)
                counter++;
        }
}
//*****************************************************************
//Main Thread.
int main()
{
        DWORD id;
        CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE) Process_A,
(LPVOID) 0, 0, &id);
        CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE) Process_B,
(LPVOID) 0, 0, &id);
        Sleep (100000);
        return 0;
}
```

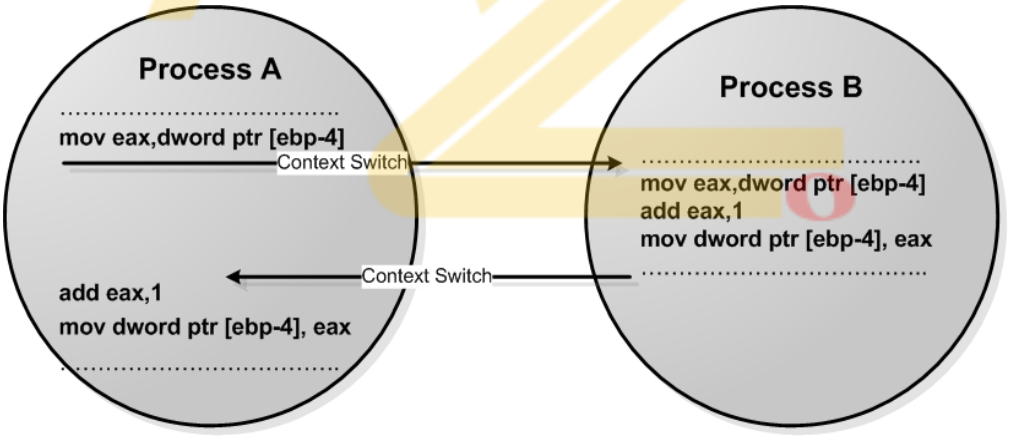
- From a programmer perspective, the value of counter will be 10 at the end of execution of processes A & B. But it need not be always.
  - The program statement *counter++;* looks like a single statement from a high level programming language (C Language) perspective. The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the high level program statement *counter++;* under Windows XP operating system running on an Intel Centrino Duo processor is given below.

```
mov eax, dword ptr [ebp-4]      ;Load counter in Accumulator
add eax, 1                      ; Increment Accumulator by 1
mov dword ptr [ebp-4], eax      ;Store counter with Accumulator
```

- At the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX Register). The memory variable counter is represented using a pointer. The base pointer register (EBP Register) is used for pointing to the memory variable counter. After loading the contents of the variable counter to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes; Process A and Process B contain the program statement *counter++;* Translating this into the machine instruction.

Process A	Process B
mov eax,dword ptr [ebp-4]	mov eax, dword ptr [ebp-4]
add eax, 1	add eax, 1
mov dword ptr [e bp-4], eax	mov dword ptr [ebp-4], eax

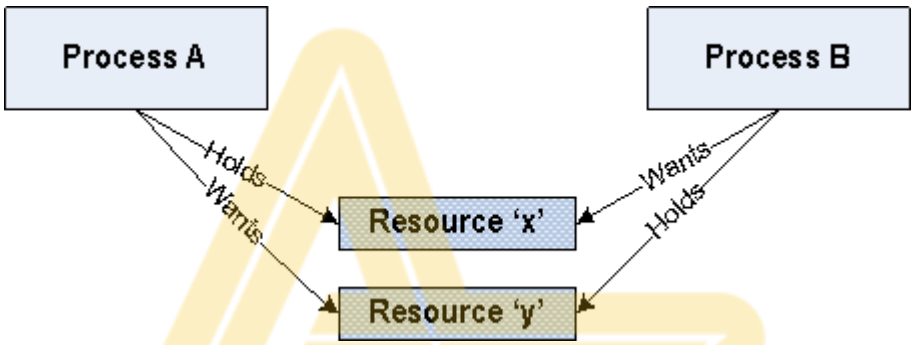
- Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the *counter++;* statement. Process A accomplishes the *counter++;* statement through three different low level instructions. Now imagine that the process switching happened at the point, where Process A executed the low level instruction *mov eax, dword ptr [ebp-4]* and is about to execute the next instruction *add eax, 1*. The scenario is illustrated in the following Figure.



- Process B increments the shared variable „counter" in the middle of the operation where Process A tries to increment it. When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers *eax* and *ebp* for executing *counter++;* instruction, the original content of these registers will be saved as part of context saving and it will be retrieved back as part of the context retrieval, when Process A gets the CPU for execution. Hence the content of *eax* and *ebp* remains intact irrespective of context switching). Though the variable counter is incremented by Process B,

Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the variable counter.

2. ***Deadlock:*** *Deadlock* is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process; hence, none of the processes are able to make any progress in their execution.
   - Process A holds a resource „*x*" and it wants a resource „*y*" held by Process B. Process B is currently holding resource „*y*" and it wants the resource „*x*" which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes.



   - ***Conditions Favoring Deadlock:***
     - *Mutual Exclusion:* The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display device in an embedded device.
     - *Hold & Wait:* The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
     - *No Resource Preemption:* The criteria that Operating System cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.
     - *Circular Wait:* A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general there exists a set of waiting process P0, P1 …. Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, ……,Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on… This forms a circular wait queue.
   - ***Handling Deadlock:*** The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

- *Ignore Deadlocks:* Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.
- *Detect and Recover:* This approach suggests the detection of a deadlock situation and recovery from it.
    - This is similar to the deadlock condition that may arise at a traffic junction. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as „*back up cars*" technique.
    - Operating Systems keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analyzing the resource graph by graph analyzer algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

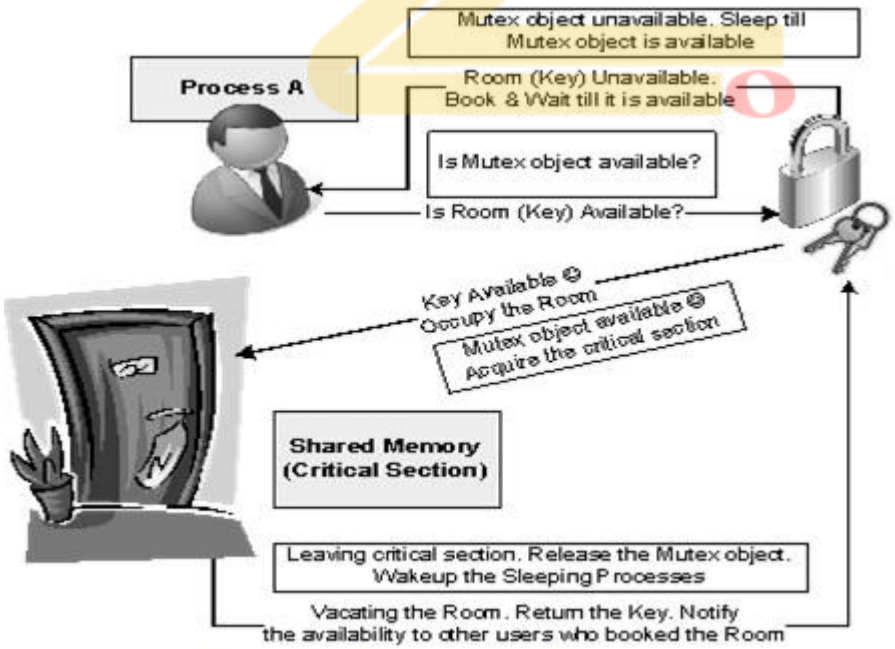
- ***Avoid Deadlocks:*** Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.
- ***Prevent Deadlocks:*** Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.
- Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/ guidelines in allocating resources to processes.
    1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
    2. Grant resource allocation requests from processes only if the process does not hold a resource currently.
- Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/ guidelines in resources allocation and releasing:
    1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfill immediately.

2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Task Synchronization Techniques:

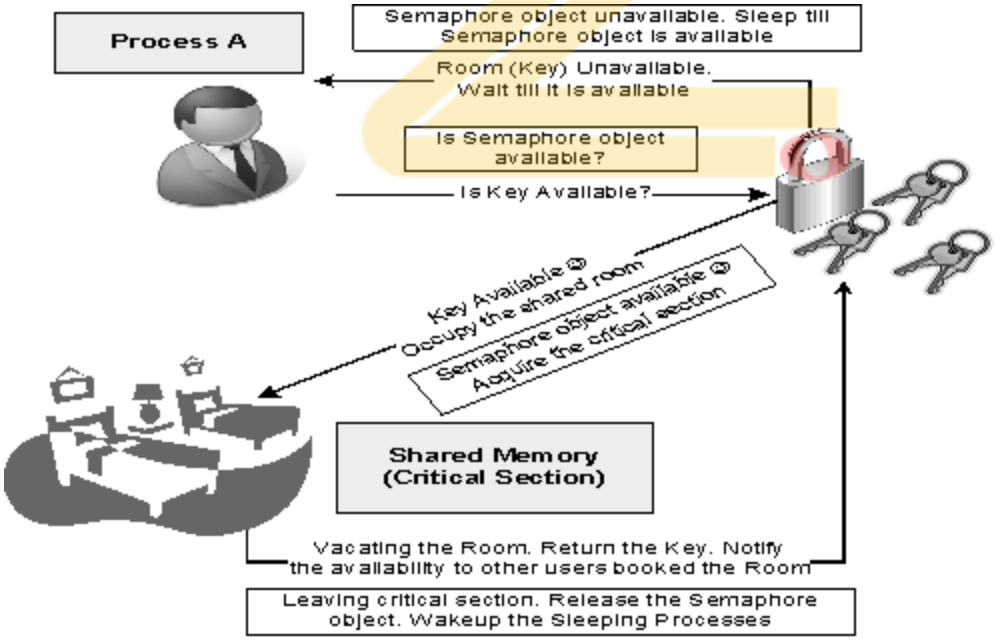
The technique used for task synchronization in a multitasking environment is *mutual exclusion*. Mutual exclusion blocks a process. Based on the behavior of blocked process, mutual exclusion methods can be classified into two categories: Mutual exclusion through busy waiting/ spin lock & Mutual exclusion through sleep & wakeup.

- ***Semaphore:*** *Semaphore* is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource; and a process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently in use by it.
- The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time.
- The display device of an embedded system is a typical example of a shared resource which needs exclusive access by a process. The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes.
- Based on the implementation, Semaphores can be classified into *Binary Semaphore* and *Counting Semaphore*.



The concept of Binary Semaphore

- ***Binary Semaphore:*** Implements exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being used by a process.
  - „Only one process/ thread" can own the binary semaphore at a time.
  - The state of a „binary semaphore" object is set to signaled when it is not owned by any process/ thread, and set to non-signaled when it is owned by any process/ thread.
  - The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as mutex.
- ***Counting Semaphore:*** Maintains a count between zero and a maximum value. It limits the usage of resource by a fixed number of processes/ threads.
- The count associated with a „*Semaphore object*" is decremented by one when a process/ thread acquires it and the count is incremented by one when a process/ thread releases the „Semaphore object".
- The state of the counting semaphore object is set to „signaled" when the count of the object is greater than zero.
- The state of the „Semaphore object" is set to non-signaled when the semaphore is acquired by the maximum number of processes/ threads that the semaphore can support (i.e. when the count associated with the „Semaphore object" becomes zero).
- The creation and usage of „counting semaphore object" is OS kernel dependent.



The concept of Counting Semaphore

HOW TO CHOOSE AN RTOS:

The decision of choosing an RTOS for an embedded design is very crucial. A lot of factors need to be analyzed carefully before making a decision on the selection of an RTOS. These factors can be either *functional* or *non-functional*.

Functional Requirements:

- *Processor Support:* It is not necessary that all RTOS"s support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.
- *Memory Requirements:* The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.
- *Real-time Capabilities:* It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems-are 'Real-time' in behavior. The task/ process scheduling policies play an important role in the 'Real-time' behavior of an OS. Analyze the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.
- *Kernel and Interrupt Latency:* The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.
- *Inter Process Communication and Task Synchronization:* The implementation of Inter Process Communication and Synchronization is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.
- *Modularization Support:* Most of the operating systems provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports moclularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.
- *Support for Networking and Communication:* The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.
- *Development Language Support:* Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customized for the Operating System is essential for running java applications. Similarly

the .NET Compact Framework (.NETCF) is required for running Microsoft .NET applications on top of the Operating System. The OS may include these components as built-in component, if not; check the availability of the same from a third party vendor or the OS under consideration.

Non-functional Requirements:

- *Custom Developed or Off the Shelf:* Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customizing an Open source OS. The decision on which to select is purely de• pendent on the development cost, licensing fees for the OS, development time and availability of skilled resources.
- *Cost*: The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.
- *Development and Debugging Tools Availability:* The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.
- *Ease of Use:* How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.
- *After Sales:* For a commercial embedded RTOS, after sales in the fom1 of e-mail, on-call services etc., for bug fixes, critical patch updates and support for production issues, etc., should be analyzed thoroughly.

INTREGRATION AND TESTING OF EMBEDDED HARDWARE AND FIRMWARE

Integration testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development.

- The final embedded hardware constitute of a PCB with all necessary components affixed to it as per the original schematic diagram.
- Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirements on the product. Embedded firmware will be in a target processor/ controller understandable format called machine language (sequence of ls and 0s-Binary).

- The target embedded hardware without embedding the firmware is a dumb device and cannot function properly. If you power up the hardware without embedding the firmware, the device may behave in an unpredicted manner.
- Both embedded hardware and firmware should be independently tested (Unit Tested) to ensure their proper functioning.
- Functioning of individual hardware sections can be done by writing small utilities which checks the operation of the specified part.
- The functionalities of embedded firmware can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE. By simulating the firmware, the memory contents, register details, status of various flags and registers can easily be monitored and it gives an approximate picture of "What happens inside the processor/ controller and what are the states of various peripherals" when the firmware is running on the target hardware. The IDE gives necessary support for simulating the various inputs required from the external world, like inputting data on ports, generating an interrupt condition, etc.

INTEGRATION OF HARDWARE AND FIRMWARE:

Integration of hardware and firmware deals with the embedding of firmware into the target hardware board. It is the process of '*Embedding Intelligence*' to the product.

- The embedded processors/ controllers used in the target board may or may not have built in code memory. For non-operating system based embedded products, if the processor/ controller contain internal memory and the total size of the firmware is fitting into the code memory area, the code memory is downloaded into the target controller/ processor.
- If the processor/ controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/ controller, an external dedicated EPROM/ FLASH memory chip is used for holding the firmware. This chip is interfaced to the processor/ controller.

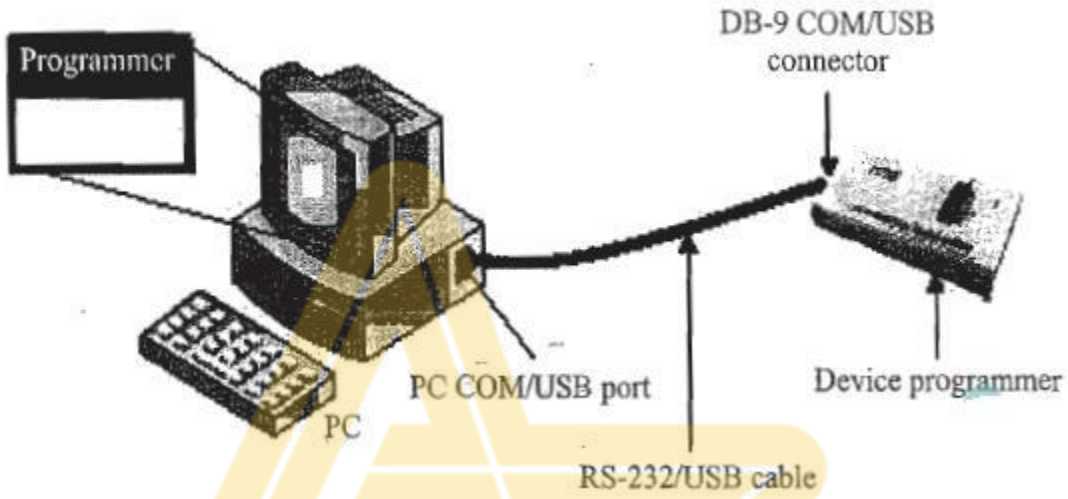
A variety of techniques are used for embedding the firmware into the target board. The commonly used firmware embedding techniques for a non-OS based embedded system are explained below. The non-OS based embedded systems store the firmware either in the on-chip processor/ controller memory or off-chip memory (FLASHI/ NVRAM, etc.).

Out-of-Circuit Programming:

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

The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals. Most of the programming devices available in the market are capable of programming different family of devices.

The programming device will be under the control of a utility program running on a PC. Usually the programming device is interfaced to the PC through RS-232C/USB/Parallel Port Interface. The commands to control the programmer are sent from the utility program to the programmer through the interface (see the following Figure).



The sequence of operations for embedding the firmware with a programmer is listed below:

1. Connect the programming device to the specified port of PC (USB/COM port/Parallel port)
2. Power up the device (Most of the programmers incorporate LED to indicate Device power up. Ensure that the power indication LED is ON)
3. Execute the programming utility on the PC and ensure proper connectivity is established between PC and programmer. In case of error turn off device power and try connecting it again
4. Unlock the ZIF socket by turning the lock pin
5. Insert the device to be programmed into the open socket as per the insert diagram shown on the programmer
6. Lock the ZIF socket
7. Select the device name from the list of supported devices
8. Load the hex file which is to be embedded into the device
9. Program the device by 'Program' option of utility program
10. Wait till the completion of programming operation (Till busy LED of programmer is off)
11. Ensure that programming is success by checking the status LED on the programmer (Usually 'Green' for success and 'Red' for error condition) or by noticing the feedback from the utility program
12. Unlock the ZIF socket and d take the device out of programmer.

Now the firmware is successfully embedded into the device. Insert the device into the board, power up the board and test it for the required functionalities. It is to be noted that the most of programmers support only Dual Inline Package (DIP) chips, since its ZIF socket is designed to accommodate only DIP chips.

Option for setting firmware protection will be available on the programming utility. If you really want the firmware to be protected against unwanted external access, and if the device is supporting memory protection, enable the memory protection on the utility before programming the device.

The programmer usually erases the existing content of the chip before programming the chip. Only EEPROM and FLASH memory chips are erasable by the programmer.

The major drawback of out-of-circuit programming is the high development time. Whenever the firmware is changed, the chip should be taken out of the development board for re-programming. This is tedious and prone to chip damages due to frequent insertion and removal.

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

In System Programming (ISP):

With ISP, programming is done 'within the system', meaning the firmware is embedded into the target device without removing it from the target board. It is the most flexible and easy way of firmware embedding. The only pre-requisite is that the target device must have an ISP support. Apart from the target board, PC, ISP cable and ISP utility, no other additional hardware is required for ISP.

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 will be in a serial format. The serial protocols used for ISP may be '*Joint Test Act Group (JTAG)*' or '*Serial Peripheral Interface (SPI)*' or any other proprietary protocol.

In System Programming with SPI Protocol: Devices with SPI (Serial Peripheral Interface) ISP (In System Programming) support contains a built-in SPI interface and the on-chip EEPROM or FLASH memory. The primary I/O lines involved in SPI-In System Programming are listed below:

MOSI – Master Out Slave In

MISO – Master In Slave Out

SCK – System Clock

RST – Reset of Target Device

GND – Ground of Target Device

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

acts as the clock for data transfer. A utility program can be developed on the PC side to generate the above signal lines.

Standard SPI-ISP utilities are feely available on the internet and, there is no need for going for writing own program. For ISP operations, the target device needs to be powered up in a pre-defined sequence. The power up sequence for In System Programming for Atmel's AT89S series microcontroller family is listed below:

1. Apply supply voltage between VCC and GND pins of target chip
2. Set RST pin to "HIGH" state
3. If a crystal is not connected across pins XTAL 1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTALl pin and wait for at least 10 milliseconds
4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/ Pl.5. The frequency of the shift clock supplied at pin SCK/ P1.7 needs to be less than the CPU clock at XTALl divided by 40
5. The Code or Data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is first erased before the new data is written. The write cycle is self-timed and typically takes less than 2.5 ms at 5V
6. Any memory location can be verified by using the Read instruction, which returns the content at the selected address at serial output MISO/ Pl.6
7. After successfully programming the device, set RST pin low or turn off the chip power supply and turn it ON to commence the normal operation.

The key player behind ISP is a factory programmed memory (ROM) called '*Boot ROM*'. The Boot ROM normally resides at the top end of code memory space and it varies in the order of a few Kilo Bytes (For a controller with 64K code memory space and lK Boot ROM, the Boot ROM resides at memory location FC00H to FFFFH). It contains a set of Low-level Instruction APIs and these APIs allow the processor/ controller to perform the FLASH memory programming, erasing and Reading operations. The contents of the Boot ROM are provided by the chip manufacturer and the same is masked into every device.

In Application Programming (IAP):

In Application Programming is a technique used by the firmware running on the target device for modifying a selected portion of the code memory. It is not a technique for first time embedding of user written firmware. It modifies the program code memory under the control of the embedded application.

Updating calibration data, look-up tables, etc., which are stored in code memory, are typical examples of IAP.

Use of Factory Programmed Chip:

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

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

Firmware Loading for Operating System Based Devices:

The OS based embedded systems are programmed using the In System Programming (ISP) technique. OS based embedded systems contain a special piece of code called '*Boot loader*' program which takes control of the OS and application firmware embedding and copying of the OS image to the RAM of the system for execution.

The 'Boot 1oader' for such embedded systems comes as pre-loaded or it can be loaded to the memory using the various interface supported like JTAG. The boot loader contains necessary driver initialization implementation for initializing the supported interfaces like UART/ I2C, TCP/ IP, etc. Boot loader implements menu options for selecting the source for OS image to load (Typical menu item examples are Load from FLASH ROM, Load from Network, Load through UART, etc).

Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

BOARD BRING UP:

Once the firmware is embedded into the target board using one of the programming techniques, then power up the board. You may be expecting the device functioning exactly in a way as you designed. But in real scenario it need not be and if the board functions well in the first attempt itself you are very lucky. Sometimes the first power up may end up in a messy explosion leaving the smell of burned components behind. It may happen due to various reasons, like Proper care was not taken in applying the power and power applied in reverse polarity (+ve of supply connected to –ve of the target board and vice versa), components were not placed in the correct polarity order (E.g. a capacitor on the target board is connected to the board with +ve terminal to –ve of the board and vice versa), etc ... etc ...

The prototype/ evaluation/ production version must pass through a varied set of tests to verify that embedded hardware and firmware functions as expected. *Bring up* process includes –

- basic hardware spot checks/ validations to make sure that the individual components and busses/ interconnects are operational – which involves checking power, clocks, and basic functional connectivity;
- basic firmware verification to make sure that the processor is fetching the code and the firmware execution is happening in the expected manner;
- running advanced validations such as memory validations, signal integrity validation, etc.

THE EMBEDDED SYSTEM DEVELOPMENT ENVIRONMENT

The embedded system development environment consists of –

- Development Computer (PC) or Host – acts as the heart of the development environment
- Integrated Development Environment (IDE) Tool – for embedded firmware development and debugging
- Electronic Design Automation (IDA) Tool – for embedded hardware design
- An emulator hardware – for debugging the target board
- Signal sources (like CRO, Multimeter, Logic Analyzer, etc.)
- Target hardware.

THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE):

In embedded system development context, *Integrated Development Environment (IDE)* stands for an integrated environment for developing and debugging the target processor specific embedded firmware. IDE is a software package which bundles –

- a "Text Editor (Source Code Editor)",
- "Cross-complier (for cross platform development and complier for same platform development)",
- "Linker", and
- a "Debugger".

Some IDEs may provide –

- interface to target board emulators,
- target processor"s/ controller"s Flash memory programmer, etc.

IDE may be command line based or GUI based.

NOTE: The Keil µVision IDE & An Overview of IDEs – lest as an exercise/ self study topic.

DISASSEMBLER/ DECOMPLIER:

Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/ instructions. The process of converting machine codes into Assembly code is known as *'Disassembling*'. In operation, disasseri1bling is complementary to assembling/ cross-assembling.

Decompiler is the utility program for translating machine codes into corresponding high level language instructions. Decompiler performs the reverse operation of compiler/ cross-compiler.

The *disassemblers/ decompilers* for different family of processors/ controllers are different. Disassemblers/ Decompilers are deployed in reverse engineering. *Reverse engineering* is the process of revealing the technology behind the working of a product. Reverse engineering in Embedded Product development is employed to find out the secret behind the working of popular proprietary products. Disassemblers /decompilers help the reverse engineering process by translating the embedded firmware into Assembly/ high level language instructions.

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

It is not possible for a disassembler/ decompiler to generate an exact replica of the original assembly code/ high level source code in terms of the symbolic constants and comments used. However disassemblers/ decompilers generate a source code which is somewhat matching to the original source code from which the binary code is generated.

SIMULATORS, EMULATORS AND DEBUGGING:

Simulators and *emulators* are two important tools used in embedded system development.

- *Simulator* is a software tool use for simulating the various conditions for checking the functionality of the application firmware. The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality. In certain scenarios, simulator refers to a soft model (GUI model) of the embedded product.
    - 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 as given in the end product can be developed in software. Soft phone is an example for such a simulator.
- *Emulator* is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.

Simulators:

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

The features of simulator based debugging are listed below.

1. Purely software based
2. Doesn't require a real target system
3. Very primitive (Lack of featured I/O support. Everything is a simulated one)

4. Lack of Real-time behavior.

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

- *No Need for Original Target Board:* Simulator based debugging technique is purely software oriented. IDE's software support simulates the CPU of the target board. User only needs to know about the memory map of various devices within the target board and the firmware should be written on the basis of it. Since the real hardware is not required, firmware development can start well in advance immediately after the device interface and memory maps are finalized. This saves development time.
- *Simulate I/O Peripherals:* Simulator provides the option to simulate various I/O peripherals. Using simulator's I/O support you can edit the values for I/O registers and can be used as the input/ output value in the firmware execution. Hence it eliminates the need for connecting I/O devices for debugging the firmware.
- *Simulates Abnormal Conditions:* With simulator's simulation support you can input any desired value for any parameter during debugging the firmware and can observe the control flow of firmware. It really helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behavior of the firmware under abnormal input conditions.

Limitations of Simulator Based Debugging: Though simulation based firmware debugging technique is very helpful in embedded applications, they possess certain limitations and we cannot fully rely on the simulator-based firmware debugging. Some of the limitations of simulator-based debugging are explained below:

- *Deviation from Real Behavior:* Simulation-based firmware debugging is always carried out in a development environment where the developer may not be able to debug the firmware under all possible combinations of input. Under certain operating conditions, we may get some particular result and it need not be the same when the firmware runs in a production environment.
- *Lack of Real Timeliness:* The major limitation of simulator based debugging is that it is not real-time in behavior. The debugging is developer driven and it is no way capable of creating a real time behavior. Moreover in a real application the I/O condition may be varying or unpredictable. Simulation goes for simulating those conditions for known values.

Emulators and Debuggers:

Debugging 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 examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design.

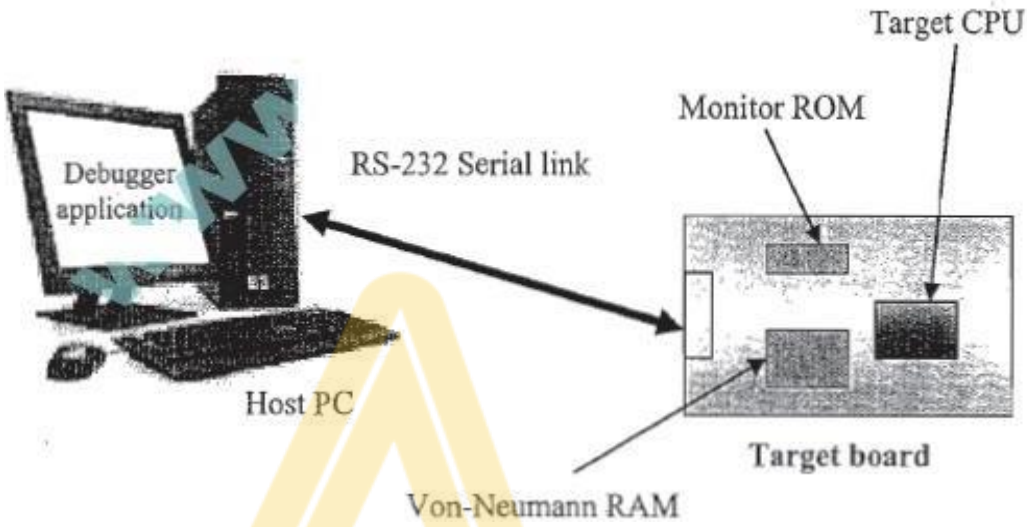
Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behavior. The following section describes the improvements over firmware debugging starting from the most primitive type of debugging to the most sophisticated On Chip Debugging (OCD):

- ***Incremental EEPROM Burning Technique:*** This is the most primitive type of firmware debugging technique where the code is separated into different functional code units. Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremental order, where the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.

- ***Inline Breakpoint Based Firmware Debugging:*** Inline breakpoint based debugging is another primitive method of firmware debugging. Within the firmware where you want to ensure that firmware execution is reaching up to a specified point, insert an inline debug code immediately after the point. The debug code is a *printf()* function which prints a string given as per the firmware. You can insert debug codes (*printf()*) commands at each point where you want to ensure the firmware execution is covering that point. Cross-compile the source code with the debug codes embedded within it. Burn the corresponding hex file into the EEPROM.

- ***Monitor Program Based Firmware Debugging:*** Monitor program based firmware debugging is the first adopted invasive method for firmware debugging (see the following Figure). In this approach a monitor program which acts as a supervisor is developed. The monitor program controls the downloading of user code into the code memory, inspects and modifies register/ memory locations; allows single stepping of source code, etc. The monitor program implements the debug functions as per a pre-defined command set from the debug application interface. The

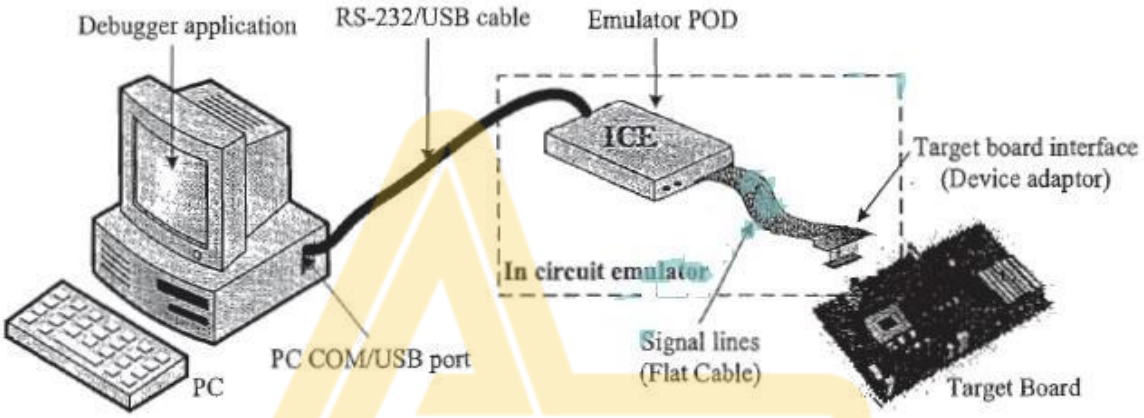
monitor program always listens to the serial port of the target device and according to the command received from the serial interface it performs command specific actions like firmware downloading, memory inspection/ modification, firmware single stepping and sends the debug information (various register and memory contents) back to the main debug program running on the development PC, etc.



- The first step in any monitor program development is determining a set of commands for performing various operations like firmware downloading, memory/ register inspection/ modification, single stepping, etc. The entire code stuff handling the command reception and corresponding action implementation is known as the "*monitor program*". The most common type of interface used between target board and debug application is RS-232C Serial interface.
- The monitor program contains the following set of minimal features:
    1. Command set interface to establish communication with the debugging application
    2. Firmware download option to code memory
    3. Examine and modify processor registers and working memory (RAM)
    4. Single step program execution
    5. Set breakpoints in firmware execution
    6. Send debug information to debug application running on host machine.

- ***In Circuit Emulator (ICE) Based Firmware Debugging:*** The terms 'Simulator' and 'Emulator' are little bit confusing and sounds similar. Though their basic functionality is the same-"Debug the target firmware", the way in which they achieve this functionality is totally different. The simulator 'simulates' the target board CPU and the emulator 'emulates' the target board CPU.

- '*Simulator*' is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU.
- '*Emulator*' is a self-contained hardware device which emulates the target CPU. The emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end.
- The Emulator POD (see the following Figure) forms the heart of any emulator system and it contains the following functional units.



- *Emulation Device:* is a replica of the target CPU which receives various signals from the target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug commands from the debug application.
- *Emulation Memory:* is the Random Access Memory (RAM) incorporated in the Emulator device. It acts as a replacement to the target board's EEPROM where the code is supposed to be downloaded after each firmware modification. Hence the original EEPROM memory is emulated by the RAM of emulator. This is known as 'ROM Emulation'. ROM emulation eliminates the hassles of ROM burning and it offers the benefit of infinite number of reprogramming.
- *Emulator Control Logic:* is the logic circuits used for implementing complex hardware breakpoints, trace buffer trigger detection, trace buffer control, etc. Emulator control logic circuits are also used for implementing logic analyzer functions in advanced emulator devices. The 'Emulator POD' is connected to the target board through a 'Device adaptor' and signal cable.
- *Device Adaptors:* act as an interface between the target board and emulator POD. Device adaptors are normally pin-to-pin compatible sockets which can be inserted/ plugged into the target board for routing the various signals from pins assigned for the target

processor. The device adaptor is usually connected to the emulator POD using ribbon cables.

- ***On Chip Firmware Debugging (OCD):*** Advances in semiconductor technology has brought out new dimensions to target firmware debugging. Today almost all processors/controllers in• corporate built in debug modules called On Chip Debug (OCD) support. Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging. The On Chip Debug facilities integrated to the processor/ controller are chip vendor dependent and most of them are proprietary technologies like Background Debug Mode (BDM), OnCE, etc.

TARGET HARDWARE DEBUGGING:

Even though the firmware is bug free and everything is intact in the board, your embedded product need not function as per the expected behavior in the first attempt for various hardware related reasons like dry soldering of components, missing connections in the PCB due to any un-noticed errors in the PCB layout design, misplaced components, signal corruption due to noise, etc. The only way to sort out these issues and figure out the real problem creator is debugging the target board.

Hardware debugging is not similar to firmware debugging. Hardware debugging involves the monitoring of various signals of the target board (address/ data lines, port pins, etc.), checking the inter connection among various components, circuit continuity checking, etc.

The various hardware debugging tools used in Embedded Product Development are explained below.

Magnifying Glass (Lens):

You might have noticed watch repairer wearing a small magnifying glass while engaged -in repairing a watch. They use the magnifying glass to view the minute components inside the watch in an enlarged manner so that they can easily work with them.

Similar to a watch repairer, *magnifying glass* is the primary hardware debugging tool for an embedded hardware debugging professional.

A *magnifying glass* is a powerful visual inspection tool. With a magnifying glass (lens), the surface of the target board can be examined thoroughly for dry soldering of components, missing components, improper placement of components, improper soldering, track (PCB connection) damage, short of tracks, etc. Nowadays high quality magnifying stations are available for visual inspection.

Multimeter:

A *multimeter* is used for measuring various electrical quantities like voltage (Both AC and DC), current (DC as well as AC), resistance, capacitance, continuity checking, transistor checking, cathode and anode identification of diode, etc.

Any multimeter will work over a specific range for each measurement. A multimeter is the most valuable tool in the tool kit of an embedded hardware developer. It is the primary debugging tool for physical contact based hardware debugging and almost all developers start debugging the hardware with it.

Digital CRO:

Cathode Ray Oscilloscope (CRO) is a little more sophisticated tool compared to a multimeter. CRO is used for waveform capturing and analysis, measurement of signal strength, etc. By connecting the point under observation on the target board to the Channels of the Oscilloscope, the waveforms can be captured and analyzed for expected behavior.

CRO is a very good tool in analyzing interference noise in the power supply line and other signal lines. Monitoring the crystal oscillator signal from the target board is a typical example of the usage of CRO for waveform capturing and analysis in target board debugging.

CROs are available in both analog and digital versions. Though Digital CROs are costly, feature-wise they are best suited for target board debugging applications. Digital CROs are available for high frequency support and they also incorporate modem techniques for recording waveform over a period of time, capturing waves on the basis of a configurable event (trigger) from the target board.

Various measurements like phase, amplitude, etc. are also possible with CROs. Tektronix, Agilent, Philips, etc. are the manufacturers of high precision good quality digital CROs.

Logic Analyzer:

A logic analyzer is the big brother of digital CRO. *Logic analyzer* is used for capturing digital data (logic 1 and 0) from a digital circuitry whereas CRO is employed in capturing all kinds of waves including logic signals. Another major limitation of CRO is that the total number of logic signals/ waveforms that can be captured with a CRO is limited to the number of channels.

A logic analyzer contains special connectors and clips which can be attached to the target board for capturing digital data. In target board debugging applications, a logic analyzer captures the states of various port pins, address bus and data bus of the target processor/ controller, etc.

Logic analyzers give an exact reflect on of what happens when a particular line of firmware is running. This is achieved by capturing the address line logic and data line logic of target hardware. Most modem logic analyzers contain provisions for storing captured data, selecting a desired region of the captured waveform, zooming selected region of the captured waveform, etc. Tektronix, Agilent, etc. are the giants in the logic analyzer market.

Function Generator:

Function generator is not a debugging tool. It is a input signal simulator tool. A *function generator* is capable of producing various periodic waveforms like sine wave, square wave, saw-tooth wave, etc. with different frequencies and amplitude.

Sometimes the target board may require some kind of periodic waveform with a particular frequency as input to some part of the board. Thus, in a debugging environment, the function generator serves the purpose of generating and supplying required signals.

BOUNDARY SCAN:

As the complexity of the hardware increase, the number of chips present in the board and the interconnection among them may also increase. The device packages used in the PCB become miniature to reduce the total board space occupied by them and multiple layers may be required to route the interconnections among the chips. With miniature device packages and multiple layers for the PCB it will be very difficult to debug the hardware using magnifying glass, multimeter, etc. to check the interconnection among the various chips.

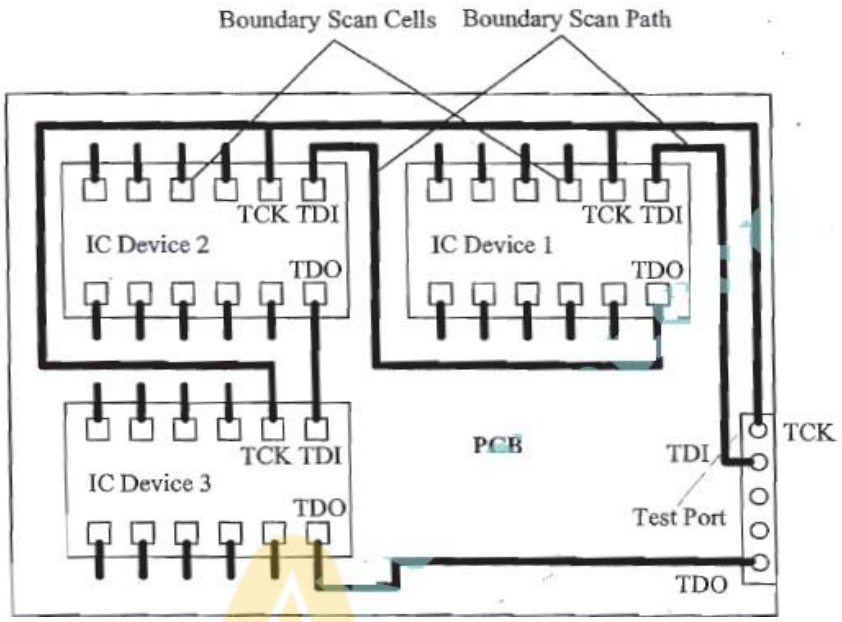
Boundary scan is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board. Chips which support boundary scan associate a boundary scan cell with each pin of the device.

A JTAG port contains the five signal lines, namely, TDI, TDO, TCK, TRST and TMS form the Test Access Port (TAP) for a JTAG supported chip. Each device will have its own TAP. The PCB also contains a TAP for connecting the JTAG signal lines to the external world.

A boundary scan path is formed inside the board by interconnecting the devices through JTAG signal lines. The TDI pin of the TAP of the PCB is connected to the TDI pin of the first device.

The TDO pin of the first device is connected to the TDI pin of the second device. In this way all devices are interconnected and the TDO pin of the last JTAG device is connected to the TDO pin of the TAP of the PCB. The clock line TCK and the Test Mode Select (TMS) line of the devices are connected to the clock line and Test mode select line of the Test Access Port of the PCB respectively. This forms a boundary scan path.

The following Figure illustrates the same.



By: DR. MAHESH PRASANNA K.

MR. SANDESHA KARANTH P. K.

DEPT. OF CSE, VCET.

****

****