MODULE – 1

ARM - 32 bit Microcontroller

What is ARM?

- ARM is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) developed by ARM Holdings
- It was named the Advanced RISC Machine, and before that, the Acorn RISC Machine

What is the ARM Cortex-M3 Processor?

- The ARM Cortex-M3 processor, the first of the Cortex generation of processors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market.
- The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously available only in high-end processors.

Background of ARM

- ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology.
- In 1991, ARM introduced the ARM6 processor family.
  - VLSI became the initial licensee.
  - Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products.

Background of ARM (continued)

- ARM does not manufacture processors or sell the chips directly.
- Instead, ARM licenses the processor designs to business partners, including a majority of the world's leading semiconductor companies.
- Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, microcontrollers, and system-on-chip solutions.
- This business model is commonly called intellectual property (IP) licensing.
- In addition to processor designs, ARM also licenses systems-level IP and various software IPs.
  - To support these products, ARM has developed a strong base of development tools, hardware, and software products to enable partners to develop their own products.

THE CORTEX-M3 PROCESSOR VERSUS CORTEX-M3-BASED MCUs

The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based microcontroller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features. This book focuses on the architecture of the processor core. For details about the rest of the chip, readers are advised to check the particular chip manufacturer's documentation.

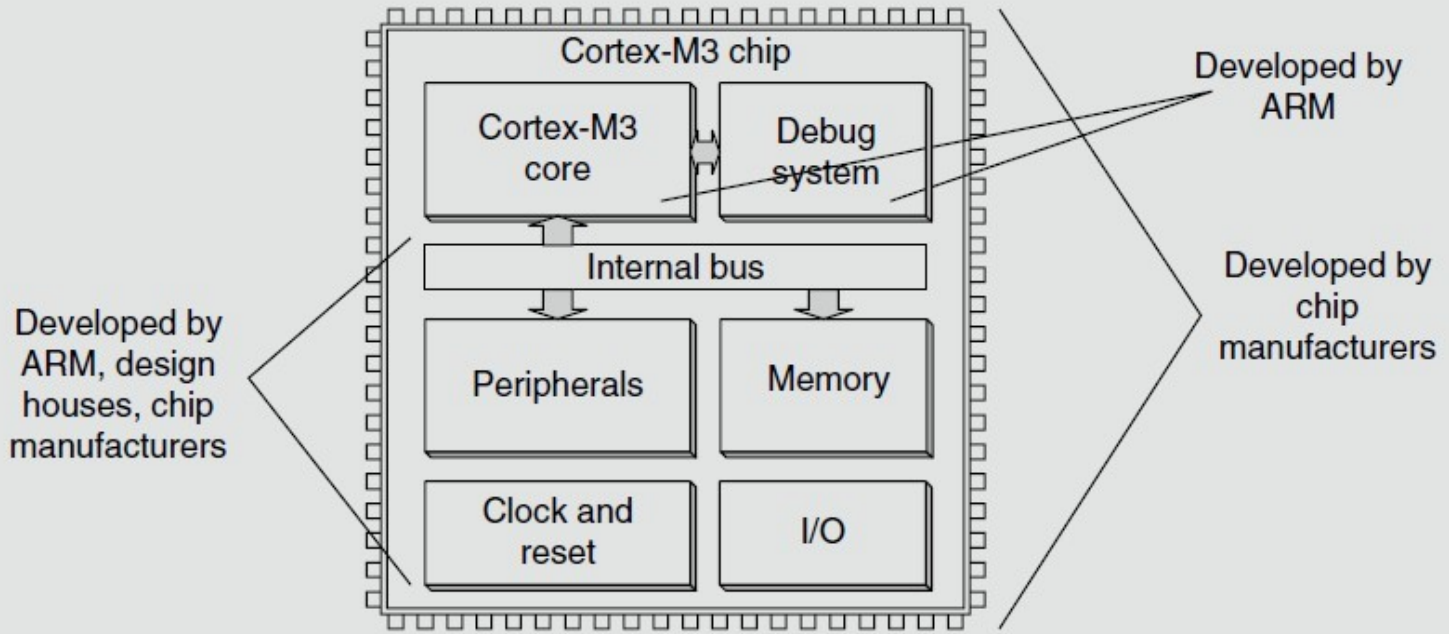


FIGURE 1.1

The Cortex-M3 Processor versus the Cortex-M3-Based MCU.

Architecture Versions

- Classic Processors
- ARM Cortex-A Processors
- ARM Cortex-R Processors
- ARM Cortex-M Processors

ARM7

- The Arm7TDMI-S is an excellent workhorse processor capable of a wide array of applications. Traditionally used in mobile handsets, the processor is now broadly in many non-mobile applications.

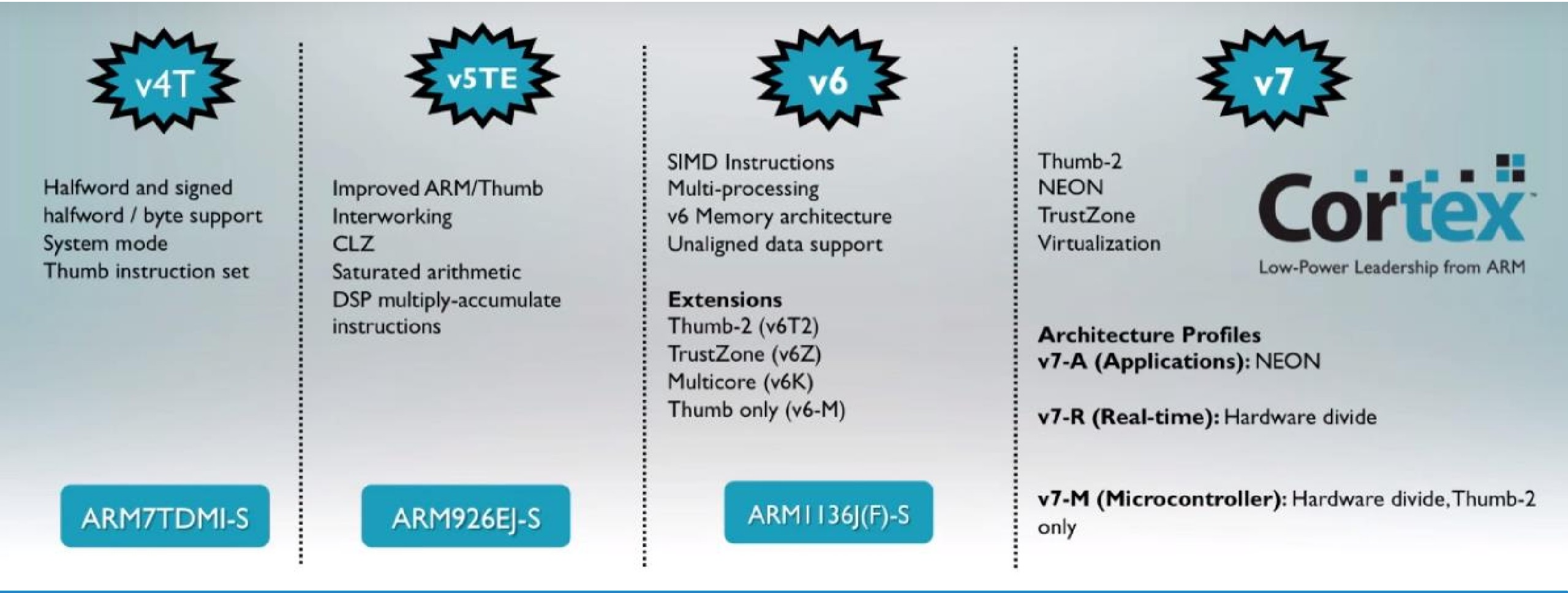
ARM9

- The Arm9 family includes three processors:
- Arm968E-S is the smallest and lowest-power Arm9 processor, built with interfaces for Tightly Coupled Memory and aimed at real-time applications.
- Arm946E-S is a real-time orientated processor with optional cache interfaces, a full Memory Protection Unit, and Tightly Coupled Memory.
- Arm926EJ-S is the entry point processor capable of supporting full Operating Systems including Linux, WindowsCE, and Symbian.

ARM11

- The Arm11 family includes four processors:
- Arm11MPCore introduced multicore technology and is still used in a wide range of applications.
- Arm1176JZ(F)-S is the highest-performance single-core processor in the Classic Arm family. It also introduced TrustZone technology to enable secure execution outside of the reach of malicious code.
- Arm1156T2(F)-S is the highest-performance processor in the real-time Classic Arm family.
- Arm1136J(F)-S is very similar to Arm926EJ-S, but includes an extended pipeline, basic SIMD (Single Instruction Multiple Data) instructions, and improved frequency and performance.

Development of the ARM Architecture



ARM Cortex Family

ARM Cortex Processors

- Cortex-A Series
  - Designed for high-performance open application platforms
- Cortex-R Series
  - Designed for high-end embedded systems in which real-time performance is needed
- Cortex-M Series
  - Designed for deeply embedded microcontroller-type systems

A Profile (ARMv7-A)

- Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) (e.g., Symbian, Linux, and Windows Embedded).
- These processors requiring the highest processing power, virtual memory system support with memory management units (MMUs), and, optionally, enhanced Java support and a secure program execution environment.
- Example products include high-end mobile phones and electronic wallets for financial transactions.

R Profile (ARMv7-R)

- Real-time, high-performance processors targeted primarily at the higher end of the real-time market
- Suitable for those applications, such as high-end breaking systems and hard drive controllers, in which high processing power and high reliability are essential and for which low latency is important.

M Profile (ARMv7-M)

- Processors targeting low-cost applications in which processing efficiency is important and cost, power consumption, low interrupt latency, and ease of use are critical, as well as industrial control applications, including real-time control systems.
- The Cortex processor families are the first products developed on architecture v7, and the Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

Evolution of ARM Processor Architecture

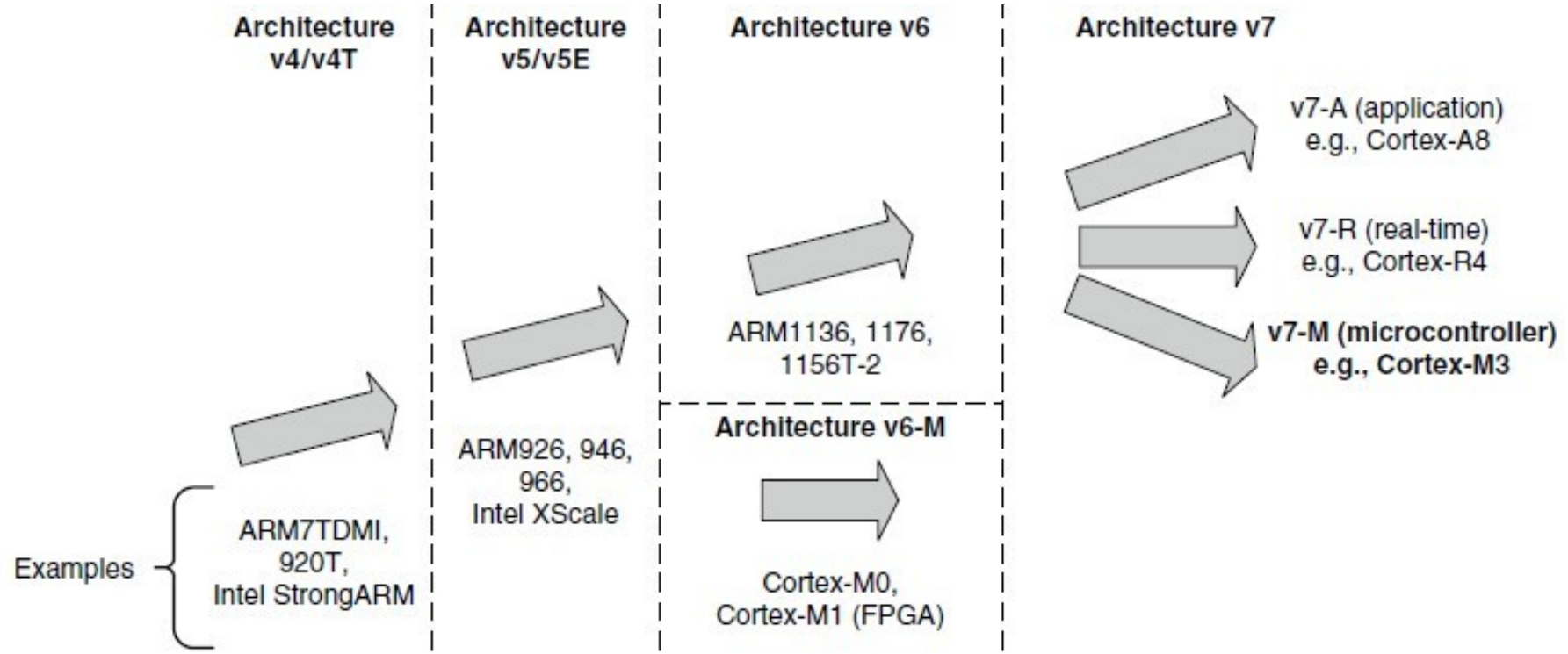
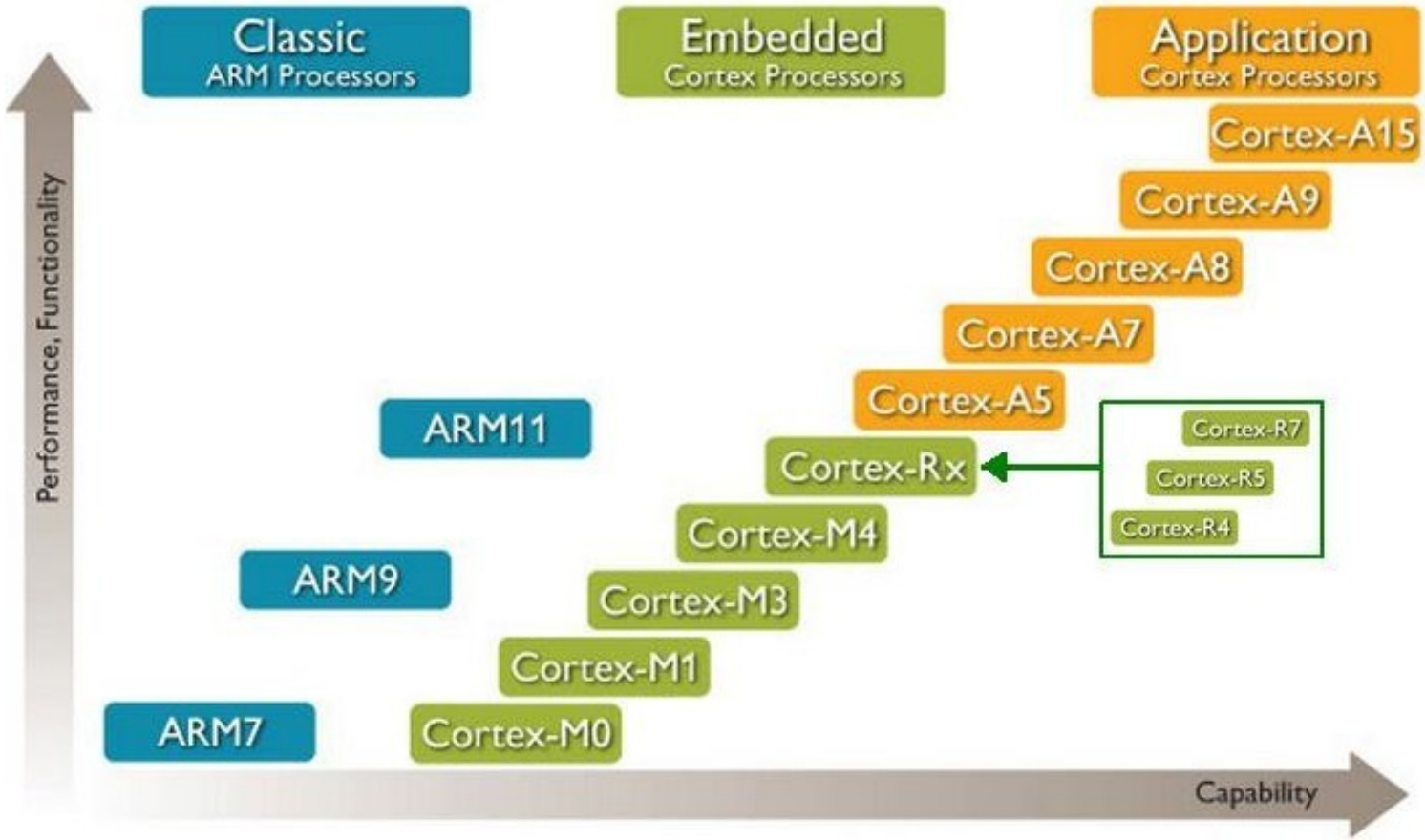


FIGURE 1.2

The Evolution of ARM Processor Architecture.

Evolution of ARM Processor Architecture (continued)



Instruction Set Development

- Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor:
- the ARM instructions that are 32 bits and Thumb instructions that are 16 bits.
- During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets.
- The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density.
  - It is useful for products with tight memory requirements.

Instruction Set Development (continued)

- As the architecture version has been updated, extra instructions have been added to both ARM instructions and Thumb instructions.
- In 2003, ARM announced the Thumb-2 instruction set, which is a new superset of Thumb instructions that contains both 16-bit and 32-bit instructions.

The Thumb-2 Technology and Instruction Set Architecture

- The Thumb-2 technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance.
- The extended instruction set in Thumb-2 is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions.
- It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state.

The Thumb-2 Technology and Instruction Set Architecture (continued)

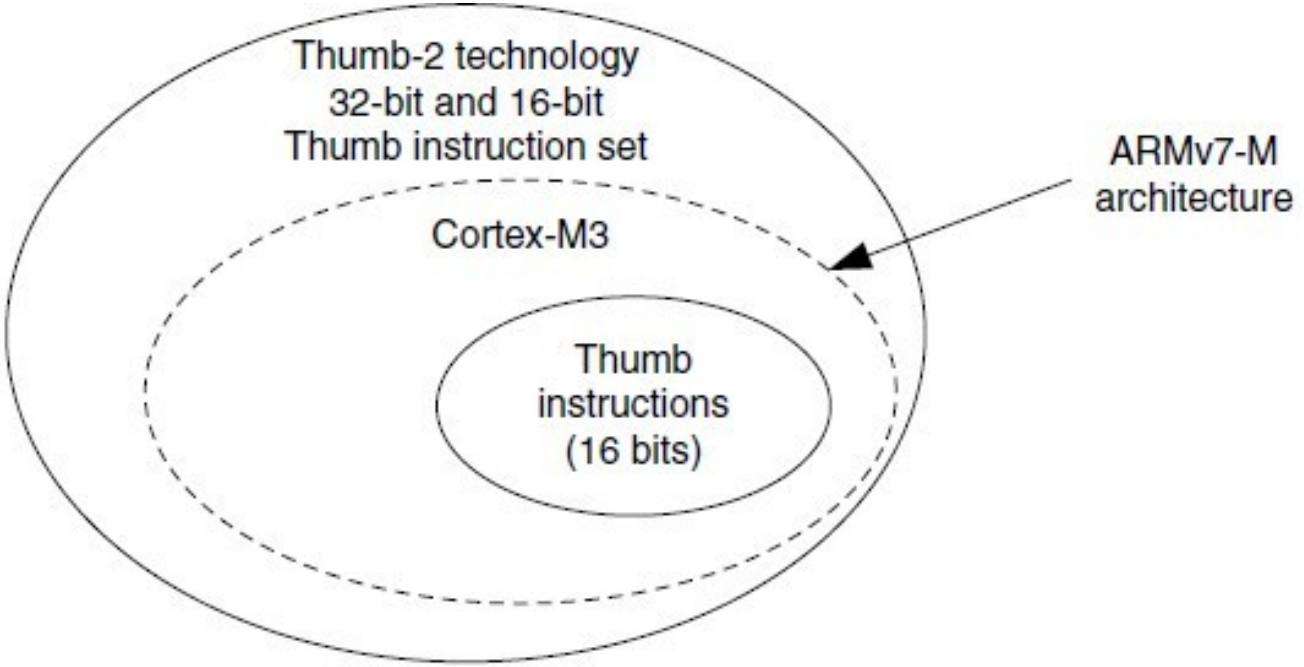


FIGURE 1.4

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

The Thumb-2 Technology and Instruction Set Architecture (continued)

- The Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set.
- Instead of using ARM instructions for some operations, as in traditional ARM processors, it uses the Thumb-2 instruction set for all operations.
- As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors.

The Thumb-2 Technology and Instruction Set Architecture (continued)

- With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions).
- In the Cortex-M3 processor, 32-bit instructions can be mixed with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity.
- The Thumb-2 instruction set is a very important feature of the ARMv7 architecture.

Cortex-M3 Processor Applications

- Low-cost microcontrollers:
  - The Cortex-M3 processor is ideally suited for low-cost microcontrollers, which are commonly used in consumer products, from toys to electrical appliances.
  - It is a highly competitive market due to the many well-known 8-bit and 16-bit microcontroller products on the market.
  - Its lower power, high performance, and ease-of-use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture.
- Automotive:
  - The Cortex-M3 processor has very high-performance efficiency and low interrupt latency, allowing it to be used in real-time systems.
  - The Cortex-M3 processor supports up to 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional MPU, making it ideal for highly integrated and cost-sensitive automotive applications.

Cortex-M3 Processor Applications (continued)

- Data communications:
  - The processor's low power and high efficiency, coupled with instructions in Thumb-2 for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.
- Industrial control:
  - In industrial control applications, simplicity, fast response, and reliability are key factors.
  - Again, the Cortex-M3 processor's interrupt feature, low interrupt latency, and enhanced fault-handling features make it a strong candidate in this area.

Cortex-M3 Processor Applications (continued)

- Consumer products:
  - In many consumer products, a high-performance microprocessor (or several of them) is used.
  - The Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection

Advantages of Cortex-M3 Processor

- Greater performance efficiency: Allows more work to be done without increasing the frequency or power requirements
- Low power consumption: Enables longer battery life, especially critical in portable products
- Enhanced determinism: Guarantees that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles
- Improved code density: Ensures that code fits in even the smallest memory footprints
- Ease of use: Provides easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32 bits
- Lower cost solutions: Reduces 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit microcontrollers to be priced at less than US$1 for the first time
- Wide choice of development tools: From low-cost or free compilers to full-featured development suites from many development tool vendors

Architecture of ARM Cortex-M3

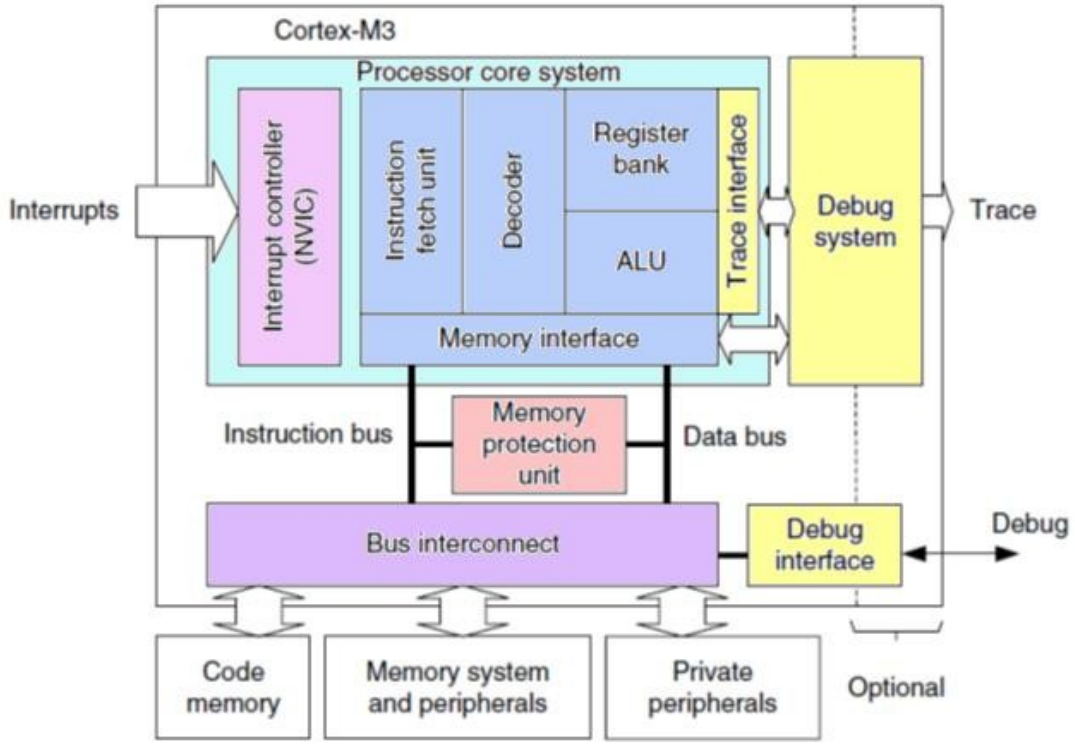


FIGURE 2.1

A Simplified View of the Cortex-M3.

Architecture of ARM Cortex-M3 (continued)

- The Cortex-M3 is a 32-bit microprocessor.
  - It has a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces.
- The processor has a Harvard architecture, which means that it has a separate instruction bus and data bus.
  - This allows instructions and data accesses to take place at the same time.
  - The performance of the processor increases because data accesses do not affect the instruction pipeline.
  - This feature results in multiple bus interfaces on Cortex-M3, each with optimized usage and the ability to be used simultaneously.
- However, the instruction and data buses share the same memory space (a unified memory system).
  - In other words, you cannot get 8 GB of memory space just because you have separate bus interfaces.

Architecture of ARM Cortex-M3 (continued)

- For complex applications that require more memory system features, the Cortex-M3 processor has an optional Memory Protection Unit (MPU), and it is possible to use an external cache if it’s required.
- Both little endian and big endian memory systems are supported.
- The Cortex-M3 processor includes a number of fixed internal debugging components.
  - These components provide debugging operation supports and features, such as breakpoints and watchpoints.
- In addition, optional components provide debugging features, such as instruction trace, and various types of debugging interfaces.

Features of ARM Cortex-M3

- Three-stage pipeline design
- Harvard bus architecture with unified memory space: instructions and data use the same address space
- 32-bit addressing, supporting 4GB of memory space
- On-chip bus interfaces based on ARM AMBA (Advanced Microcontroller Bus Architecture) Technology, which allow pipelined bus operations for higher throughput
- An interrupt controller called NVIC (Nested Vectored Interrupt Controller) supporting up to 240 interrupt requests and from 8 to 256 interrupt priority levels (dependent on the actual device implementation)

Features of ARM Cortex-M3 (continued)

- Support for various features for OS (Operating System) implementation such as a system tick timer, shadowed stack pointer
- Sleep mode support and various low power features
- Support for an optional MPU (Memory Protection Unit) to provide memory protection features like programmable memory, or access permission control
- Support for bit-data accesses in two specific memory regions using a feature called Bit Band
- The option of being used in single processor or multi-processor designs

Registers

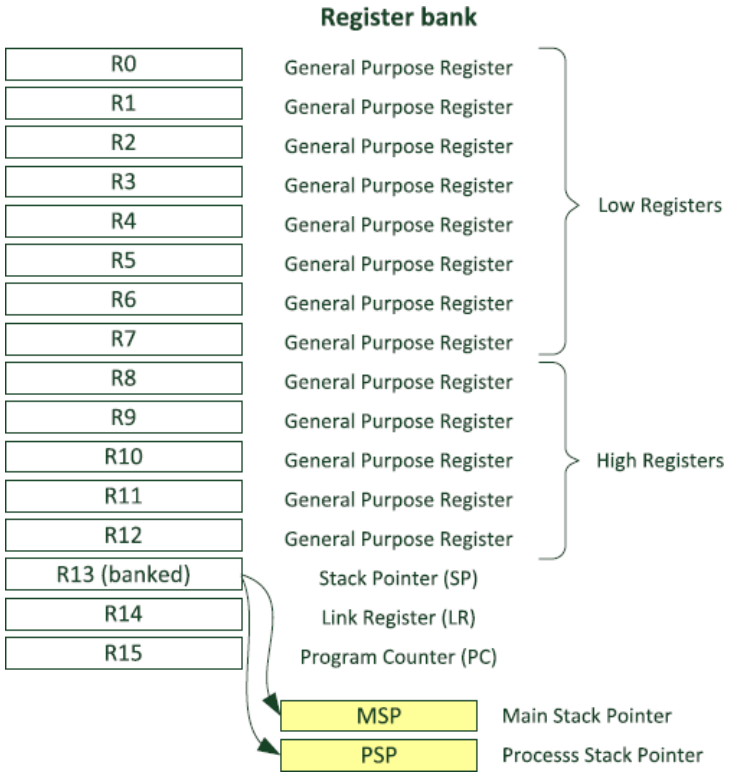


FIGURE 2.2

Registers in the Cortex-M3.

Registers (continued)

- The Cortex-M3 processor has registers R0 through R15.
- R0–R12: General-Purpose Registers
  - R0–R12 are 32-bit general-purpose registers for data operations.
  - Some 16-bit Thumb instructions can only access a subset of these registers (low registers, R0–R7).
- R13: Stack Pointers
  - The Cortex-M3 contains two stack pointers (R13).
  - They are banked so that only one is visible at a time.
  - The two stack pointers are as follows:
    - Main Stack Pointer (MSP): The default stack pointer, used by the operating system (OS) kernel and exception handlers
    - Process Stack Pointer (PSP): Used by user application code
  - The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

Registers (continued)

- R14: Link Register
  - When a subroutine is called, the return address is stored in the link register.
- R15: Program Counter
  - The program counter is the current program address.
  - This register can be written to control the program flow.

Registers (continued)

- Special Registers
  - The Cortex-M3 processor also has a number of special registers.
  - They are as follows:
    - Program Status registers (PSRs)
    - Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
    - Control register (CONTROL)
  - These registers have special functions and can be accessed only by special instructions.
  - They cannot be used for normal data processing.

Registers (continued)

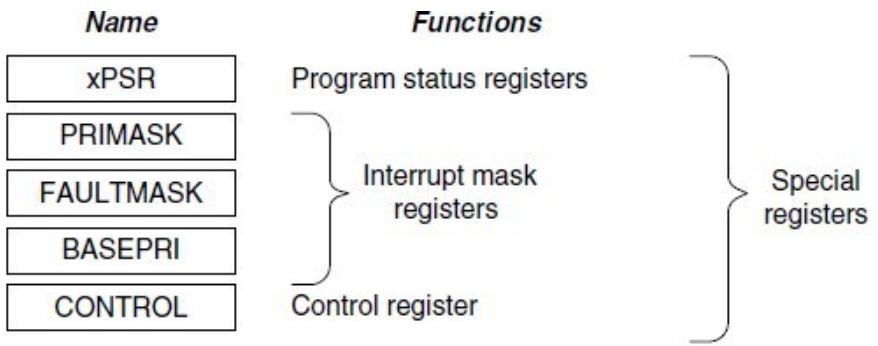


FIGURE 2.3

Special Registers in the Cortex-M3.

Table 2.1 Special Registers and Their Functions

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

Registers

- The Cortex-M3 processor has registers R0 through R15 and a number of special registers.
- General Purpose Registers R0 through R7
  - The R0 through R7 general purpose registers are also called low registers.
  - They can be accessed by all 16-bit Thumb instructions and all 32-bit Thumb-2 instructions.
  - These registers are all 32 bits.
  - The reset value is unpredictable
- General Purpose Registers R8 through R12
  - The R8 through R12 registers are also called high registers.
  - They are accessible by all Thumb-2 instructions but not by all 16-bit Thumb instructions.
  - These registers are all 32 bits
  - The reset value is unpredictable.

Registers (continued)

- Stack Pointer R13
  - R13 is the stack pointer (SP).
  - In the Cortex-M3 processor, there are two SPs.
  - This duality allows two separate stack memories to be set up.
  - When using the register name R13, we can only access the current SP; the other one is inaccessible unless we use special instructions MSR and MRS.
  - The two SPs are as follows:
    - Main Stack Pointer (MSP) or *SP_main*:
      - This is the default SP
      - It is used by the operating system (OS) kernel, exception handlers, and all application codes that require privileged access.
    - Process Stack Pointer (PSP) or *SP_process*:
      - This is used by the base-level application code (when not running an exception handler).

STACK PUSH AND POP

Stack is a memory usage model. It is simply part of the system memory, and a pointer register (inside the processor) is used to make it work as a first-in/last-out buffer. The common use of a stack is to save register contents before some data processing and then restore those contents from the stack after the processing task is done.

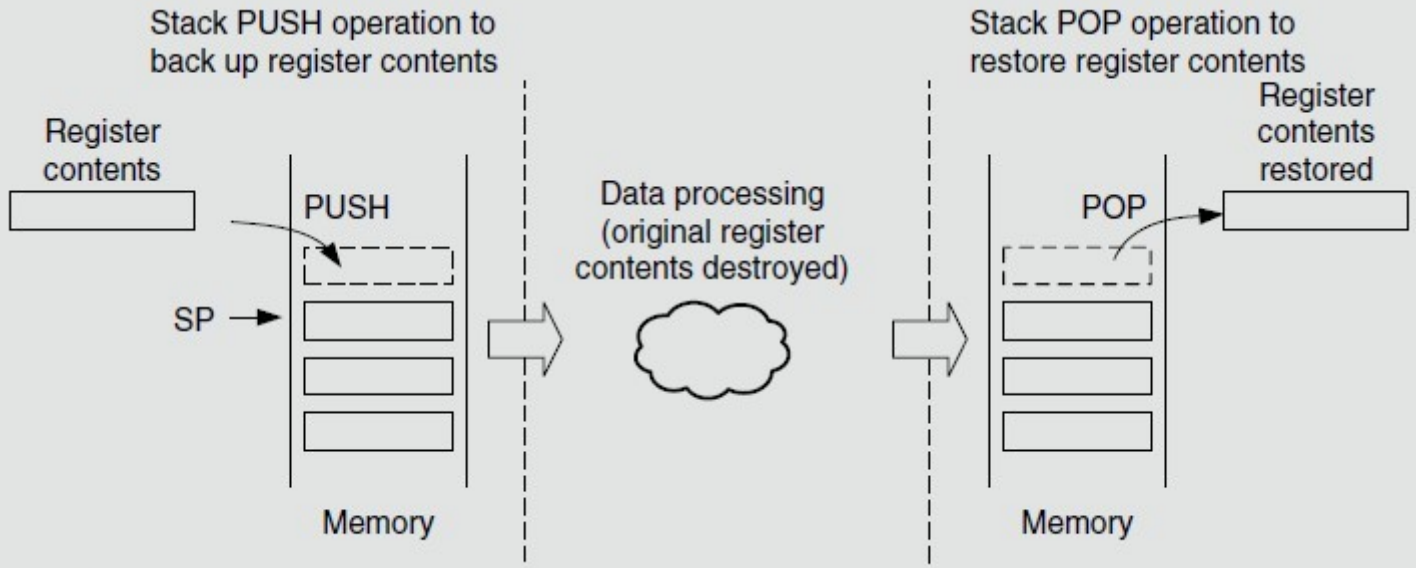


FIGURE 3.2

Basic Concept of Stack Memory.

When doing PUSH and POP operations, the pointer register, commonly called stack pointer, is adjusted automatically to prevent next stack operations from corrupting previous stacked data. More details on stack operations are provided on later part of this chapter.

Registers (continued)

- Stack Pointer R13 (continued)
  - It is not necessary to use both SPs.
  - Simple applications can rely purely on the MSP.
  - In the Cortex-M3, the instructions for accessing stack memory are PUSH and POP.
  - The assembly language syntax is as follows

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

  - PUSH and POP are usually used to save register contents to stack memory at the start of a subroutine and then restore the registers from stack at the end of the subroutine.

Registers (continued)

- Stack Pointer R13 (continued)
  - We can PUSH or POP multiple registers in one instruction:

```
subroutine_1
     PUSH   {R0-R7, R12, R14} ; Save registers
     ...                      ; Do your processing
     POP    {R0-R7, R12, R14} ; Restore registers
     BX     R14               ; Return to calling function
```

  - Instead of using R13, you can use SP in program codes.
  - Because register PUSH and POP operations are always word aligned (their addresses must be 0x0, 0x4, 0x8, ...), the SP/R13 bit 0 and bit 1 are hardwired to 0 and always read as zero (RAZ).

Registers (continued)

- Link Register R14
  - R14 is the link register (LR).
  - Inside an assembly program, we can write it as either R14 or LR.
  - LR is used to store the return program counter (PC) when a subroutine or function is called.
  - E.g.: when we're using the branch and link (BL) instruction:

```
main ; Main program
     ...
     BL function1 ; Call function1 using Branch with Link instruction.
                  ; PC = function1 and
                  ; LR = the next instruction in main
     ...
function1
     ...          ; Program code for function 1
     BX LR        ; Return
```

Registers (continued)

- Link Register R14 (continued)
  - Despite the fact that bit 0 of the PC is always 0 (because instructions are word aligned or half word aligned), the LR bit 0 is readable and writable.
  - This is because in the Thumb instruction set, bit 0 is often used to indicate ARM/Thumb states.
  - To allow the Thumb-2 program for the Cortex-M3 to work with other ARM processors that support the Thumb-2 technology, this least significant bit (LSB) is writable and readable.

Registers (continued)

- Program Counter R15
  - R15 is the Program Counter.
  - It can be accessed in assembler code by either R15 or PC.
  - Because of the pipelined nature of the Cortex-M3 processor, when you read this register, you will find that the value is different than the location of the executing instruction, normally by 4.
    - For example:

```
0x1000 : MOV R0, PC ; R0 = 0x1004
```

  - Because an instruction address must be half word aligned, the LSB (bit 0) of the PC read value is always 0.
  - However, in branching, either by writing to PC or using branch instructions, the LSB of the target address should be set to 1 because it is used to indicate the Thumb state operations.
  - If it is 0, it can imply trying to switch to the ARM state and will result in a fault exception in the Cortex-M3.

Special Registers

- The special registers in the Cortex-M3 processor include the following:
  - Program Status registers (PSRs)
  - Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
  - Control register (CONTROL)
- Special registers can only be accessed via MSR and MRS instructions; they do not have memory addresses.

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

Program Status Registers

- The PSRs are subdivided into three status registers:
  - Application Program Status register (APSR)
  - Interrupt Program Status register (IPSR)
  - Execution Program Status register (EPSR)
- The three PSRs can be accessed together or separately using the special register access instructions MSR and MRS.
- When they are accessed as a collective item, the name *xPSR* is used.

Program Status Registers (continued)

- You can read the PSRs using the MRS instruction.
- You can also change the APSR using the MSR instruction, but EPSR and IPSR are read-only.
- For example:

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

Program Status Registers (continued)

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

FIGURE 3.3

Program Status Registers (PSRs) in the Cortex-M3.

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

FIGURE 3.4

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

Program Status Registers (continued)

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

```
MRS     r0, PSR     ; Read the combined program status word
MSR     PSR, r0     ; Write combined program state word
```

- The descriptions for the bit fields in PSR are shown in Table 3.1.

Table 3.1 Bit Fields in Cortex-M3 Program Status Registers

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

Program Status Registers (continued)

- If you compare this with the Current Program Status register (CPSR) in ARM7, you might find that some bit fields that were used in ARM7 are gone.
  - The Mode (M) bit field is gone because the Cortex-M3 does not have the operation mode as defined in ARM7.
  - Thumb-bit (T) is moved to bit 24.
  - Interrupt status (I and F) bits are replaced by the new interrupt mask registers (PRIMASKs), which are separated from PSR.
- For comparison, the CPSR in traditional ARM processors is shown in Figure 3.5.

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
ARM (general)	N	Z	C	V	Q	IT	J	Reserved	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7 TDMI	N	Z	C	V	Reserved								I	F	T	M[4:0]

FIGURE 3.5

Current Program Status Registers in Traditional ARM Processors.

Interrupt Mask Registers

- The PRIMASK, FAULTMASK, and BASEPRI registers are used to disable exceptions.
- The PRIMASK and BASEPRI registers are useful for temporarily disabling interrupts in timing-critical tasks.
- An OS could use FAULTMASK to temporarily disable fault handling when a task has crashed.
  - In this scenario, a number of different faults might be taking place when a task crashes.
  - Once the core starts cleaning up, it might not want to be interrupted by other faults caused by the crashed process.
  - Therefore, the FAULTMASK gives the OS kernel time to deal with fault conditions.

Interrupt Mask Registers (continued)

Table 3.2 Cortex-M3 Interrupt Mask Registers

Register Name	Description
PRIMASK	A 1-bit register, when this is set, it allows nonmaskable interrupt (NMI) and the hard fault exception; all other interrupts and exceptions are masked. The default value is 0, which means that no masking is set.
FAULTMASK	A 1-bit register, when this is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled. The default value is 0, which means that no masking is set.
BASEPRI	A register of up to 8 bits (depending on the bit width implemented for priority level). It defines the masking priority level. When this is set, it disables all interrupts of the same or lower level (larger priority value). Higher priority interrupts can still be allowed. If this is set to 0, the masking function is disabled (this is the default).

Interrupt Mask Registers (continued)

- To access the PRIMASK, FAULTMASK, and BASEPRI registers, a number of functions are available in the device driver libraries provided by the microcontroller vendors.
- For example, the following:

```
x = __get_BASEPRI(); // Read BASEPRI register
x = __get_PRIMARK(); // Read PRIMASK register
x = __get_FAULTMASK(); // Read FAULTMASK register
__set_BASEPRI(x); // Set new value for BASEPRI
__set_PRIMASK(x); // Set new value for PRIMASK
__set_FAULTMASK(x); // Set new value for FAULTMASK
__disable_irq(); // Clear PRIMASK, enable IRQ
__enable_irq(); // Set PRIMASK, disable IRQ
```

Interrupt Mask Registers (continued)

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

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

- The PRIMASK, FAULTMASK, and BASEPRI registers cannot be set in the user access level.

The Control Register

- The control register is used to define the privilege level and the SP selection.
- This register has 2 bits, as shown in Table 3.3.

Table 3.3 Cortex-M3 Control Register

Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be 0 when the processor is in handler mode.
CONTROL[0]	0 = Privileged in thread mode 1 = User state in thread mode If in handler mode (not thread mode), the processor operates in privileged mode.

The Control Register (continued)

- CONTROL[1]
  - In the Cortex-M3, the CONTROL[1] bit is always 0 in handler mode.
  - However, in the thread or base level, it can be either 0 or 1.
  - This bit is writable only when the core is in thread mode and privileged.
  - In the user state or handler mode, writing to this bit is not allowed.
  - Aside from writing to this register, another way to change this bit is to change bit 2 of the LR when in exception return.

The Control Register (continued)

- CONTROL[0]
  - The CONTROL[0] bit is writable only in a privileged state.
  - Once it enters the user state, the only way to switch back to privileged is to trigger an interrupt and change this in the exception handler.
  - To access the control register in C, the following Cortex Microcontroller Software Interface Standard (CMSIS) functions are available in CMSIS compliant device driver libraries:

```
x = __get_CONTROL(); // Read the current value of CONTROL
__set_CONTROL(x); // Set the CONTROL value to x
```

  - To access the control register in assembly, the MRS and MSR instructions are used:

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

Operation Modes

- The Cortex-M3 processor has two operation modes and two privilege levels.
- The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler.
- The privilege levels (privileged level and user level) provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.

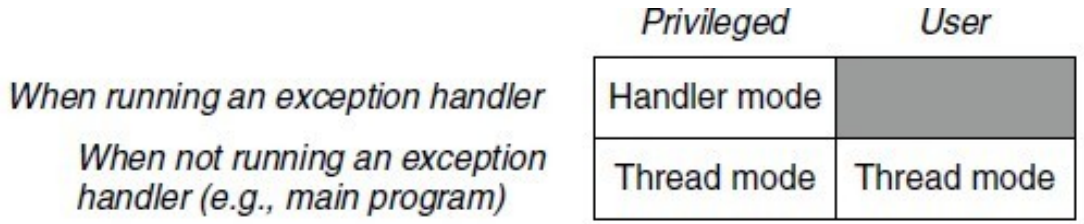


FIGURE 2.4

Operation Modes and Privilege Levels in Cortex-M3.

Operation Modes (continued)

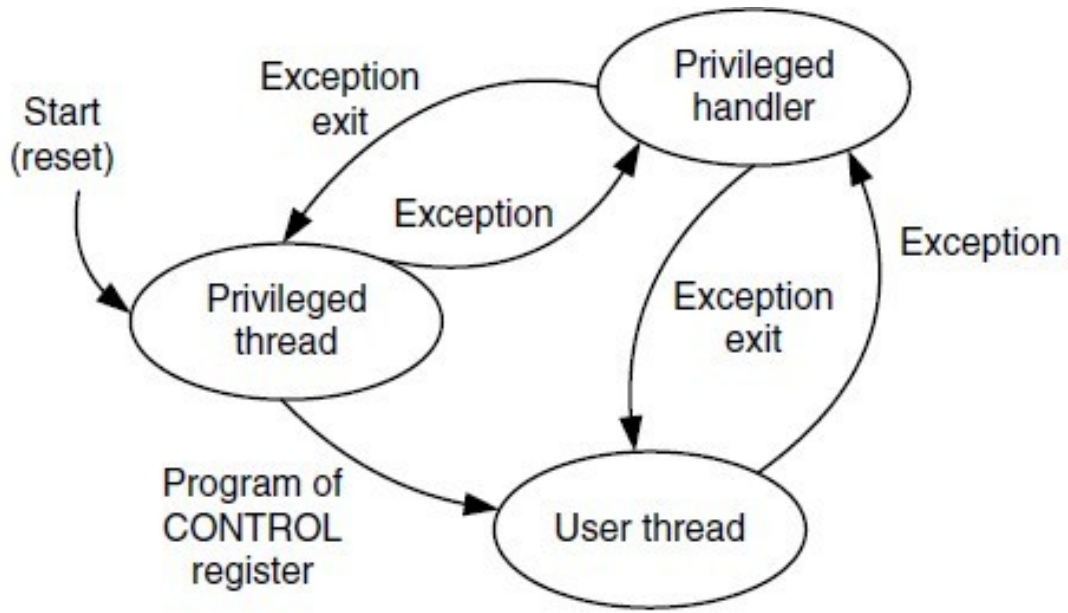


FIGURE 2.5

Allowed Operation Mode Transitions.

Operation Modes (continued)

- When the processor is running a main program (thread mode), it can be either in a privileged state or a user state, but exception handlers can only be in a privileged state.
- When the processor exits reset, it is in thread mode, with privileged access rights.
- In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions.

Operation Modes (continued)

- Software in the privileged access level can switch the program into the user access level using the control register.
- When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler.
- A user program cannot change back to the privileged state by writing to the control register.
- It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode.

Operation Modes (continued)

- The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs.
- If an MPU is available, it can be used in conjunction with privilege levels to protect critical memory locations, such as programs and data for OSs.
  - For example, with privileged accesses, usually used by the OS kernel, all memory locations can be accessed (unless prohibited by MPU setup).
  - When the OS launches a user application, it is likely to be executed in the user access level to protect the system from failing due to a crash of untrusted user programs.

Operation Modes

- The Cortex-M3 processor supports two modes (thread mode and handler mode) and two privilege levels (privileged level and user level).
- When the processor is running in thread mode, it can be in either the privileged or user level, but handlers can only be in the privileged level.

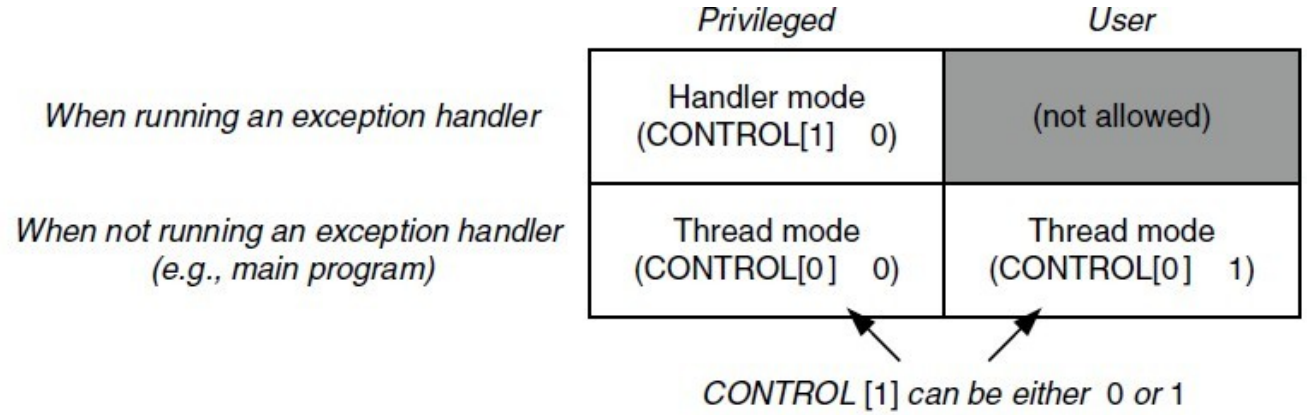


FIGURE 3.6

Operation Modes and Privilege Levels in Cortex-M3.

Operation Modes (continued)

- In the user access level (thread mode), access to the system control space (SCS)—a part of the memory region for configuration registers and debugging components—is blocked.
- Furthermore, instructions that access special registers (such as MSR, except when accessing APSR) cannot be used.
- If a program running at the user access level tries to access SCS or special registers, a fault exception will occur.

Operation Modes (continued)

- Software in a privileged access level can switch the program into the user access level using the control register.
- When an exception takes place, the processor will always switch to a privileged state and return to the previous state when exiting the exception handler.
- A user program cannot change back to the privileged state directly by writing to the control register.
  - It has to go through an exception handler that programs the control register to switch the processor back into privileged access level when returning to thread mode.

Operation Modes (continued)

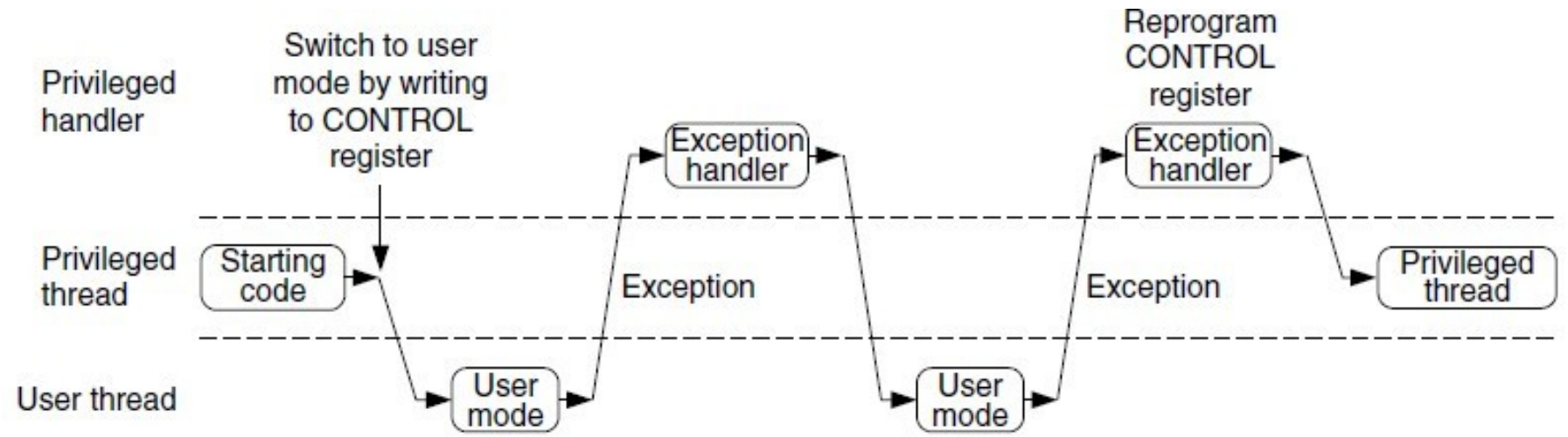


FIGURE 3.7

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

Operation Modes (continued)

- In simple applications, there is no need to separate the privileged and user access levels.
  - In these cases, there is no need to use user access level and no need to program the control register.

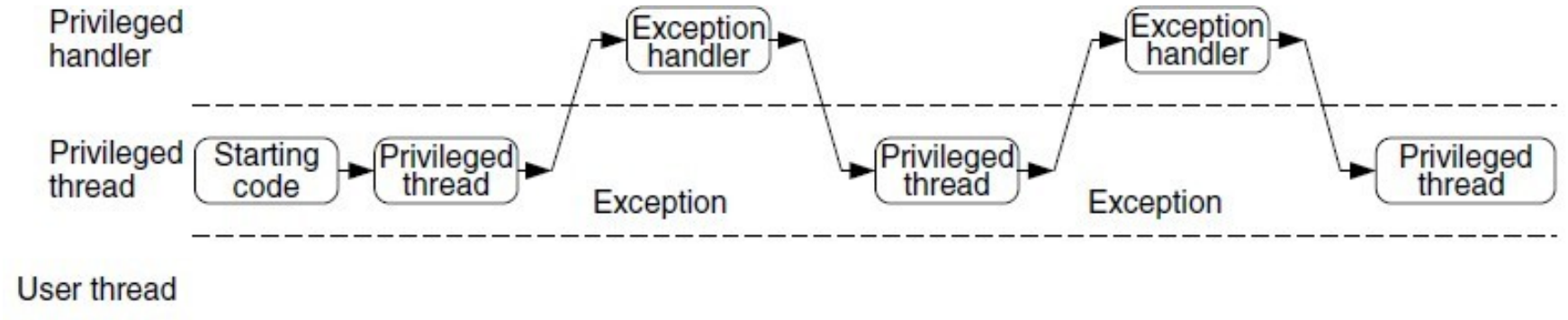


FIGURE 3.8

Simple Applications Do Not Require User Access Level in Thread Mode.

Operation Modes (continued)

- The mode and access level of the processor are defined by the control register.
- When the control register bit 0 is 0, the processor mode changes when an exception takes place.
- When control register bit 0 is 1 (thread running user application), both processor mode and access level change when an exception takes place.
- Control register bit 0 is programmable only in the privileged level.
  - For a user-level program to switch to privileged state, it has to raise an interrupt (for example, supervisor call [SVC]) and write to CONTROL[0] within the handler.

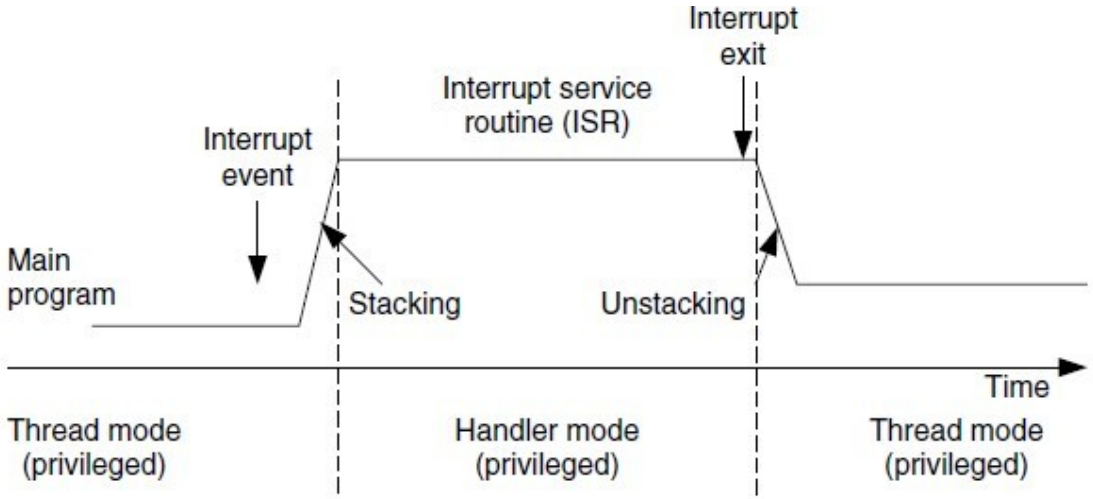


FIGURE 3.9

Switching Processor Mode at Interrupt.

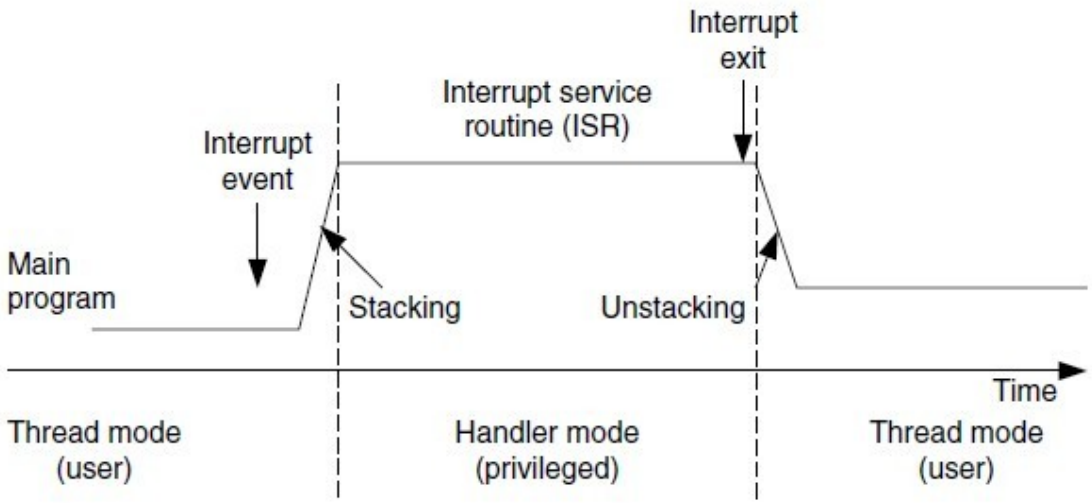


FIGURE 3.10

Switching Processor Mode and Privilege Level at Interrupt.

Nested Vectored Interrupt Controller (NVIC)

- The Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC).
- It is closely coupled to the processor core and provides a number of features as follows:
  - Nested interrupt support
  - Vectored interrupt support
  - Dynamic priority changes support
  - Reduction of interrupt latency
  - Interrupt masking

Nested Vectored Interrupt Controller (NVIC) (continued)

- Nested Interrupt Support
  - All the external interrupts and most of the system exceptions can be programmed to different priority levels.
  - When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level.
  - If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.
- Vectored Interrupt Support
  - When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory.
  - There is no need to use software to determine and branch to the starting address of the ISR.
  - Thus, it takes less time to process the interrupt request.

Nested Vectored Interrupt Controller (NVIC) (continued)

- Dynamic Priority Changes Support
  - Priority levels of interrupts can be changed by software during run time.
  - Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re-entry.
- Reduction of Interrupt Latency
  - The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency.
  - These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling of late arrival interrupts.

Nested Vectored Interrupt Controller (NVIC) (continued)

- Interrupt Masking
  - Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK.
  - They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

The Memory Map

- The Cortex-M3 has a predefined memory map.
- This allows the built-in peripherals, such as the interrupt controller and the debug components, to be accessed by simple memory access instructions.
- Thus, most system features are accessible in C program code.
- The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.
- Overall, the 4 GB memory space can be divided into ranges as shown in Figure 2.6.

The Memory Map (continued)

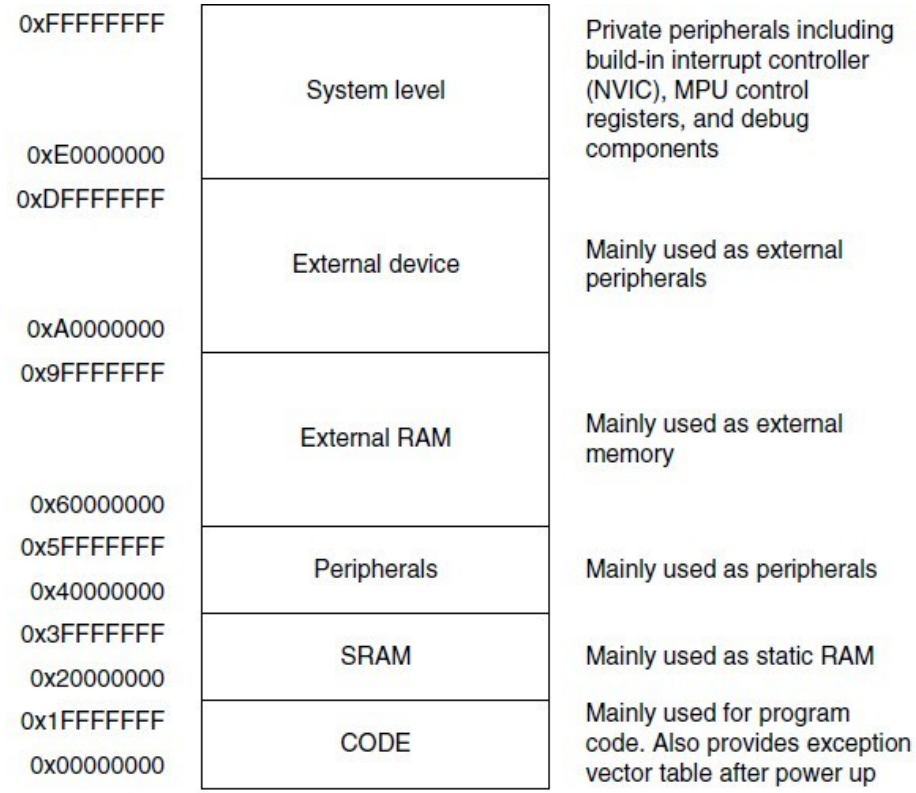


FIGURE 2.6

The Cortex-M3 Memory Map.

The Memory Map (continued)

- The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage.
- In addition, the design allows these regions to be used differently.
- For example, data memory can still be put into the CODE region, and program code can be executed from an external Random Access Memory (RAM) region.
- The system-level memory region contains the interrupt controller and the debug components.
- By having fixed addresses for these peripherals, you can port applications between different Cortex-M3 products much more easily.

The Bus Interface

- There are several bus interfaces on the Cortex-M3 processor.
- They allow the Cortex-M3 to carry instruction fetches and data accesses at the same time.
- The main bus interfaces are as follows:
  - Code memory buses
    - The code memory region access is carried out on the code memory buses, which physically consist of two buses, one called I-Code and other called D-Code.
    - These are optimized for instruction fetches for best instruction execution speed.
  - System bus
    - The system bus is used to access memory and peripherals.
    - This provides access to the Static Random Access Memory (SRAM), peripherals, external RAM, external devices, and part of the system-level memory regions.
  - Private peripheral bus
    - The private peripheral bus provides access to a part of the system-level memory dedicated to private peripherals, such as debugging components.

The Memory Protection Unit (MPU)

- The Cortex-M3 has an optional MPU.
- This unit allows access rules to be set up for privileged access and user program access.
- When an access rule is violated, a fault exception is generated, and the fault exception handler will be able to analyse the problem and correct it, if possible.
- The MPU can be used in various ways.
- In common scenarios, the OS can set up the MPU to protect data use by the OS kernel and other privileged processes to be protected from untrusted user programs.
- The MPU can also be used to make memory regions read-only, to prevent accidental erasing of data or to isolate memory regions between different tasks in a multitasking system.
- Overall, it can help make embedded systems more robust and reliable.
- The MPU feature is optional and is determined during the implementation stage of the microcontroller or SoC design.

The Instruction Set

- The Cortex-M3 supports the Thumb-2 instruction set.
  - It allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency.
  - It is flexible and powerful yet easy to use.
- In previous ARM processors, the central processing unit (CPU) had two operation states – a 32-bit ARM state and a 16-bit Thumb state.
  - In the ARM state, the instructions are 32 bits and can execute all supported instructions with very high performance.
  - In the Thumb state, the instructions are 16 bits, so there is a much higher instruction code density
    - The Thumb state does not have all the functionality of ARM instructions and may require more instructions to complete certain types of operations.

The Instruction Set (continued)

- Many applications have mixed ARM and Thumb codes.
  - However, there is overhead (in terms of both execution time and instruction space) to switch between the states, and ARM and Thumb codes might need to be compiled separately in different files.
  - This increases the complexity of software development and reduces maximum efficiency of the CPU core.
- With the introduction of the Thumb-2 instruction set, it is now possible to handle all processing requirements in one operation state.
  - There is no need to switch between the two.
  - In fact, the Cortex-M3 does not support the ARM code.
  - Even interrupts are now handled with the Thumb state.

The Instruction Set (continued)

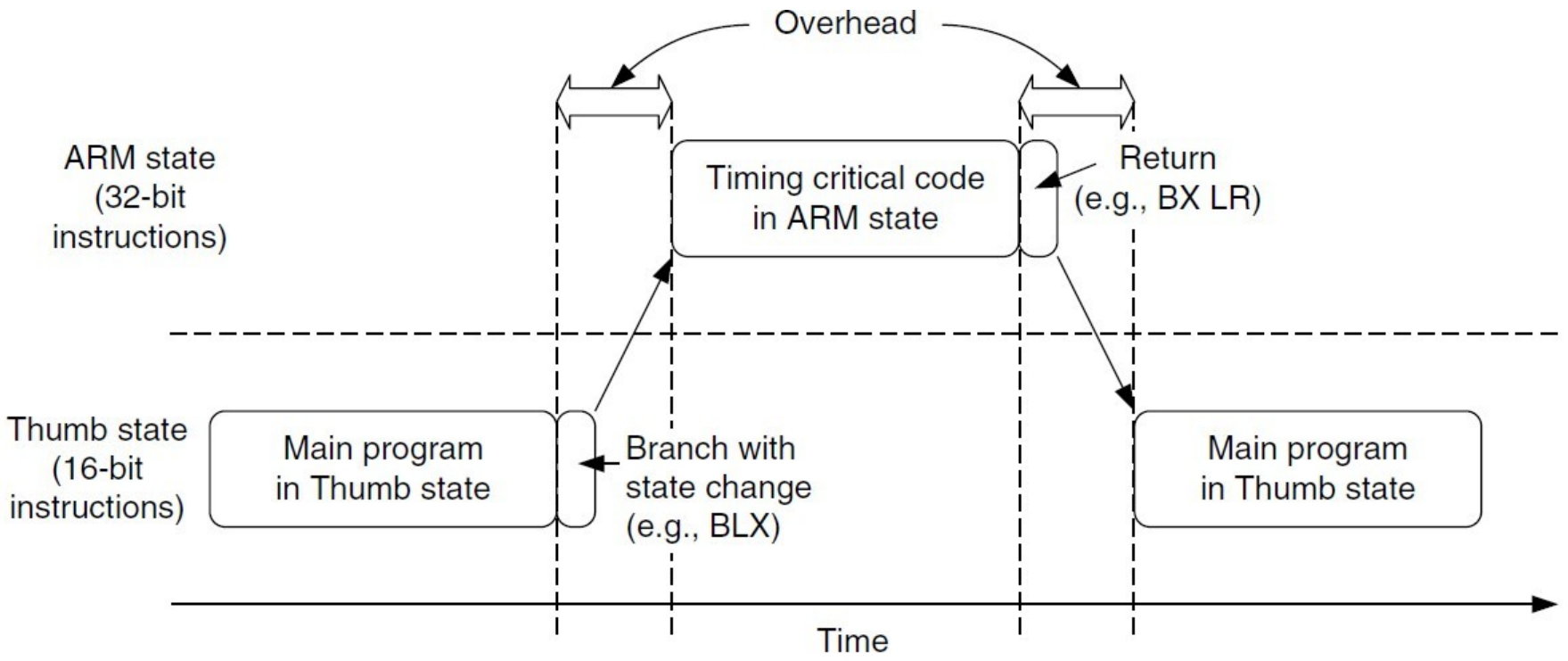


FIGURE 2.7

Switching between ARM Code and Thumb Code in Traditional ARM Processors Such as the ARM7.

The Instruction Set (continued)

- Since there is no need to switch between states, the Cortex-M3 processor has a number of advantages over traditional ARM processors, such as:
  - No state switching overhead, saving both execution time and instruction space
  - No need to separate ARM code and Thumb code source files, making software development and maintenance easier
  - It's easier to get the best efficiency and performance, in turn making it easier to write software, because there is no need to worry about switching code between ARM and Thumb to try to get the best density/performance

The Instruction Set (continued)

- The Cortex-M3 processor has a number of interesting and powerful instructions. Here are a few examples:
  - UFBX, BFI, and BFC: Bit field extract, insert, and clear instructions
  - UDIV and SDIV: Unsigned and signed divide instructions
  - WFE, WFI, and SEV: Wait-For-Event, Wait-For-Interrupts, and Send-Event; these allow the processor to enter sleep mode and to handle task synchronization on multiprocessor systems
  - MSR and MRS: Move to special register from general-purpose register and move special register to general-purpose register; for access to the special registers

Interrupts and Exceptions

- The Cortex-M3 processor implements a new exception model, introduced in the ARMv7-M architecture.
  - Enables very efficient exception handling.
- It has a number of system exceptions plus a number of external Interrupt Request (IRQs) (external interrupt inputs).
- Interrupt priority handling and nested interrupt support are now included in the interrupt architecture.
- The interrupt features in the Cortex-M3 are implemented in the NVIC.
- Aside from supporting external interrupts, the Cortex-M3 also supports a number of internal exception sources, such as system fault handling.
- As a result, the Cortex-M3 has a number of predefined exception types, as shown in Table 2.2.

Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (external NMI input)
3	Hard fault	–1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system service
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
…	…	…	…
255	IRQ #239	Programmable	External interrupt #239

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported. In addition, the Cortex-M3 also has an NMI interrupt input. When it is asserted, the NMI-ISR is executed unconditionally.

Exceptions and Interrupts

- The Cortex-M3 supports a number of exceptions, including a fixed number of system exceptions and a number of interrupts, commonly called *IRQ*.
- The number of interrupt inputs on a Cortex-M3 microcontroller depends on the individual design.
  - The typical number of interrupt inputs is 16 or 32.
- Besides the interrupt inputs, there is also a nonmaskable interrupt (NMI) input signal.
  - In most cases, the NMI could be connected to a watchdog timer or a voltage-monitoring block that warns the processor when the voltage drops below a certain level.
  - The NMI exception can be activated any time, even right after the core exits reset.

Table 3.4 Exception Types in Cortex-M3

Exception Number	Exception Type	Priority	Function
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt
3	Hard fault	–1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	Bus fault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16–255	IRQ	Settable	IRQ input #0–239

Vector Tables

- When an exception event takes place on the Cortex-M3 and is accepted by the processor core, the corresponding exception handler is executed.
- To determine the starting address of the exception handler, a vector table mechanism is used.
- The vector table is an array of word data inside the system memory, each representing the starting address of one exception type.
- The vector table is relocatable, and the relocation is controlled by a relocation register in the NVIC (see Table 3.5).
- After reset, this relocation control register is reset to 0; therefore, the vector table is located in address 0x0 after reset.

Vector Tables (continued)

Table 3.5 Vector Table Definition after Reset

Exception Type	Address Offset	Exception Vector
18–255	0x48–0x3FF	IRQ #2–239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug monitor
11	0x2C	SVC
7–10	0x1C–0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x00	Starting value of the MSP

Vector Tables (continued)

- For example, if the reset is exception type 1, the address of the reset vector is 1 times 4 (each word is 4 bytes), which equals 0x00000004, and NMI vector (type 2) is located in 2 × 4 = 0x00000008.
- The address 0x00000000 is used to store the starting value for the MSP.
- The LSB of each exception vector indicates whether the exception is to be executed in the Thumb state.
- Because the Cortex-M3 can support only Thumb instructions, the LSB of all the exception vectors should be set to 1.

Low Power and High Energy Efficiency

- The Cortex-M3 processor is designed with various features to allow designers to develop low power and high energy efficient products.
  - It has sleep mode and deep sleep mode supports, which can work with various system-design methodologies to reduce power consumption during idle period.
  - Its low gate count and design techniques reduce circuit activities in the processor to allow active power to be reduced.
  - It has high code density and hence it has lowered the program size requirement.
  - It allows processing tasks to be completed in a short time, so that the processor can return to sleep modes as soon as possible to cut down energy use.
- Starting from Cortex-M3 revision 2, a new feature called Wakeup Interrupt Controller (WIC) is available.
  - This feature allows the whole processor core to be powered down, while processor states are retained and the processor can be returned to active state almost immediately when an interrupt takes place.
  - This makes the Cortex-M3 even more suitable for many ultra-low power applications

Debugging Support

- The Cortex-M3 processor includes a number of debugging features, such as program execution controls, including halting and stepping, instruction breakpoints, data watchpoints, registers and memory accesses, profiling, and traces.
- The debugging hardware of the Cortex-M3 processor is based on the CoreSight architecture.
  - Unlike traditional ARM processors, the CPU core itself does not have a Joint Test Action Group (JTAG) interface.
  - Instead, a debug interface module is decoupled from the core, and a bus interface called the Debug Access Port (DAP) is provided at the core level.
  - Through this bus interface, external debuggers can access control registers to debug hardware as well as system memory, even when the processor is running.

Debugging Support (continued)

- The control of DAP bus interface is carried out by a Debug Port (DP) device.
- The DPs currently available are the Serial-Wire JTAG Debug Port (SWJ-DP) (supports the traditional JTAG protocol as well as the Serial-Wire protocol) or the SW-DP (supports the Serial-Wire protocol only).
- A JTAG-DP module from the ARM CoreSight product family can also be used.
- Chip manufacturers can choose to attach one of these DP modules to provide the debug interface.
- Chip manufacturers can also include an Embedded Trace Macrocell (ETM) to allow instruction trace.
  - Trace information is output via the Trace Port Interface Unit (TPIU), and the debug host (usually a Personal Computer [PC]) can then collect the executed instruction information via external trace-capturing hardware.

Debugging Support (continued)

- Within the Cortex-M3 processor, a number of events can be used to trigger debug actions.
  - Debug events can be breakpoints, watchpoints, fault conditions, or external debugging request input signals.
- When a debug event takes place, the Cortex-M3 processor can either enter halt mode or execute the debug monitor exception handler.
- The data watchpoint function is provided by a Data Watchpoint and Trace (DWT) unit in the Cortex-M3 processor.
  - This can be used to stop the processor (or trigger the debug monitor exception routine) or to generate data trace information.
  - When data trace is used, the traced data can be output via the TPIU.

Debugging Support (continued)

- In addition to these basic debugging features, the Cortex-M3 processor also provides a Flash Patch and Breakpoint (FPB) unit that can provide a simple breakpoint function or remap an instruction access from Flash to a different location in SRAM.
- An Instrumentation Trace Macrocell (ITM) provides a new way for developers to output data to a debugger.
  - By writing data to register memory in the ITM, a debugger can collect the data via a trace interface and display or process them.
  - This method is easy to use and faster than JTAG output.
- All these debugging components are controlled via the DAP interface bus on the Cortex-M3 or by a program running on the processor core, and all trace information is accessible from the TPIU.

Characteristics Summary

- High Performance
  - The Cortex-M3 processor delivers high performance in microcontroller products:
    - Many instructions are single cycle
    - Separate data and instruction buses
    - No state switching overhead
    - The Thumb-2 instruction set provides extra flexibility in programming
    - Instruction fetches are 32 bits
    - Operate at high clock frequency (over 100 MHz)

Characteristics Summary (continued)

- Advanced Interrupt-Handling Features
  - The interrupt features on the Cortex-M3 processor are easy to use, very flexible, and provide high interrupt processing throughput:
    - The built-in NVIC supports up to 240 external interrupt inputs
    - It reduces the interrupt handling latency
    - Interrupt arrangement is extremely flexible
    - A minimum of eight levels of priority are supported, and the priority can be changed dynamically.
    - Some of the multicycle operations are now interruptible
    - Immediate execution of the NMI handler is guaranteed on receipt of NMI request

Characteristics Summary (continued)

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

Characteristics Summary (continued)

- System Features
  - The Cortex-M3 processor provides various system features making it suitable for a large number of applications::
    - The system provides bit-band operation, byte-invariant big endian mode, and unaligned data access support.
    - Advanced fault-handling features include various exception types and fault status registers, making it easier to locate problems.
    - With the shadowed stack pointer, stack memory of kernel and user processes can be isolated.
    - With the optional MPU, the processor is more than sufficient to develop robust software and reliable products.

Characteristics Summary (continued)

- Debug Supports
  - The Cortex-M3 processor includes comprehensive debug features to help software developers design their products:
    - Supports JTAG or Serial-Wire debug interfaces
    - Based on the CoreSight debugging solution, processor status or memory contents can be accessed even when the core is running
    - Built-in support for six breakpoints and four watchpoints
    - Optional ETM for instruction trace and data trace using DWT
    - New debugging features, including fault status registers, new fault exceptions, and Flash Patch operations, make debugging much easier
    - ITM provides an easy-to-use method to output debug information from test code
    - PC sampler and counters inside the DWT provide code-profiling information

Stack Memory Operations

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

Basic Operations of the Stack

- In general, stack operations are memory write or read operations, with the address specified by an SP.
- Data in registers is saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation.
- The SP is adjusted automatically in PUSH and POP so that multiple data PUSH will not cause old stacked data to be erased.
- The function of the stack is to store register contents in memory so that they can be restored later, after a processing task is completed.
- For normal uses, for each store (PUSH), there must be a corresponding read (POP), and the address of the POP operation should match that of the PUSH operation.
- When PUSH/POP instructions are used, the SP is incremented/decremented automatically.

Basic Operations of the Stack (continued)

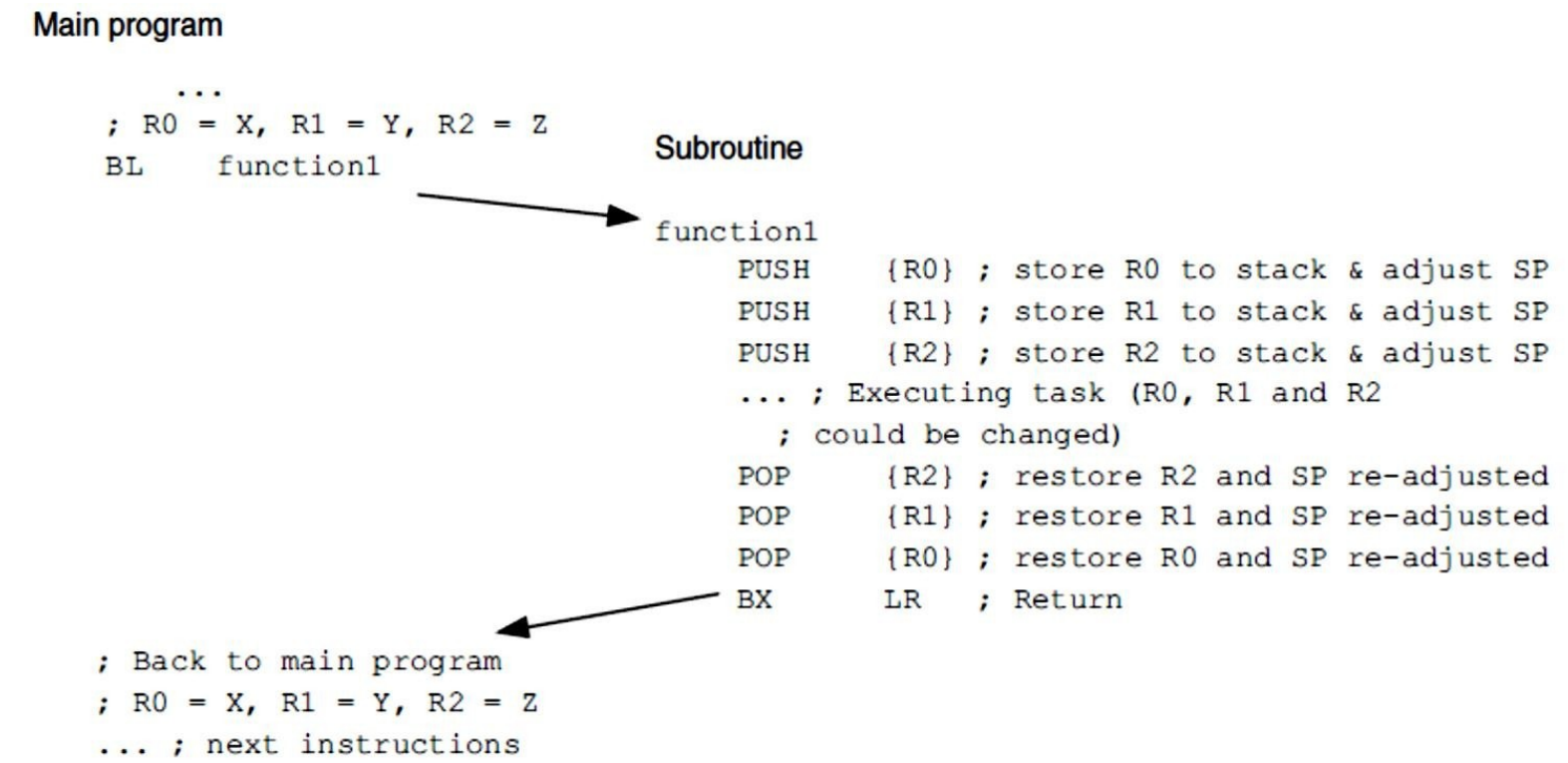


FIGURE 3.11

Stack Operation Basics: One Register in Each Stack Operation.

Basic Operations of the Stack (continued)

- When program control returns to the main program, the R0 – R2 contents are the same as before.
- Notice the order of PUSH and POP: The POP order must be the reverse of PUSH.
- These operations can be simplified, thanks to PUSH and POP instructions allowing multiple load and store.
- In this case, the ordering of a register POP is automatically reversed by the processor.

Basic Operations of the Stack (continued)

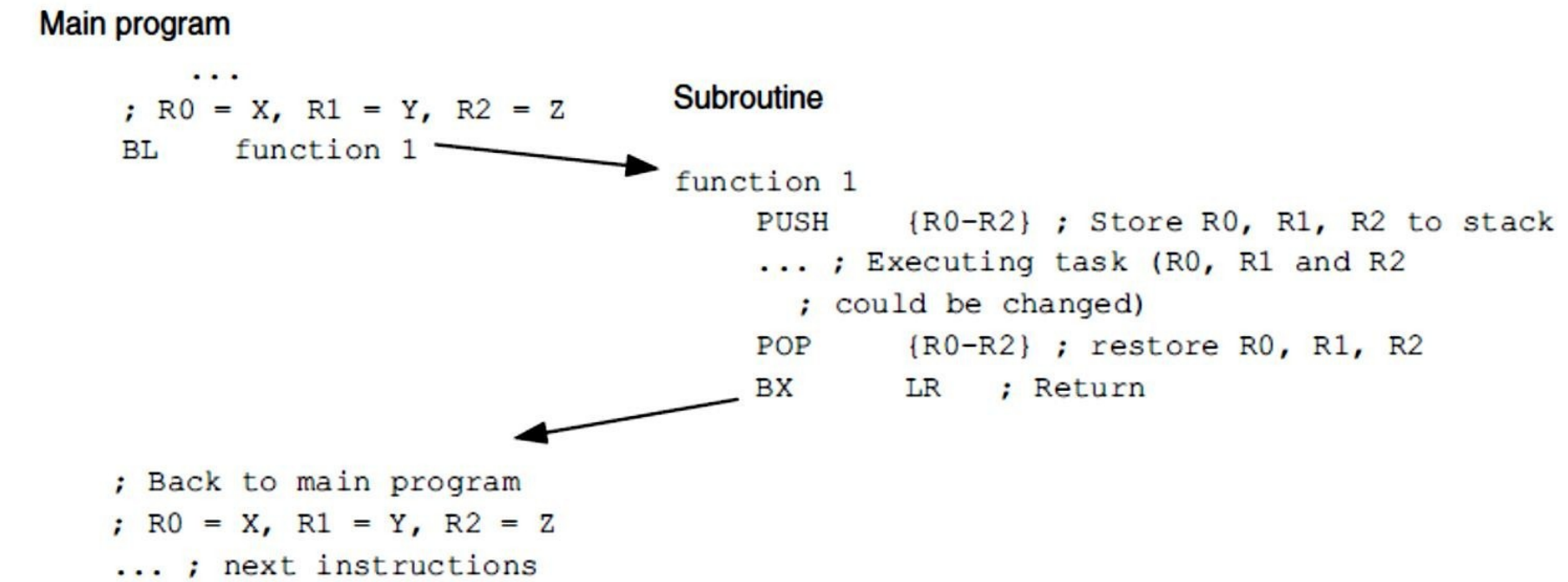


FIGURE 3.12

Stack Operation Basics: Multiple Register Stack Operation.

Basic Operations of the Stack (continued)

- We can also combine RETURN with a POP operation.
  - This is done by pushing the LR to the stack and popping it back to PC at the end of the subroutine.

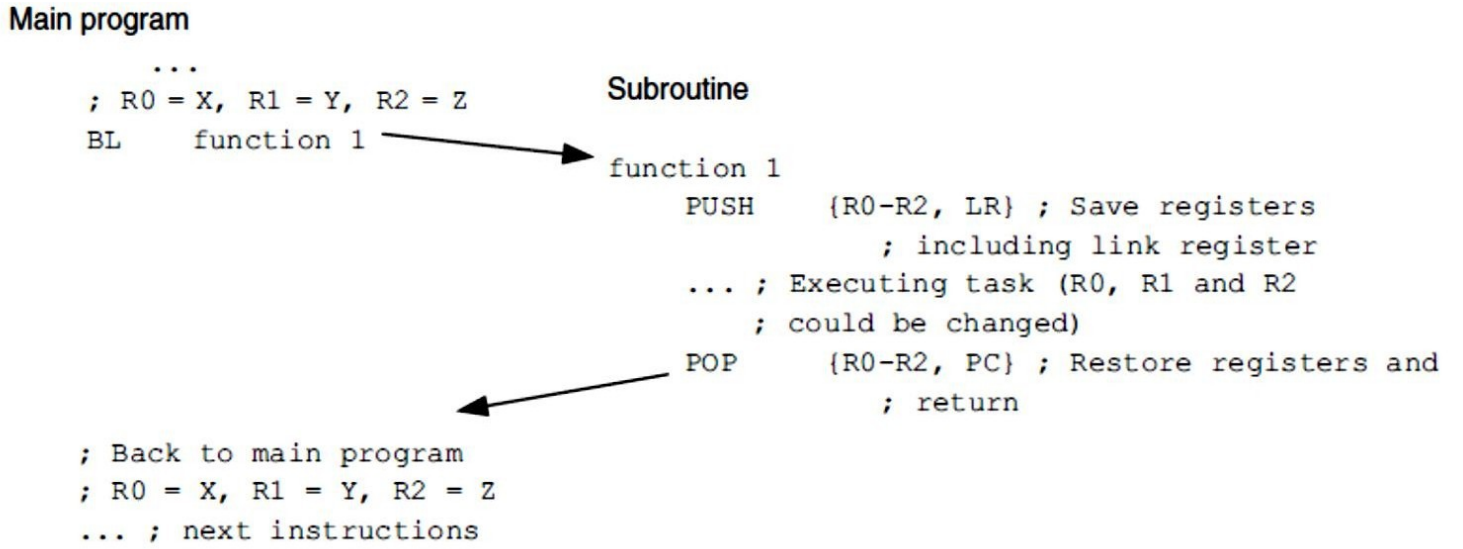


FIGURE 3.13

Stack Operation Basics: Combining Stack POP and RETURN.

Cortex-M3 Stack Implementation

- The Cortex-M3 uses a full-descending stack operation model.
- The SP points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation.
- Figure 3.14 shows execution of the instruction PUSH {R0}.

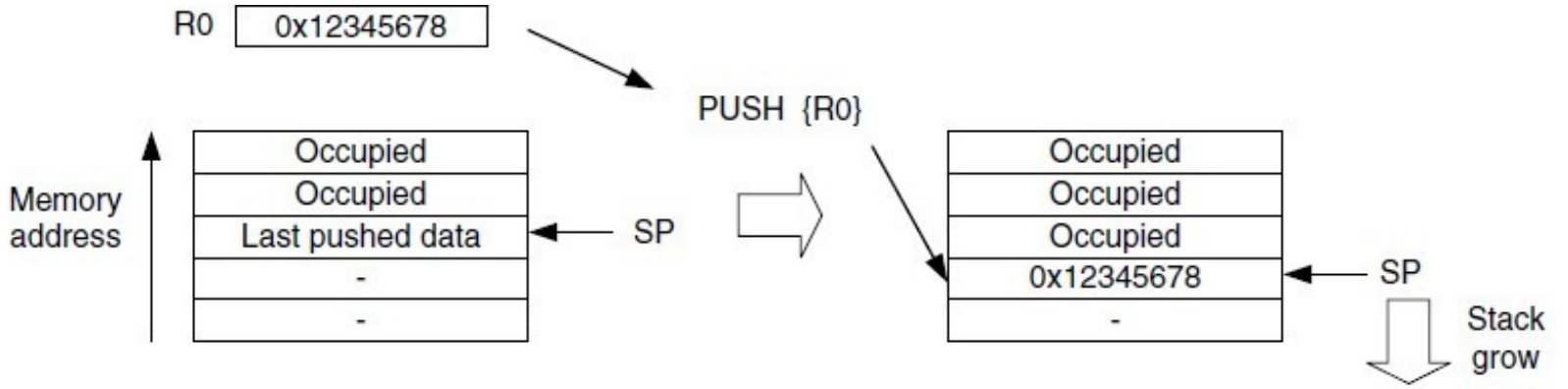


FIGURE 3.14

Cortex-M3 Stack PUSH Implementation.

Cortex-M3 Stack Implementation (continued)

- For POP operations, the data is read from the memory location pointed by SP, and then, the SP is incremented.
- The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place.

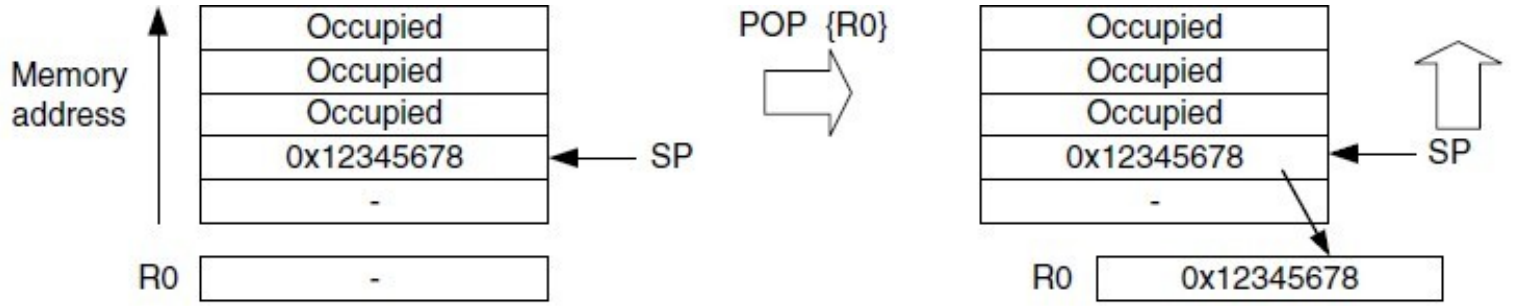


FIGURE 3.15

Cortex-M3 Stack POP Implementation.

Cortex-M3 Stack Implementation (continued)

- Because each PUSH/POP operation transfers 4 bytes of data (each register contains 1 word, or 4 bytes), the SP decrements/increments by 4 at a time or a multiple of 4 if more than 1 register is pushed or popped.
- In the Cortex-M3, R13 is defined as the SP. When an interrupt takes place, a number of registers will be pushed automatically, and R13 will be used as the SP for this stacking process.
- Similarly, the pushed registers will be restored/popped automatically when exiting an interrupt handler, and the SP will also be adjusted.

The Two-Stack Model in the Cortex-M3

- The Cortex-M3 has two SPs: the MSP and the PSP.
- The SP register to be used is controlled by the control register bit 1 (CONTROL[1]).
- When CONTROL[1] is 0, the MSP is used for both thread mode and handler mode.
  - In this arrangement, the main program and the exception handlers share the same stack memory region.
  - This is the default setting after power-up.

The Two-Stack Model in the Cortex-M3 (continued)

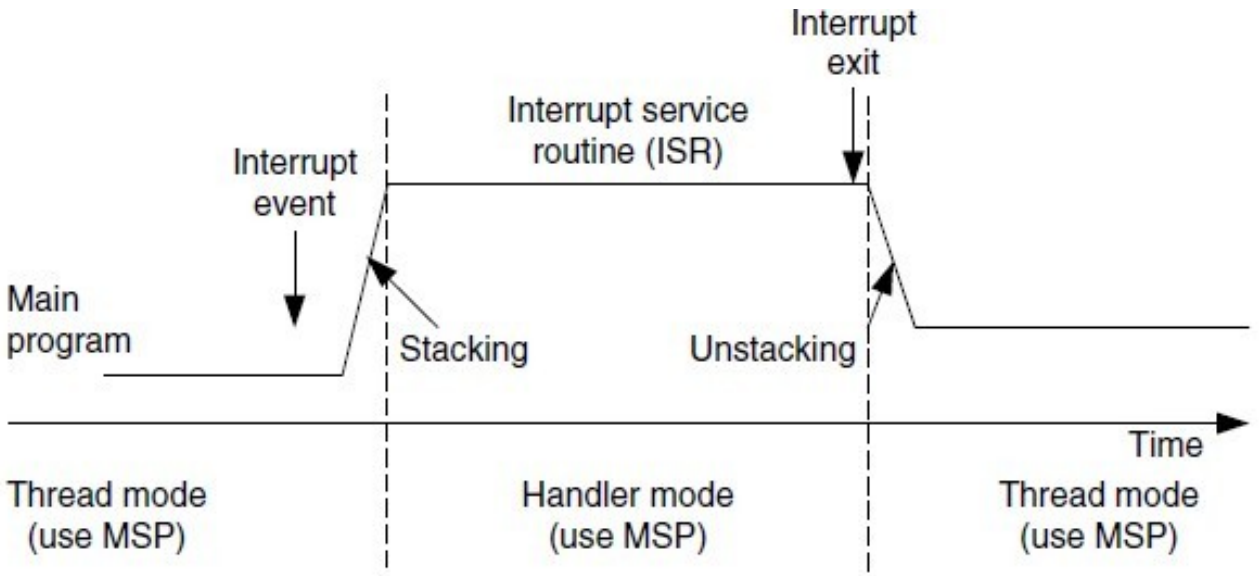


FIGURE 3.16

CONTROL[1]□0: Both Thread Level and Handler Use Main Stack.

The Two-Stack Model in the Cortex-M3 (continued)

- When the CONTROL[1] is 1, the PSP is used in thread mode.
  - In this arrangement, the main program and the exception handler can have separate stack memory regions.
  - This can prevent a stack error in a user application from damaging the stack used by the OS.
  - The automatic stacking and unstacking mechanism will use PSP, whereas stack operations inside the handler will use MSP.

The Two-Stack Model in the Cortex-M3 (continued)

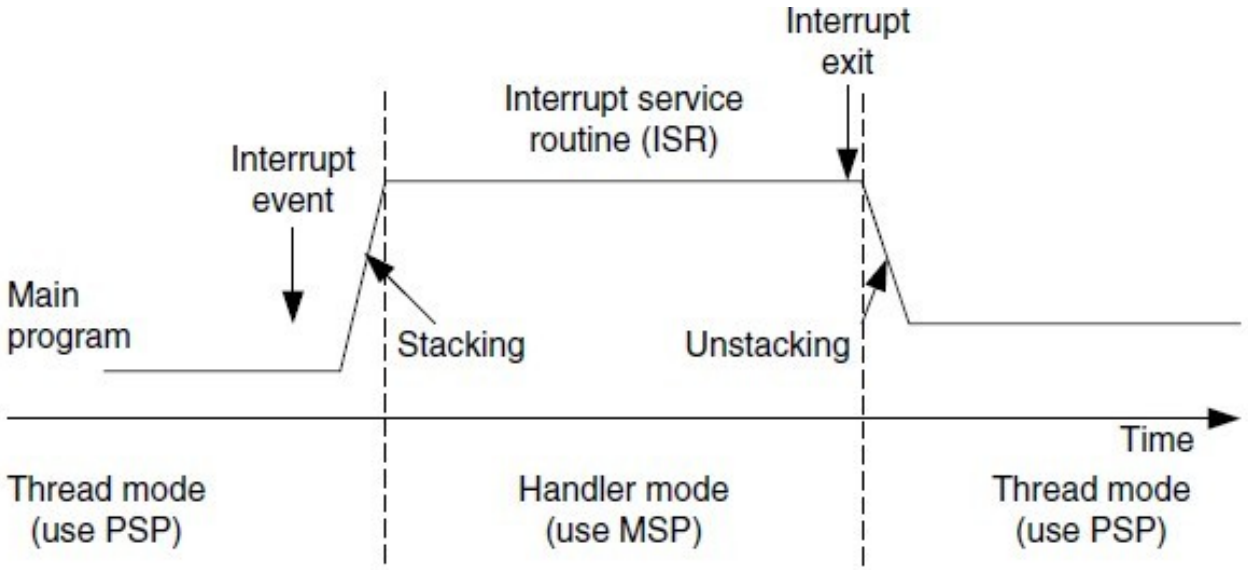


FIGURE 3.17

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

The Two-Stack Model in the Cortex-M3 (continued)

- It is possible to perform read/write operations directly to the MSP and PSP, without any confusion of which R13 you are referring to.
- Provided that you are in privileged level, you can access MSP and PSP values:

```
x = __get_MSP(); // Read the value of MSP
__set_MSP(x); // Set the value of MSP
x = __get_PSP(); // Read the value of PSP
__set_PSP(x); // Set the value of PSP
```

- In general, it is not recommended to change current selected SP values in a C function, as the stack memory could be used for storing local variables.

The Two-Stack Model in the Cortex-M3 (continued)

- To access the SPs in assembly, you can use the MRS and MSR instructions:

```
MRS R0, MSP  ; Read Main Stack Pointer to R0
MSR MSP, R0  ; Write R0 to Main Stack Pointer
MRS R0, PSP  ; Read Process Stack Pointer to R0
MSR PSP, R0  ; Write R0 to Process Stack Pointer
```

- By reading the PSP value using an MRS instruction, the OS can read data stacked by the user application (such as register contents before SVC).
- In addition, the OS can change the PSP pointer value—for example, during context switching in multitasking systems.

Reset Sequence

- After the processor exits reset, it will read two words from memory
  - Address 0x00000000: Starting value of R13 (the SP)
  - Address 0x00000004: Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state)

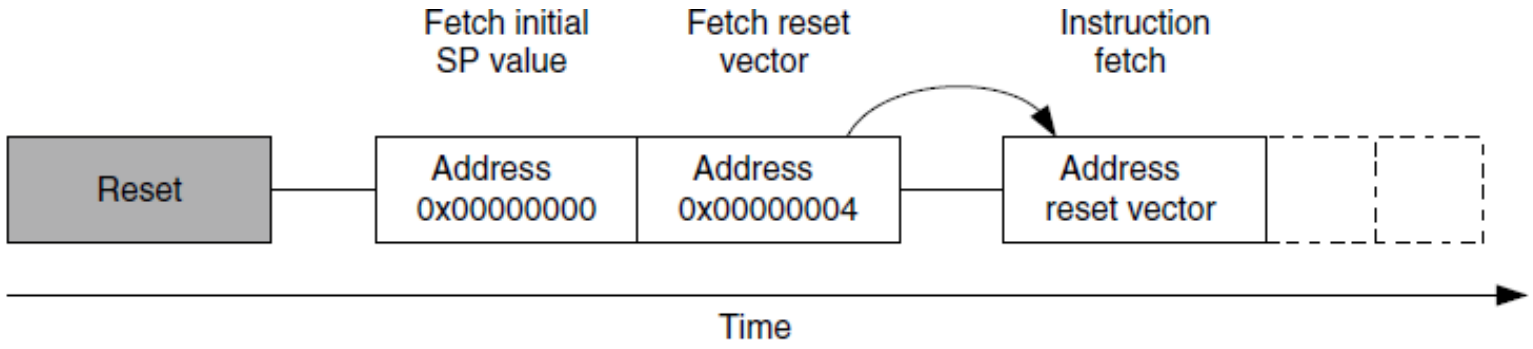


FIGURE 3.18

Reset Sequence.

Reset Sequence (continued)

- Because the stack operation in the Cortex-M3 is a full descending stack (SP decrement before store), the initial SP value should be set to the first memory after the top of the stack region.
- For example, if you have a stack memory range from 0x20007C00 to 0x20007FFF (1 KB), the initial stack value should be set to 0x20008000.

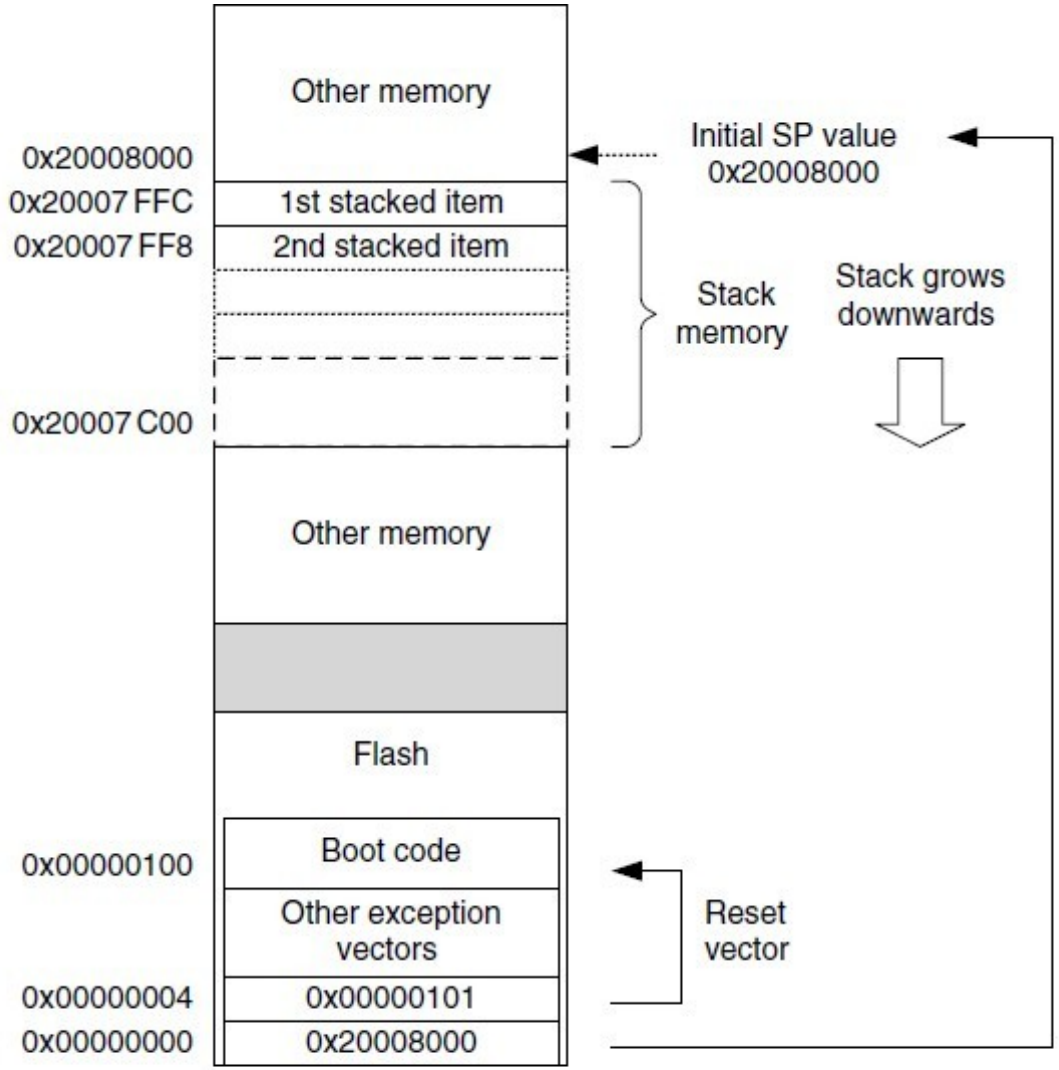


FIGURE 3.19

Initial Stack Pointer Value and Initial Program Counter Value Example.

Reset Sequence (continued)

- The vector table starts after the initial SP value.
- The first vector is the reset vector.
- In the Cortex-M3, vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code.
- For that reason, the previous example has 0x101 in the reset vector, whereas the boot code starts at address 0x100 (see Figure 3.19).
- After the reset vector is fetched, the Cortex-M3 can then start to execute the program from the reset vector address and begin normal operations.
- It is necessary to have the SP initialized, because some of the exceptions (such as NMI) can happen right after reset, and the stack memory could be required for the handler of those exceptions.

References

1. Joseph Yiu, ***"The Definitive Guide to the ARM Cortex-M3"***, 2nd Edition, Newnes (Elsevier), 2010.
2. https://www.arm.com

ARM MICROCONTROLLER & EMBEDDED SYSTEMS (18EC62)

MODULE – 2

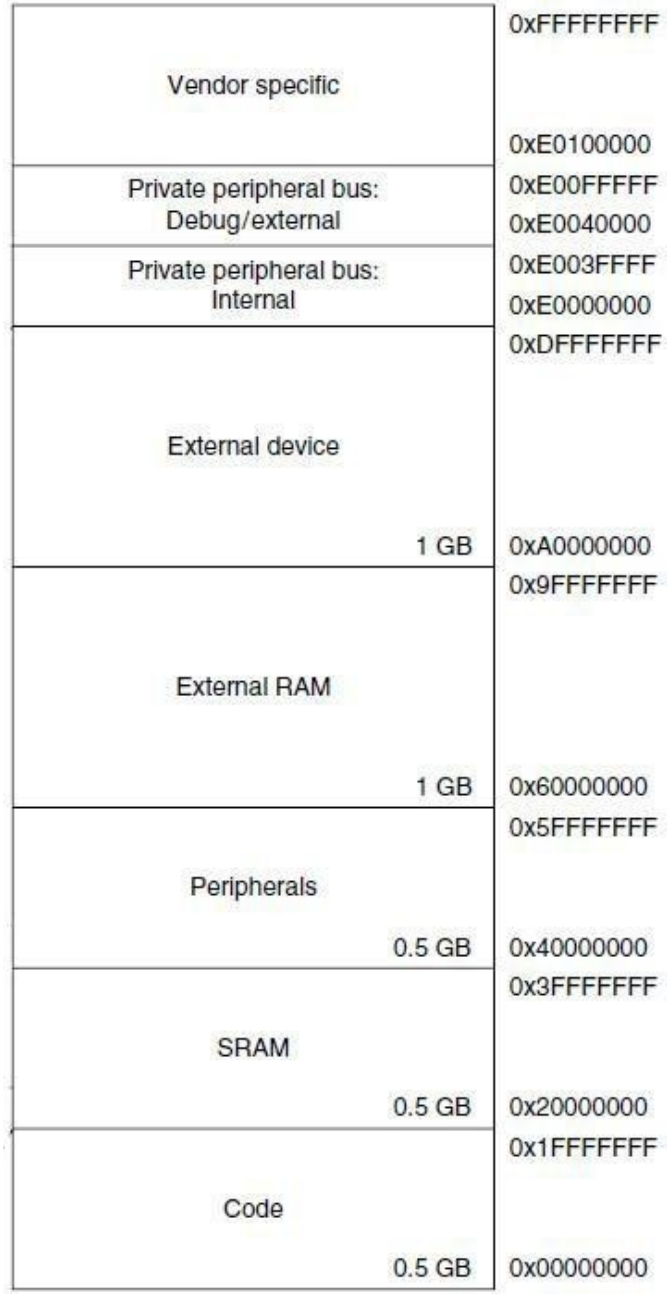
Memory Mapping, Bit-Band Operations and CMSIS

Memory System Features Overview

- The Cortex-M3 processor has different memory architecture from that of traditional ARM processors.
- First, it has a predefined memory map that specifies which bus interface is to be used when a memory location is accessed.
  - This feature also allows the processor design to optimize the access behavior when different devices are accessed.
- Another feature of the memory system in the Cortex-M3 is the bit-band support.
  - This provides atomic operations to bit data in memory or peripherals.
  - The bit-band operations are supported only in special memory regions.
- The Cortex-M3 memory system also supports unaligned transfers and exclusive accesses.
  - These features are part of the v7-M architecture.
- Finally, the Cortex-M3 supports both little endian and big endian memory configuration.

Memory Maps

- The Cortex-M3 processor has a fixed memory map.
- This makes it easier to port software from one Cortex-M3 product to another.
- For example, components described in previous sections, such as Nested Vectored Interrupt Controller (NVIC) and Memory Protection Unit (MPU), have the same memory locations in all Cortex-M3 products.
- Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.
- Some of the memory locations are allocated for private peripherals such as debugging components.
  - They are located in the private peripheral memory region.


Vendor specific
0xFFFFFFFF
0xE0100000
Private peripheral bus: Debug/external
0xE00FFFFF
0xE0040000
Private peripheral bus: Internal
0xE003FFFF
0xE0000000
External device
1 GB
0xDFFFFFFF
0xA0000000
External RAM
1 GB
0x9FFFFFFF
0x60000000
Peripherals
0.5 GB
0x5FFFFFFF
0x40000000
SRAM
0.5 GB
0x3FFFFFFF
0x20000000
Code
0.5 GB
0x1FFFFFFF
0x00000000

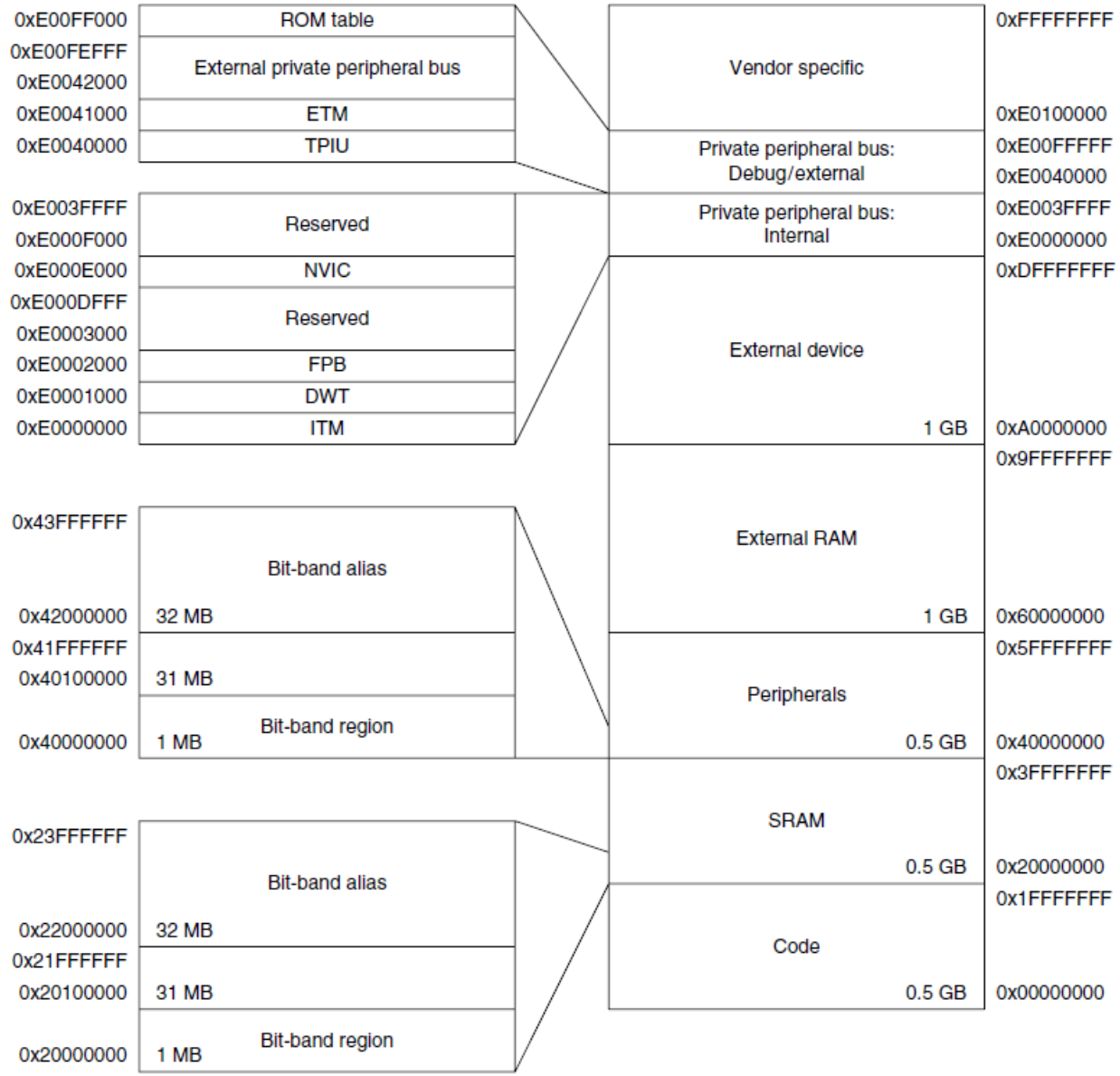


FIGURE 5.1

Cortex-M3 Predefined Memory Map.

Memory Maps (continued)

- The Cortex-M3 processor has a total of 4 GB of address space.
- Program code can be located in the code region, the Static Random Access Memory (SRAM) region, or the external RAM region.
- However, it is best to put the program code in the code region because with this arrangement, the instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces.

Memory Maps (continued)

- The SRAM memory range is for connecting internal SRAM.
- Access to this region is carried out via the system interface bus.
- In this region, a 32-MB range is defined as a bit-band alias.
- Within the 32-bit-band alias memory range, each word address represents a single bit in the 1-MB bit-band region.
- A data write access to this bit-band alias memory range will be converted to an atomic READ-MODIFY-WRITE operation to the bit-band region so as to allow a program to set or clear individual data bits in the memory.
- The bit-band operation applies only to data accesses not instruction fetches.
- By putting Boolean information (single bits) in the bit-band region, we can pack multiple Boolean data in a single word while still allowing them to be accessible individually via bit-band alias, thus saving memory space without the need for handling READ-MODIFY-WRITE in software.

Memory Maps (continued)

- Another 0.5-GB block of address range is allocated to on-chip peripherals.
- Similar to the SRAM region, this region supports bit-band alias and is accessed via the system bus interface.
- However, instruction execution in this region is not allowed.
- The bit-band support in the peripheral region makes it easy to access or change control and status bits of peripherals, making it easier to program peripheral control.

Memory Maps (continued)

- Two slots of 1-GB memory space are allocated for external RAM and external devices.
- The difference between the two is that program execution in the external device region is not allowed, and there are some differences with the caching behaviors.

Memory Maps (continued)

- The last 0.5-GB memory is for the system-level components, internal peripheral buses, external peripheral bus, and vendor-specific system peripherals.
- There are two segments of the private peripheral bus (PPB):
  - Advanced High-Performance Bus (AHB) PPB, for Cortex-M3 internal AHB peripherals only
    - This includes NVIC, FPB, DWT, and ITM
  - Advance Peripheral Bus (APB) PPB, for Cortex-M3 internal APB devices as well as external peripherals (external to the Cortex-M3 processor)
    - The Cortex-M3 allows chip vendors to add additional on-chip APB peripherals on this private peripheral bus via an APB interface

Memory Maps (continued)

- The NVIC is located in a memory region called the system control space (SCS).
- Besides providing interrupt control features, this region also provides the control registers for SYSTICK, MPU, and code debugging control.

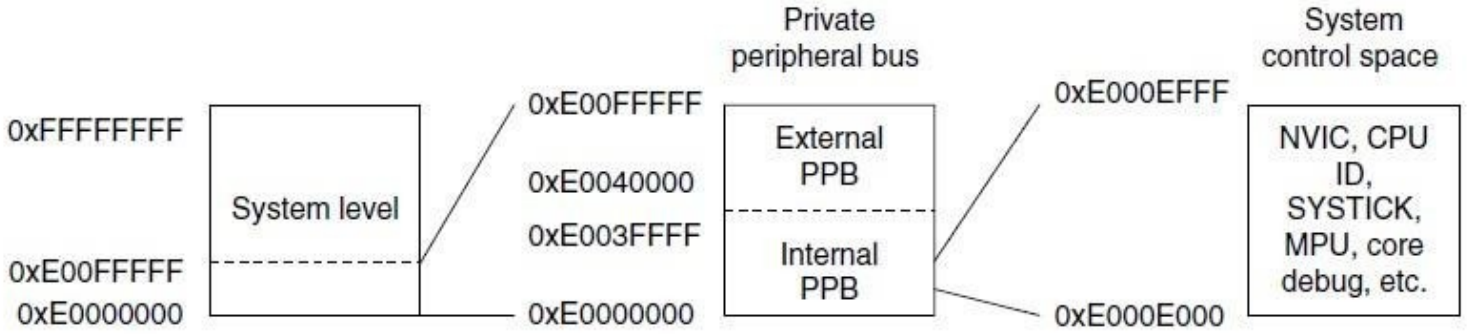


FIGURE 5.2

The System Control Space.

Memory Maps (continued)

- The remaining unused vendor-specific memory range can be accessed via the system bus interface.
  - However, instruction execution in this region is not allowed.
- The Cortex-M3 processor also comes with an optional MPU.
  - Chip manufacturers can decide whether to include the MPU in their products.

Memory Access Attributes

- The memory map shows what is included in each memory region.
- Aside from decoding which memory block or device is accessed, the memory map also defines the memory attributes of the access.
- The memory attributes you can find in the Cortex-M3 processor include the following:
  - *Bufferable:* Write to memory can be carried out by a write buffer while the processor continues on next instruction execution.
  - *Cacheable:* Data obtained from memory read can be copied to a memory cache so that next time it is accessed the value can be obtained from the cache to speed up the program execution.
  - *Executable:* The processor can fetch and execute program code from this memory region.
  - *Sharable:* Data in this memory region could be shared by multiple bus masters. Memory system needs to ensure coherency of data between different bus masters in shareable memory region.

Memory Access Attributes (continued)

- The memory access attributes for each memory region are as follows:
  - *Code memory region* (0x00000000–0x1FFFFFFF): This region is executable, and the cache attribute is write through (WT). You can put data memory in this region as well. If data operations are carried out for this region, they will take place via the data bus interface. Write transfers to this region are bufferable.
  - *SRAM memory region* (0x20000000–0x3FFFFFFF): This region is intended for on-chip RAM. Write transfers to this region are bufferable, and the cache attribute is write back, write allocated (WB-WA). This region is executable, so you can copy program code here and execute it.
  - *Peripheral region* (0x40000000–0x5FFFFFFF): This region is intended for peripherals. The accesses are noncacheable. You cannot execute instruction code in this region (Execute Never, or XN in ARM documentation, such as the Cortex-M3 TRM).

Memory Access Attributes (continued)

- *External RAM region* (0x60000000–0x7FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WB-WA), and you can execute code in this region.
- *External RAM region* (0x80000000–0x9FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WT), and you can execute code in this region.
- *External devices* (0xA0000000–0xBFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- *External devices* (0xC0000000–0xDFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- *System region* (0xE0000000–0xFFFFFFFF): This region is for private peripherals and vendor-specific devices. It is nonexecutable. For the PPB memory range, the accesses are strongly ordered (noncacheable, nonbufferable). For the vendor-specific memory region, the accesses are bufferable and noncacheable.

Default Memory Access Permissions

- The Cortex-M3 memory map has a default configuration for memory access permissions.
- This prevents user programs (non-privileged) from accessing system control memory spaces such as the NVIC.
- The default memory access permission is used when either no MPU is present or MPU is present but disabled.
- If MPU is present and enabled, the access permission in the MPU setup will determine whether user accesses are allowed.

Table 5.1 Default Memory Access Permissions

Memory Region	Address	Access in User Program
Vendor specific	0xE0100000–0xFFFFFFFF	Full access
ROM table	0xE00FF000–0xE00FFFFF	Blocked; user access results in bus fault
External PPB	0xE0042000–0xE00FEFFF	Blocked; user access results in bus fault
ETM	0xE0041000–0xE0041FFF	Blocked; user access results in bus fault
TPIU	0xE0040000–0xE0040FFF	Blocked; user access results in bus fault
Internal PPB	0xE000F000–0xE003FFFF	Blocked; user access results in bus fault
NVIC	0xE000E000–0xE000EFFF	Blocked; user access results in bus fault, except Software Trigger Interrupt Register that can be programmed to allow user accesses
FPB	0xE0002000–0xE0003FFF	Blocked; user access results in bus fault
DWT	0xE0001000–0xE0001FFF	Blocked; user access results in bus fault
ITM	0xE0000000–0xE0000FFF	Read allowed; write ignored except for stimulus ports with user access enabled
External device	0xA0000000–0xDFFFFFFF	Full access
External RAM	0x60000000–0x9FFFFFFF	Full access
Peripheral	0x40000000–0x5FFFFFFF	Full access
SRAM	0x20000000–0x3FFFFFFF	Full access
Code	0x00000000–0x1FFFFFFF	Full access

When a user access is blocked, the fault exception takes place immediately.

Bit-Band Operations

- Bit-band operation support allows a single load/store operation to

- access (read/write) to a single data bit.
- In the Cortex-M3, this is supported in two predefined memory regions called bit-band regions.
- One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region.
- These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the bit- band alias.
- When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.

Bit-Band Operations (continued)

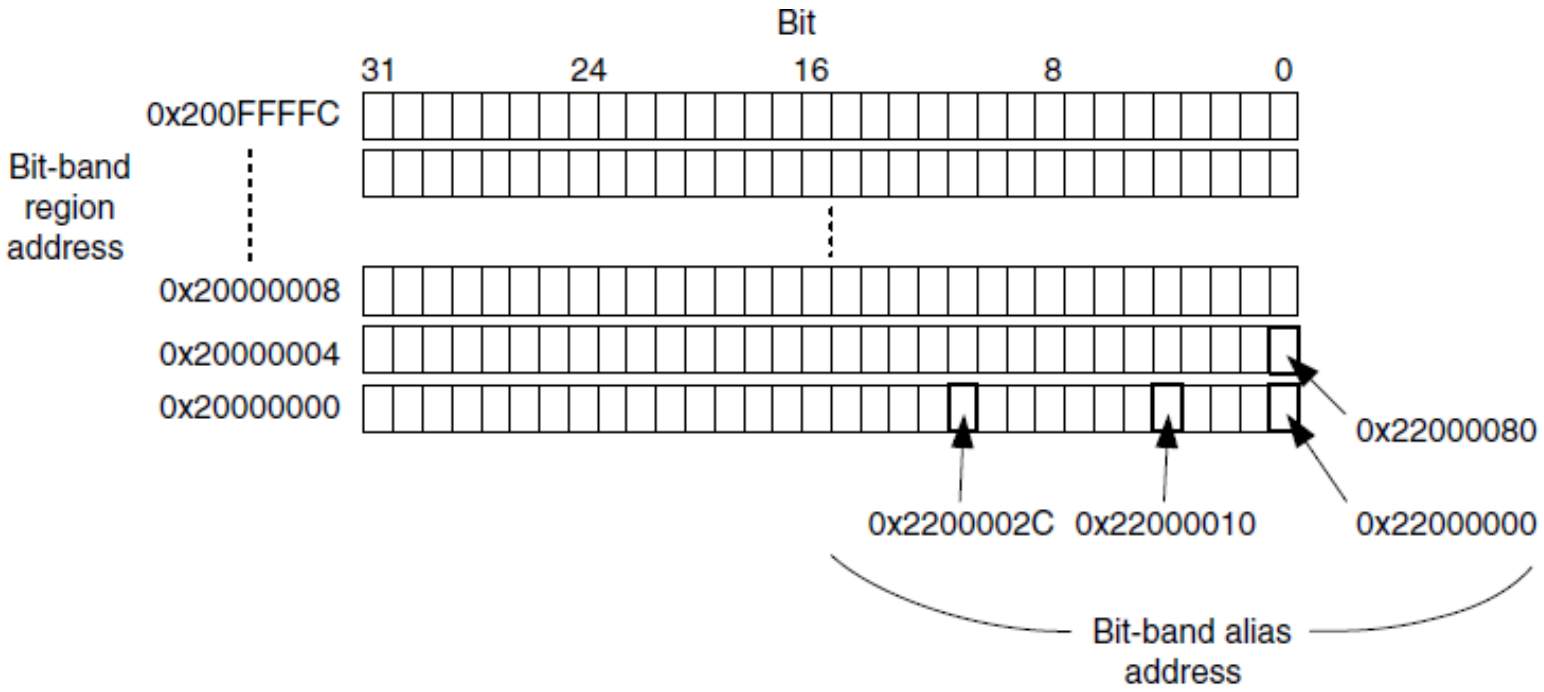


FIGURE 5.3

Bit Accesses to Bit-Band Region via the Bit-Band Alias.

Bit-Band Operations (continued)

- For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction (see Figure 5.4).
- The assembler sequence for these two cases could be like the one shown in Figure 5.5.

Bit-Band Operations (continued)

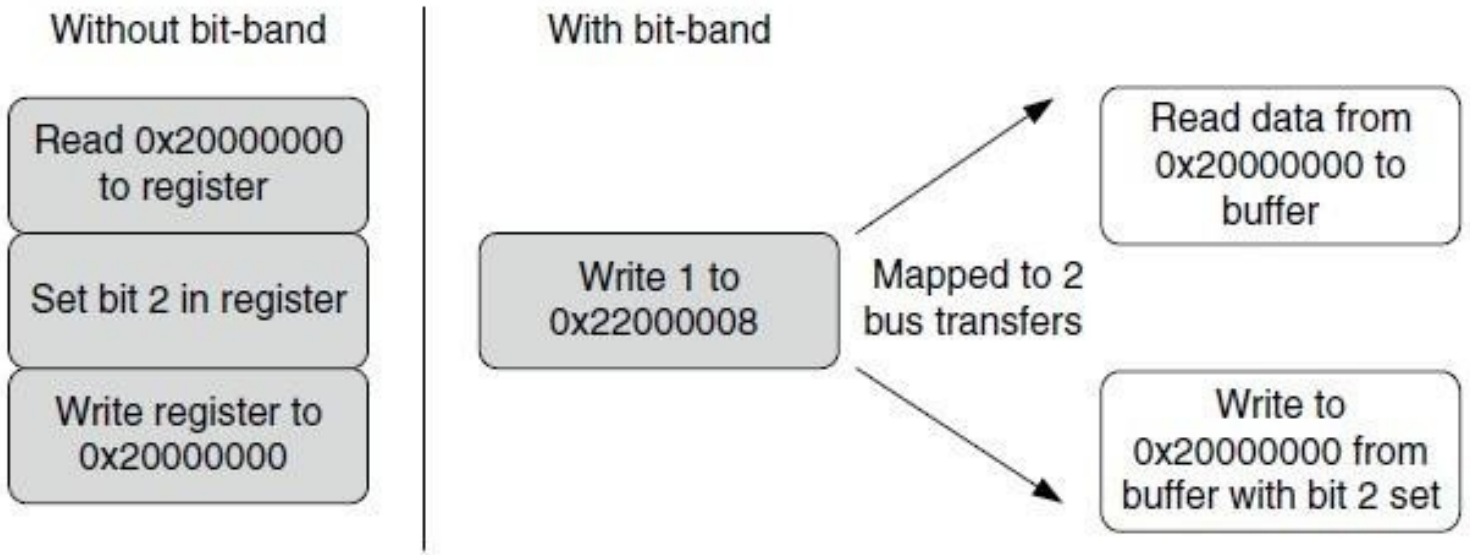


FIGURE 5.4

Write to Bit-Band Alias.

Bit-Band Operations (continued)

Without bit-band

```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0]       ; Write back result
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup add
MOV   R1, #1         ; Setup dat
STR   R1, [R0]       ; Write
```

FIGURE 5.5

Example Assembler Sequence to Write a Bit with and without Bit-Band.

Bit-Band Operations (continued)

- Similarly, bit-band support can simplify application code if we need to read a bit in a memory location.
- For example, if we need to determine bit 2 of address 0x20000000, we use the steps outlined in Figure 5.6.
- The assembler sequence for these two cases could be like the one shown in Figure 5.7.

Bit-Band Operations (continued)

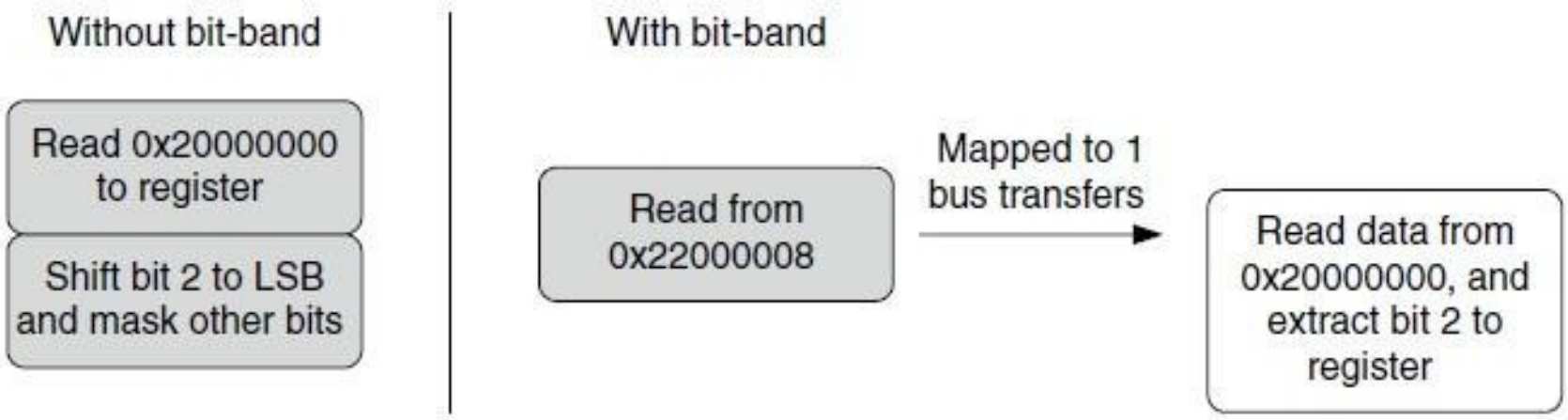


FIGURE 5.6

Read from the Bit-Band Alias.

Bit-Band Operations (continued)

Without bit-band

```
LDR     R0,=0x20000000 ; Setup address
LDR     R1, [R0]       ; Read
UBFX.W  R1,  R1, #2, #1 ; Extract bit[2]
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup address
LDR   R1, [R0]       ; Read
```

FIGURE 5.7

Read from the Bit-Band Alias.

Bit-Band Operations (continued)

- The Cortex-M3 uses the following terms for the bit-band memory addresses:
  - Bit-band region: This is a memory address region that supports bit-band operation.
  - Bit-band alias: Access to the bit-band alias will cause an access (a bit-band operation) to the bit-band region.
    - Note: A memory remapping is performed.

Bit-Band Operations (continued)

- Within the bit-band region, each word is represented by an LSB of 32 words in the bit-band alias address range.
- What actually happens is that when the bit-band alias address is accessed, the address is remapped into a bit-band address.
- For read operations, the word is read and the chosen bit location is shifted to the LSB of the read return data.
- For write operations, the written bit data are shifted to the required bit position, and a READ-MODIFY-WRITE is performed.
- There are two regions of memory for bit-band operations:
  - 0x20000000–0x200FFFFF (SRAM, 1 MB)
  - 0x40000000–0x400FFFFF (peripherals, 1 MB)

Bit-Band Operations (continued)

- For the SRAM memory region, the remapping of the bit-band alias is shown in Table 5.2.

Table 5.2 Remapping of Bit-Band Addresses in SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
…	…
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
…	…
0x20000004 bit[31]	0x220000FC bit[0]
…	…
0x200FFFFC bit[31]	0x23FFFFFC bit[0]

Bit-Band Operations (continued)

- Similarly, the bit-band region of the peripheral memory region can be accessed via bit-band aliased addresses, as shown in Table 5.3.

Table 5.3 Remapping of Bit-Band Addresses in Peripheral Memory Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
…	…
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
…	…
0x40000004 bit[31]	0x420000FC bit[0]
…	…
0x400FFFFC bit[31]	0x43FFFFFC bit[0]

Bit-Band Operations (continued)

- Here's a simple example:
  1. Set address 0x20000000 to a value of 0x3355AACC.
  2. Read address 0x22000008. This read access is remapped into read access to 0x20000000. The return value is 1 (bit[2] of 0x3355AACC).
  3. Write 0x0 to 0x22000008. This write access is remapped into a READ-MODIFY-WRITE to 0x20000000. The value 0x3355AACC is read from memory, bit 2 is cleared, and a result of 0x3355AAC8 is written back to address 0x20000000.
  4. Now, read 0x20000000. That gives you a return value of 0x3355AAC8 (bit[2] cleared).

CMSIS

- The Cortex-M3 microcontrollers are gaining momentum in the embedded application market, as more and more products based on the Cortex-M3 processor and software that support the Cortex-M3 processor are emerging.
- There are also a number of companies providing embedded software solutions, including codecs, data processing libraries, and various software and debug solutions.
- The CMSIS was developed by ARM to allow users of the Cortex-M3 microcontrollers to gain the most benefit from all these software solutions and to allow them to develop their embedded application quickly and reliably.

CMSIS (continued)

- The Cortex Microcontroller Software Interface Standard (CMSIS) was started in 2008 to improve software usability and inter-operability of ARM microcontroller software.
- It is integrated into the driver libraries provided by silicon vendors, providing a standardized software interface for the Cortex-M3 processor features, as well as a number of common system and I/O functions.
- The library is also supported by software companies including embedded OS vendors and compiler vendors.

CMSIS (continued)

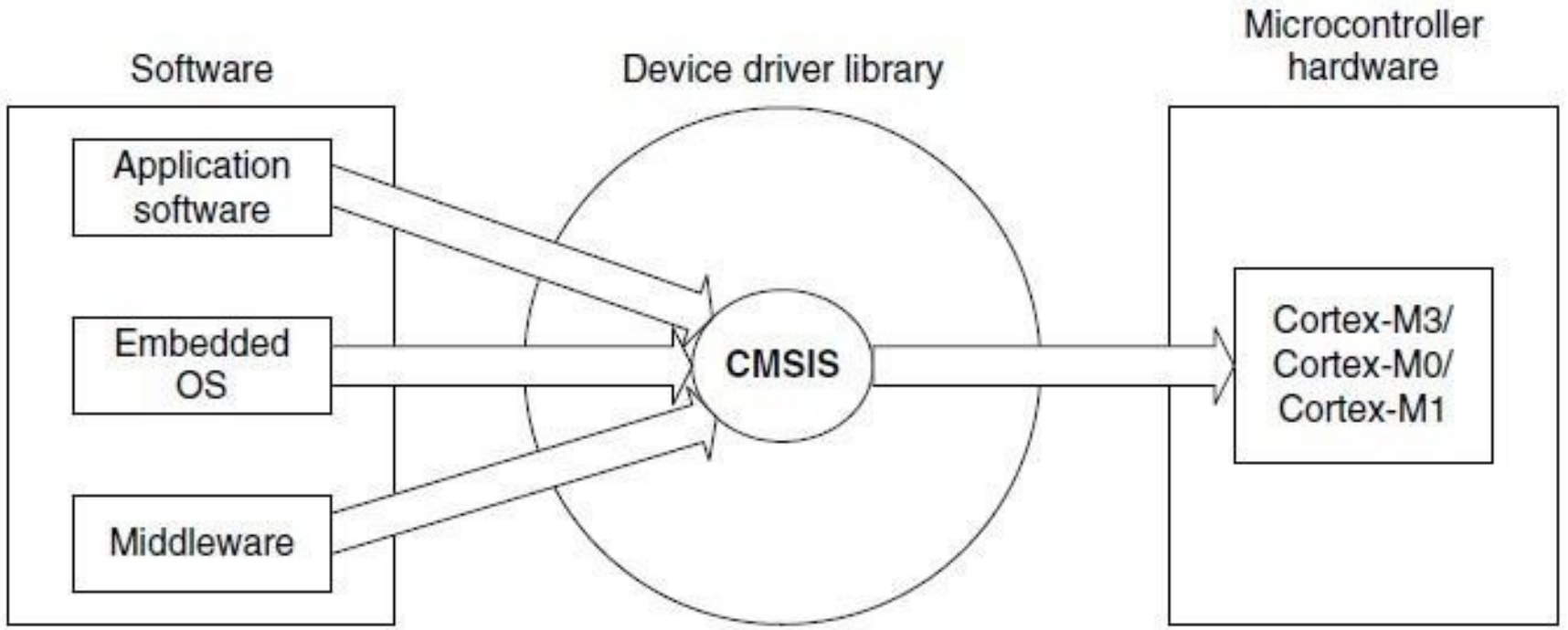


FIGURE 10.6

CMSIS Provides a Standardized Access Interface for Embedded Software Products.

CMSIS (continued)

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

CMSIS – Areas of Standardization

- The scope of CMSIS involves standardization in the following areas:
  - *Hardware Abstraction Layer (HAL) for Cortex-M processor registers:* This includes standardized register definitions for NVIC, System Control Block registers, SYSTICK register, MPU registers, and a number of NVIC and core feature access functions.
  - *Standardized system exception names:* This allows OS and middleware to use system exceptions easily without compatibility issues.
  - *Standardized method of header file organization:* This makes it easier for users to learn new Cortex microcontroller products and improve software portability.
  - *Common method for system initialization:* Each Microcontroller Unit (MCU) vendor provides a *SystemInit()* function in their device driver library for essential setup and configuration, such as initialization of clocks.
    - Again, this helps new users to start to use Cortex-M microcontrollers and aids software portability.

CMSIS – Areas of Standardization (continued)

- *Standardized intrinsic functions:* Intrinsic functions are normally used to produce instructions that cannot be generated by IEC/ISO C.
  - By having standardized intrinsic functions, software reusability and portability are considerably improved.
- *Common access functions for communication:* This provides a set of software interface functions for common communication interfaces including universal asynchronous receiver/transmitter (UART), Ethernet, and Serial Peripheral Interface (SPI).
  - By having these common access functions in the device driver library, reusability and portability of embedded software are improved.
- *Standardized way for embedded software to determine system clock frequency:* A software variable called *SystemFrequency* is defined in device driver code.
  - This allows embedded OS to set up the SYSTICK unit based on the system clock frequency.

Organization of CMSIS

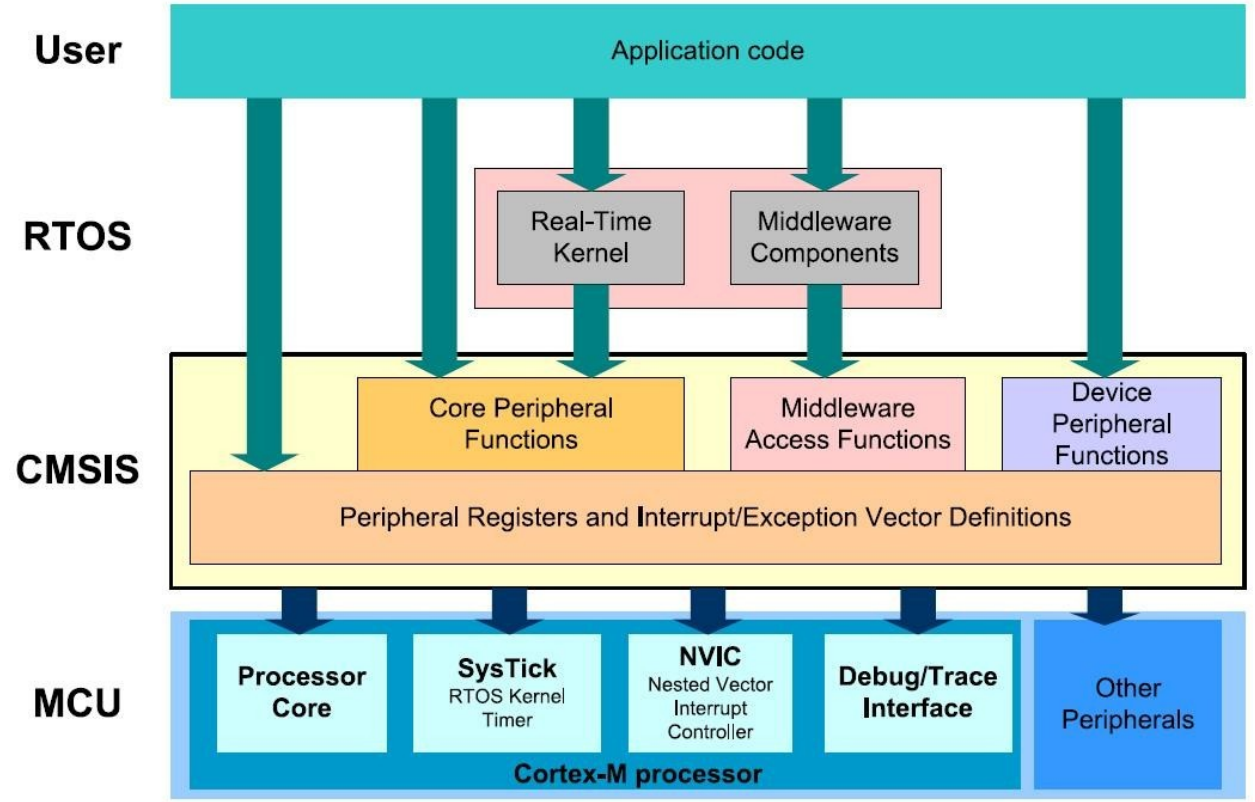


FIGURE 10.7

CMSIS Structure.

Organization of CMSIS (continued)

- The CMSIS is divided into multiple layers as follows:
  - Core Peripheral Access Layer
    - Name definitions, address definitions, and helper functions to access core registers and core peripherals
  - Middleware Access Layer
    - Common method to access peripherals for the software industry
    - Targeted communication interfaces include Ethernet, UART, and SPI.
    - Allows portable software to perform communication tasks on any Cortex microcontrollers that support the required communication interface

Organization of CMSIS (continued)

- Device Peripheral Access Layer (MCU specific)
  - Name definitions, address definitions, and driver code to access peripherals
- Access Functions for Peripherals (MCU specific)
  - Optional additional helper functions for peripherals
- The role of these layers is summarized in Figure 10.7.

Organization of CMSIS (continued)

- Device Peripheral Access Layer (MCU specific)
  - Name definitions, address definitions, and driver code to access peripherals
- Access Functions for Peripherals (MCU specific)
  - Optional additional helper functions for peripherals
- The role of these layers is summarized in Figure 10.7.

Advanced Microcontroller Bus Architecture (AMBA)

- The Advanced Microcontroller Bus Architecture (AMBA) specification defines an on-chip communications standard for high-performance embedded microcontrollers.
- Three distinct buses are defined within the AMBA specification :
  - Advanced High-performance Bus (AHB)
  - Advanced System Bus (ASB)
  - Advanced Peripheral Bus (APB)
- A test methodology is included with the AMBA specification which provides an infrastructure for modular macrocell test and diagnostic access.

Advanced High-performance Bus (AHB)

- The AMBA AHB is for high-performance, high clock frequency system modules.
- The AHB acts as the high-performance system backbone bus.
- AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions.
- AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques.

Advanced System Bus (ASB)

- The AMBA ASB is for high-performance system modules.
- AMBA ASB is an alternative system bus suitable for use where the high-performance features of AHB are not required.
- ASB also supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power Peripheral macrocell functions.

Advanced Peripheral Bus (APB)

- The AMBA APB is for low-power peripherals.
- AMBA APB is optimized for minimal power consumption and reduced interface complexity to support peripheral functions.
- APB can be used in conjunction with either version of the system bus.

Bus Interfaces on the Cortex-M3

- The bus interfaces on the Cortex-M3 processor are based on AHB-Lite and APB protocols.
- These are as follows:
  - The I-Code Bus
  - The D-Code Bus
  - The System Bus
  - The External PPB

The I-Code Bus

- The I-Code bus is a 32-bit bus based on the AHB-Lite bus protocol for instruction fetches in memory regions from 0x00000000 to 0x1FFFFFFF.
- Instruction fetches are performed in word size, even for 16-bit Thumb instructions.
- Therefore, during execution, the CPU core could fetch up to two Thumb instructions at a time.

The D-Code Bus

- The D-Code bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for data access in memory regions from 0x00000000 to 0x1FFFFFFF.
- Although the Cortex-M3 processor supports unaligned transfers, you won’t get any unaligned transfer on this bus, because the bus interface on the processor core converts the unaligned transfers into aligned transfers for you.
- Therefore, devices (such as memory) that attach to this bus need only support AHB-Lite (AMBA 2.0) aligned transfers.

The System Bus

- The system bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for instruction fetch and data access in memory regions from 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF.
- Similar to the D-Code bus, all the transfers on the system bus are aligned.

The External PPB

- The External PPB is a 32-bit bus based on the APB bus protocol.
- This is intended for private peripheral accesses in memory regions 0xE0040000 to 0xE00FFFFF.
- However, since some part of this APB memory is already used for TPIU, ETM, and the ROM table, the memory region that can be used for attaching extra peripherals on this bus is only 0xE0042000 to 0xE00FF000.
- Transfers on this bus are word aligned.

References

1. Joseph Yiu, ***"The Definitive Guide to the ARM Cortex-M3"***, 2nd Edition, Newnes (Elsevier), 2010.
2. https://www.arm.com

MODULE – 3

Embedded System Components

Introduction

What is an Embedded System?

- An embedded system is an electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).
- Every embedded system is unique and the hardware as well as the firmware is highly specialised to the application domain.

Embedded Systems vs. General Computing Systems

- The computing revolution began with the general purpose computing requirements. Later it was realised that the general computing requirements are not sufficient for the embedded computing requirements.
- The embedded computing requirements demand ‘something special’ in terms of response to stimuli, meeting the computational deadlines, power efficiency, limited memory capability, etc.

General Purpose Computing System	Embedded System
A system which is a combination of a generic hardware and a General Purpose Operating System for executing a variety of applications	A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
Contains a General Purpose Operating System (GPOS)	May or may not contain an operating system for functioning
Applications are alterable (programmable) by the user (It is possible for the end user to re-install the operating system, and also add or remove user applications)	The firmware of the embedded system is pre-programmed and it is non-alterable by the end-user (There may be exceptions for system supporting OS kernel image flashing through special hardware settings)
Performance is the key deciding factor in the selection of the system. Always, 'Faster is Better'	Application-specific requirements (like performance, power requirements, memory usage, etc.) are the key deciding factors
Less/not at all tailored towards reduced operating power requirements, options for different levels of power management	Highly tailored to take advantage of the power saving modes supported by the hardware and the operating system
Response requirements are not time-critical	For certain category of embedded systems like mission critical systems, the response time requirement is highly critical
Need not be deterministic in execution behaviour	Execution behaviour is deterministic for certain types of embedded systems like 'Hard Real Time' systems

History of Embedded Systems

- Embedded systems were in existence even before the IT revolution.
  - Built around the old vacuum tube and transistor technologies.
- Advances in semiconductor and nanotechnology and IT revolution gave way to the development of miniature embedded systems.
- The first recognised modern embedded system is the Apollo Guidance Computer (AGC) developed by the MIT Instrumentation Laboratory for the lunar expedition.
  - It had 36K words of fixed memory and 2K words of erasable memory.
  - The clock frequency of was 1.024 MHz and it was derived from a 2.048 MHz crystal clock.
- The first mass-produced embedded system was the Autonetics D-17 guidance computer for the Minuteman-I missile in 1961.
  - It was built using discrete transistor logic and a hard-disk for main memory.
- The first integrated circuit was produced in September 1958 and computers using them began to appear in 1963.

Classification of Embedded Systems

- Some of the criteria used in the classification of embedded systems are:
  1. Based on generation
  2. Complexity and performance requirements
  3. Based on deterministic behaviour
  4. Based on triggering

Classification Based on Generation

- First Generation
- Second Generation
- Third Generation
- Fourth Generation
- Next Generation

Classification Based on Generation (continued)

- First Generation
  - Early embedded systems were built around 8-bit microprocessors like 8085 and Z80 and 4-bit microcontrollers.
  - Simple in hardware circuits with firmware developed in assembly code.
  - E.g.: Digital telephone keypads, stepper motor control units, etc.

Classification Based on Generation (continued)

- Second Generation
  - Embedded systems built around 16-bit microprocessors and 8-bit or 16-bit microcontrollers.
  - Instruction set were much more complex and powerful than the first generation.
  - Some of the second generation embedded systems contained embedded operating systems for their operation.
  - E.g.: Data acquisition systems, SCADA systems, etc.

Classification Based on Generation (continued)

- Third Generation
  - Embedded systems built around 32-bit microprocessors and 16-bit microcontrollers.
  - Application and domain specific processors/controllers like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASICs) came into picture.
  - The instruction set of processors became more complex and powerful and the concept of instruction pipelining also evolved.
  - Dedicated embedded real time and general purpose operating systems entered into the embedded market.
  - Embedded systems spread its ground to areas like robotics, media, industrial process control, networking, etc.

Classification Based on Generation (continued)

- Fourth Generation
  - The advent of System on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturisation into the embedded device market.
  - The SoC technique implements a total system on a chip by implementing different functionalities with a processor core on an integrated circuit.
  - They make use of high performance real time embedded operating systems for their functioning.
  - E.g.: Smart phone devices, Mobile Internet Devices (MIDs), etc.

Classification Based on Generation (continued)

- Next Generation
  - The processor and embedded market is highly dynamic and demanding.
  - The next generation embedded systems are expected to meet growing demands in the market.

Classification Based on Complexity and Performance

- Small-Scale Embedded Systems
- Medium-Scale Embedded Systems
- Large-Scale Embedded Systems

Classification Based on Complexity and Performance (continued)

- Small-Scale Embedded Systems
  - Simple in application needs and the performance requirements are not time critical.
  - E.g.: An electronic toy
  - Usually built around low performance and low cost 8-bit or 16-bit microprocessors/microcontrollers.
  - May or may not contain an operating system for its functioning.

Classification Based on Complexity and Performance (continued)

- Medium-Scale Embedded Systems
  - Slightly complex in hardware and firmware (software) requirements.
  - Usually built around medium performance, low cost 16-bit or 32-bit microprocessors/microcontrollers or digital signal processors.
  - Usually contain an embedded operating system (either general purpose or real time operating system) for functioning.

Classification Based on Complexity and Performance (continued)

- Large-Scale Embedded Systems
  - Highly complex in hardware and firmware (software) requirements.
  - They are employed in mission critical applications demanding high performance.
  - Usually built around high performance 32-bit or 64-bit RISC processors/controllers or Reconfigurable System on Chip (RSoC) or multi-core processors and programmable logic devices.
  - May contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system.
  - Decoding/encoding of media, cryptographic function implementation, etc. are examples of processing requirements which can be implemented using a co-processor/hardware accelerator.
  - Usually contain a high performance real time operating system (RTOS) for task scheduling, prioritization and management.

Classification Based on Deterministic Behaviour

- Applicable for ‘Real Time’ systems.
- The application/task execution behaviour can be either deterministic or non-deterministic.
- Based on the execution behaviour, real time embedded systems are classified into Hard Real Time and Soft Real Time systems.

Classification Based on Triggering

- Embedded systems which are ‘Reactive’ in nature (like process control systems in industrial control applications) can be classified based on the trigger.
- Reactive systems can be either event-triggered or time-triggered.

Major Application Areas of Embedded Systems

1. Consumer electronics: Camcorders, cameras, etc.
2. Household appliances: Television, DVD players, washing machine, refrigerators, microwave oven, etc.
3. Home automation and security systems: Air conditioners, sprinklers, intruder detection alarms, closed circuit television (CCTV) cameras, fire alarms, etc.
4. Automotive industry: Anti-lock braking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
5. Telecom: Cellular telephones, telephone switches, handset multimedia applications, etc.

Major Application Areas of Embedded Systems (continued)

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. Measurements & Instrumentation: Digital multimeters, digital CROs, logic analyzers, PLC systems, etc.
10. Banking & Retail: Automated teller machines (ATM) and currency counters, point of sales (POS), etc.
11. Card readers: Barcode, smart card readers, hand held devices, etc.

Purpose of Embedded Systems

- Each embedded system is designed to serve the purpose of any one or a combination of the following tasks:
  1. Data Collection/Storage/Representation
  2. Data Communication
  3. Data (Signal) Processing
  4. Monitoring
  5. Control
  6. Application Specific User Interface

Purpose of Embedded Systems (continued)

- Data Collection/Storage/Representation
  - Embedded systems designed for the purpose of data collection performs acquisition of data from the external world.
  - Data collection is usually done for storage, analysis, manipulation and transmission.
  - The term "data" refers all kinds of information, viz. text, voice, image, video, electrical signals and any other measurable quantities.
  - Data can be either analog (continuous) or digital (discrete).
  - The collected data may be stored or transmitted or it may be processed or it may be deleted instantly after giving a meaningful representation.

- A digital camera is a typical example of an embedded system with data collection/storage/representation of data.
- Images are captured and the captured image may be stored within the memory of the camera.
- The captured image can also be presented to the user through a graphic LCD unit.

Purpose of Embedded Systems (continued)

- Data Communication
  - Embedded data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems.
  - The transmission is achieved either by a wire-line medium or by a wireless medium.
  - The data collecting embedded terminal itself can incorporate data communication units like wireless modules (Bluetooth, ZigBee, Wi-Fi, EDGE, GPRS, etc.) or wire-line modules (RS-232C, USB, TCP/IP, PS2, etc.).

Fig: A wireless network router for data communication

- Network hubs, routers, switches, etc. are typical examples of dedicated data transmission embedded systems.
- They act as mediators in data communication and provide various features like data security, monitoring etc.

Purpose of Embedded Systems (continued)

- Data (Signal) Processing
  - The data (voice, image, video, electrical signals and other measurable quantities) collected by embedded systems may be used for various kinds of data processing.
  - Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, synthesis, audio video codec, transmission applications, etc.

- A digital hearing aid is a typical example of an embedded system employing data processing.
- Digital hearing aid improves the hearing capacity of hearing impaired persons.

Purpose of Embedded Systems (continued)

- Monitoring
  - Almost embedded products coming under the medical domain are used for monitoring.
- A very good example is the electro cardiogram (ECG) machine for monitoring the heartbeat of a patient.
  - The machine is intended to do the monitoring of the heartbeat.
  - It cannot impose control over the heartbeat.
  - The sensors used in ECG are the different electrodes connected to the patient's body.
  - Some other examples of embedded systems with monitoring function are measuring instruments like digital CRO, digital multimeters, logic analyzers, etc. used in Control & Instrumentation applications.

Fig: A patient monitoring system for monitoring heartbeat

Purpose of Embedded Systems (continued)

- Control
  - Embedded systems with control functionalities impose control over some variables according to the changes in input variables.
  - A system with control functionality contains both sensors and actuators.
  - Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable.
  - The actuators connected to the output port are controlled according to the changes in input variable to put an impact on the controlling variable to bring the controlled variable to the specified range.

- An Air Conditioner System used to control the room temperature to a specified limit is a typical example for embedded system for control purpose.
- An air conditioner contains a room temperature-sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature.

Purpose of Embedded Systems (continued)

- 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.

- A mobile phone is an example for embedded system with an application-specific user interfaces.

A Typical Embedded System

A Typical Embedded System

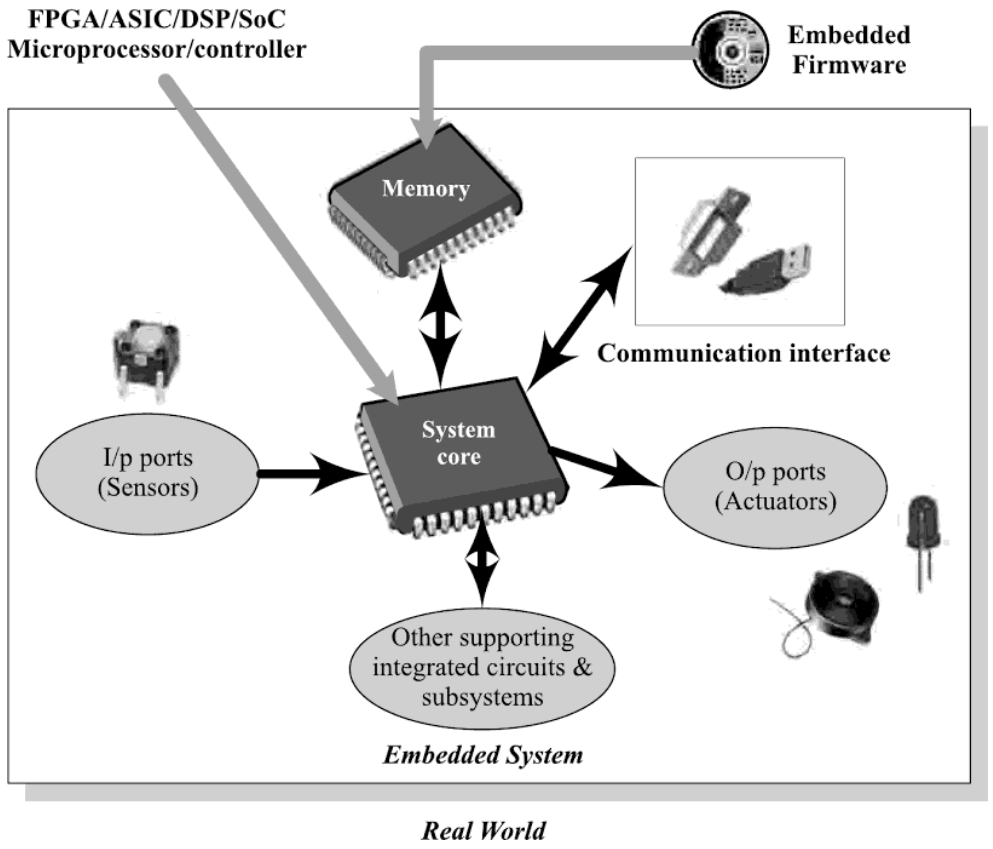


Fig: Elements of an Embedded System

A Typical Embedded System (continued)

- It contains a single chip controller, which acts as the master brain of the system.
- The controller can be
  - ✓ A microprocessor or
  - ✓ A microcontroller or
  - ✓ A Field Programmable Gate Array (FPGA) device or
  - ✓ A Digital Signal Processor (DSP) or
  - ✓ An Application Specific Integrated Circuit (ASIC)/Application Specific Standard Product (ASSP)

A Typical Embedded System (continued)

- An embedded system can be viewed as a reactive system.
- The control is achieved by processing the information coming from the sensors and user interfaces, and controlling some actuators that regulate the physical variable.
- Key boards, push button switches, etc. are examples for common user interface input devices.
- LEDs, liquid crystal displays, piezoelectric buzzers, etc. are examples for common user interface output devices for a typical embedded system.

A Typical Embedded System (continued)

- The memory of the system is responsible for holding the control algorithm and other important configuration details.
- For most of embedded systems, the memory for storing the algorithm or configuration data is of fixed type, which is a kind of Read Only Memory (ROM).
  - It is not available for the end user for modifications
  - The memory is protected from unwanted user interaction by implementing some kind of memory protection mechanism.
  - The most common types of memories used in embedded systems for control algorithm storage are OTP, PROM, UVEPROM, EEPROM and FLASH.
- Sometimes the system requires temporary memory for performing arithmetic operations or control algorithm execution and this type of memory is known as "working memory".
  - Random Access Memory (RAM) is used in most of the systems as the working memory.
  - Various types of RAM like SRAM, DRAM and NVRAM are used for this purpose.

A Typical Embedded System (continued)

- Apart from these, communication interface is essential for communicating with various subsystems of the embedded system and with the external world.
- The communication interfaces may be used to achieve onboard (I2C, SPI, UART, parallel bus interface, etc.) or external communication (wireless interfaces like Infrared, Bluetooth, Wi-Fi, etc.)

Core of the Embedded System

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
     1. Microprocessors
     2. Microcontrollers
     3. 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.
- Most of the embedded systems in the industrial control and monitoring applications make use of the commonly available microprocessors or microcontrollers.
- Domains which require signal processing such as speech coding, speech recognition, etc. make use of special kind of digital signal processors.

Microprocessors

- A Microprocessor is a silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions.
- 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, AMD, Freescale, IBM, TI, Cyrix, Hitachi, NEC, LSI Logic, etc. are the key players in the processor market.

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

General Purpose Processor (GPP)

- A General Purpose Processor or GPP is a processor designed for general computational tasks.
  - The processor running inside laptop or desktop is a typical example for general purpose processor.
- Due to the high volume production, the per unit cost for a chip is low.
- A typical general purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU).

Application-Specific Instruction Set Processor (ASIP)

- Application Specific Instruction Set Processors (ASIPs) are processors with architecture and instruction set optimised to specific-domain/application requirements like network processing, automotive, telecom, media applications, digital signal processing, control applications, etc.
- Most of the embedded systems are built around application specific instruction set processors.
  - Some microcontrollers (like automotive AVR, USB AVR from Atmel), system on chips, digital signal processors, etc. are examples for application specific instruction set processors (ASIPs).
- ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory.

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.
- A microcontroller contains all the necessary functional blocks for independent working.
  - Have greater place in embedded domain in place of microprocessors.
  - They are cheap, cost effective and are readily available in the market.
- Atmel, Texas Instruments, Toshiba, Philips, Freescale, NEC, Zilog, Hitachi, Mitsubishi, Infineon, ST Micro Electronics, National, Microchip, Analog Devices, Daewoo, Intel, Maxim, Sharp, Silicon Laboratories, TDK, Triscend, Winbond, etc. are the key players in the microcontroller market.

Microprocessor vs. Microcontroller

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

Digital Signal Processors

- Digital Signal Processors (DSPs) 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.
- 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.

Digital Signal Processors (continued)

- A typical digital signal processor incorporates the following key units:
- Program Memory: Memory for storing the program required by DSP to process the data
- Data Memory: Working memory for storing temporary variables and data/signal to be processed.
- Computational Engine: Performs the signal processing in accordance with the stored program memory.
  - It incorporates many specialised 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.
- 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.

RISC vs. CISC Processors/Controllers

- RISC stands for Reduced Instruction Set Computing.
  - All RISC processors/controllers possess lesser number of instructions, typically in the range of 30 to 40.
  - E.g.: Atmel AVR microcontroller – its instruction set contains only 32 instructions.
- CISC stands for Complex Instruction Set Computing.
  - The instruction set is complex and instructions are high in number.
  - E.g.: 8051 microcontroller – its instruction set contains 255 instructions.

RISC	CISC
Lesser number of instructions	Greater number of instructions
Instruction pipelining and increased execution speed	Generally no instruction pipelining feature
Orthogonal instruction set (Allows each instruction to operate on any register and use any addressing mode)	Non-orthogonal instruction set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction-specific)
Operations are performed on registers only, the only memory operations are load and store	Operations are performed on registers or memory depending on the instruction
A large number of registers are available	Limited number of general purpose registers
Programmer needs to write more code to execute a task since the instructions are simpler ones	Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Single, fixed length instructions	Variable length instructions
Less silicon usage and pin count	More silicon usage since more additional decoder logic is required to implement the complex instruction decoding
With Harvard Architecture	Can be Harvard or Von-Neumann Architecture

Harvard vs. Von-Neumann Processor/Controller Architecture

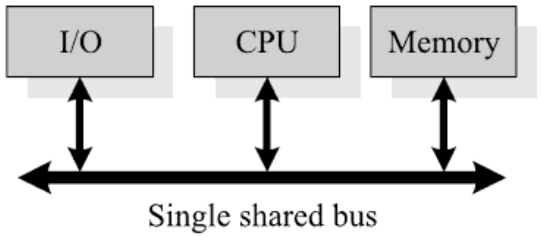
- Von-Neumann Architecture
  - Microprocessors/controllers based on the Von-Neumann architecture share a single common bus for fetching both instructions and data.
  - Program instructions and data are stored in a common main memory.
  - They first fetch an instruction and then fetch the data to support 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.

Harvard vs. Von-Neumann Processor/Controller Architecture (continued)

- Harvard Architecture
  - 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.
  - The data memory can be read and written while the 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.

Harvard vs. Von-Neumann Processor/Controller Architecture (continued)

Harvard Architecture	Von-Neumann Architecture
Separate buses for instruction and data fetching	Single shared bus for instruction and data fetching
Easier to pipeline, so high performance can be achieved	Low performance compared to Harvard architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory

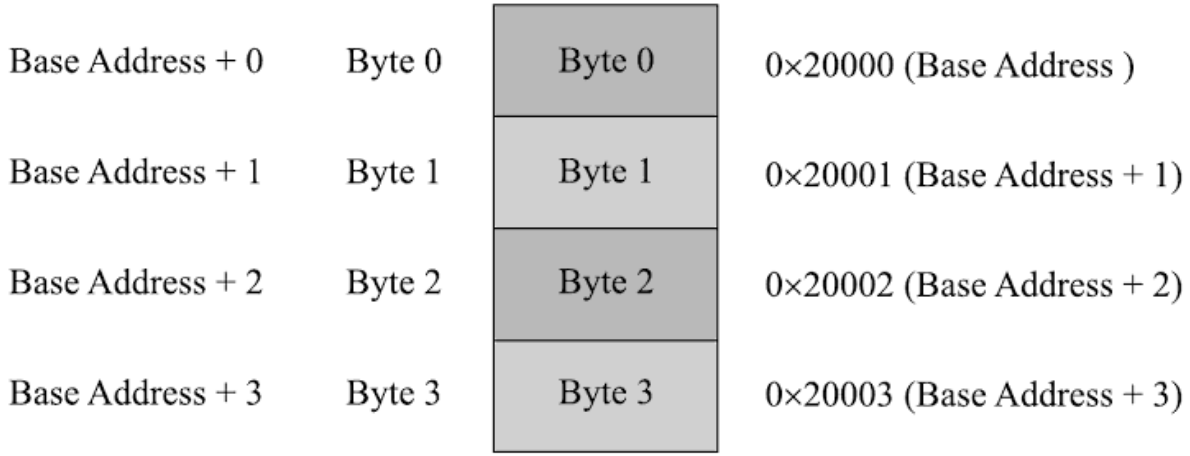


Big-Endian vs. Little-Endian Processors/Controllers

- Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system.
- 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.: Intel x86 Processors
  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.: Motorola 68000 Series Processors

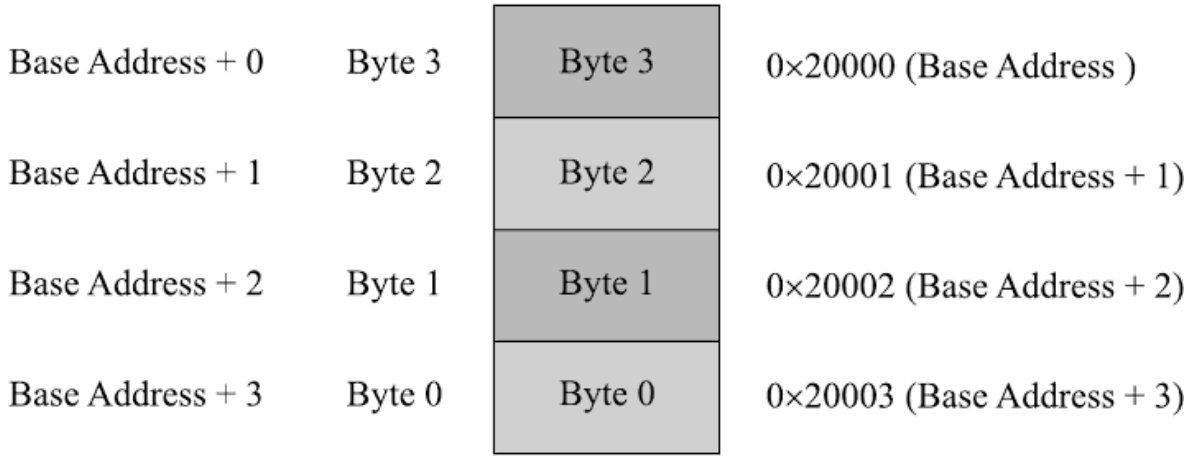
Big-Endian vs. Little-Endian Processors/Controllers (continued)

- Little-endian means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first.)
  - For example, a 4 byte long integer Byte3 Byte2 Byte1 Byte0 will be stored in the memory as shown below:



Big-Endian vs. Little-Endian Processors/Controllers (continued)

- Big-endian means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)
  - For example, a 4 byte long integer Byte3 Byte2 Byte1 Byte0 will be stored in the memory as shown below:



Load Store Operation and Instruction Pipelining

- 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:

Load Store Operation and Instruction Pipelining (continued)

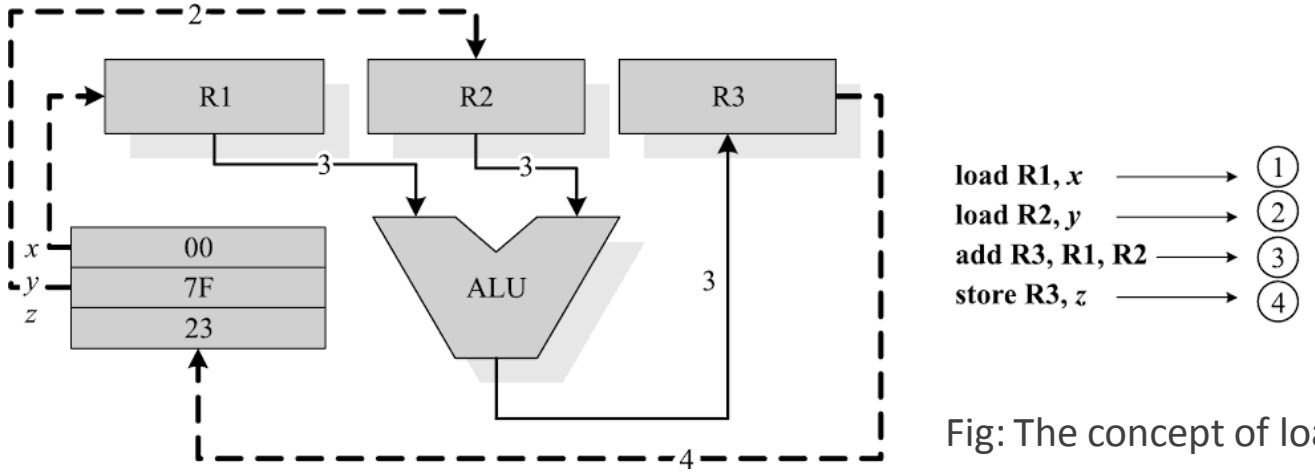


Fig: The concept of load store architecture

- 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 registers R1 and R2 and stores the result in register R3.
- The next instruction *store R3,z* stores the content of register R3 in memory location *z*.

Load Store Operation and Instruction Pipelining (continued)

- The conventional instruction execution by the processor follows the fetch-decode-execute sequence.
  - The fetch part fetches the instruction from program memory or code memory.
  - The decode part decodes the instruction to generate the necessary control signals.
  - 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. For simplicity let's consider decode and execution together.

Load Store Operation and Instruction Pipelining (continued)

- During the decode operation, the memory address bus is available and if it is possible to effectively utilise it for an instruction fetch, the processing speed can be increased.
- Instruction pipelining refers to the overlapped execution of instructions – i.e., while the current instruction is being decoded and executed, the next instruction will be fetched.
- If the current instruction in progress is a program control flow transfer instruction like jump or call instruction, the instruction fetched is flushed and a new instruction fetch is performed to fetch the instruction.
- 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.

Load Store Operation and Instruction Pipelining (continued)

- 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.
- Figure illustrates the concept of instruction pipelining for single stage pipelining.

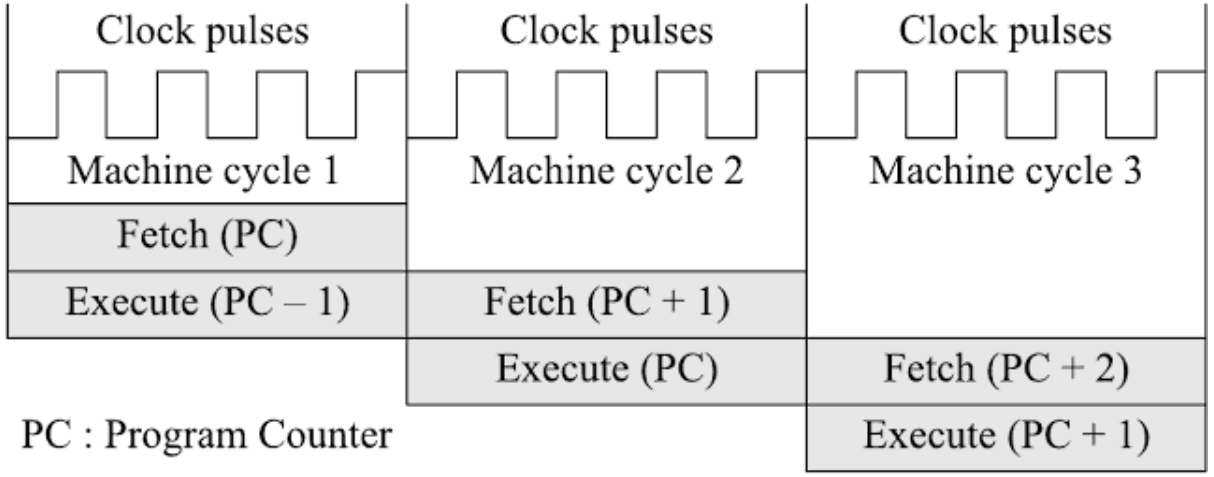


Fig: The single-stage pipelining concept

Application Specific Integrated Circuits (ASICs)

- Application Specific Integrated Circuit (ASIC) is a microchip designed to perform a specific or unique application.
  - 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.
- ASIC consumes a very small area in the total system.
  - Helps in the design of smaller systems with high capabilities/functionalities.
- Fabrication of ASICs requires a non-refundable initial investment for the process technology and configuration expenses. This investment is known as Non-Recurring Engineering Charge (NRE) and it is a one time investment.
- If the Non-Recurring Engineering Charges (NRE) is borne by a third party and the ASIC is made openly available in the market, the it is referred as Application Specific Standard Product (ASSP).
  - E.g.: ADE7760 Energy Meter ASIC developed by Analog Devices for Energy metering applications

Programmable Logic Devices

- Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.
- Logic devices can be classified into two broad categories—fixed and programmable.
- The circuits in a fixed logic device are permanent, they perform one function or set of functions—once manufactured, they cannot be changed.
- 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.
  - Designers use inexpensive software tools to quickly develop, simulate, and test their designs.
  - Then, a design can be quickly programmed into a device, and immediately tested in a live circuit.

Programmable Logic Devices (continued)

- There are no NRE costs and the final design is completed much faster than that of a custom, fixed logic device.
- Another key benefit of using PLDs is that during the design phase customers can change the circuitry as often as they want until the design operates to their satisfaction.
  - PLDs are based on re-writable memory technology to change the design, the device is simply reprogrammed.
- Once the design is final, customers can go into immediate production by simply programming as many PLDs as they need with the final software design file.
- The two major types of programmable logic devices are Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs).

CPLDs and FPGAs

FPGAs

- The 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 line of devices, provides eight million "system gates" (the relative density of logic).
- These advanced devices also offer features such as built-in hardwired processors (such as the IBM power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies.
- FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing.

CPLDs

- CPLDs offer much smaller amounts of logic—up to about 10,000 gates.
- But CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications.
- CPLDs such as the Xilinx CoolRunner series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

Advantages of PLD

- PLDs offer customers much more flexibility during the design cycle because design iterations are simply a matter of changing the programming file, and the 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 and are able to amortize those costs over the multi-year lifespan of a given line of PLDs.
- 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. The manufacturers can add new features or upgrade products that already are in the field. To do this, they simply upload a new programming file to the PLD, via the Internet, creating new hardware logic in the system.

Commercial Off-the-Shelf Components (COTS)

- 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.
- Advantages:
  - They are readily available in the market
  - Cheap
  - Developer can cut down his development time to a great extent
  - Reduces the time to market

Commercial Off-the-Shelf Components (COTS) (continued)

- 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.
- E.g.: The TCP/IP plug-in module available from various manufactures like 'WIZnet', 'Freescale', 'Dynalog', etc. are very good examples of COTS product.

Fig: An example of a COTS product for TCP/IP plug-in from WIZnet (WIZnet NM7010A Plug in Module)

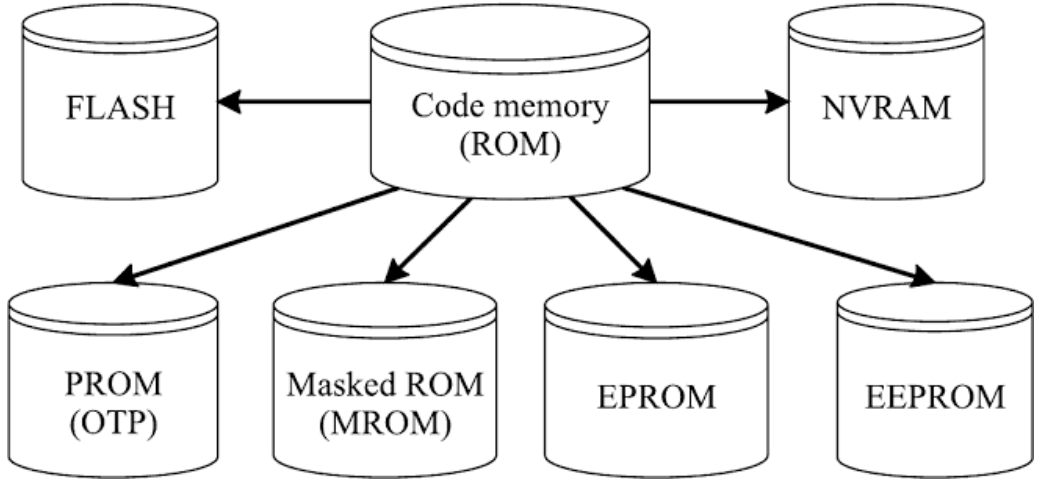
Memory

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.
- The code memory retains its contents even after the power is turned off. It is generally known as non-volatile storage memory.
- It can be classified into different types as shown:



Masked ROM (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 metallisation process at the time of production itself, according to the data provided by the end user.
- Advantage – low cost for high volume production.
- Limitation – 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.

Masked ROM (MROM) (continued)

- Different mechanisms are used for the masking process of the ROM, like
  1. Creation of an enhancement or depletion mode transistor through channel implant.
  2. By creating the memory cell either using a standard transistor or a high threshold transistor.
     - In the high threshold mode, the supply voltage required to turn ON the transistor is above the normal ROM IC operating voltage.
     - This ensures that the transistor is always off and the memory cell stores always logic 0.

Programmable Read Only Memory (PROM) / (OTP)

- One Time Programmable Memory (OTP) or 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 a 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 prototyped versions are proven and the code is finalised.
  - It is a low cost solution for commercial production.
- OTPs cannot be reprogrammed.

Erasable Programmable Read Only Memory (EPROM)

- Erasable Programmable Read Only Memory (EPROM) gives the flexibility to re-program 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.
- 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.
  - It is a tedious and time-consuming process.

Electrically Erasable Programmable Read only Memory (EEPROM)

- 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 (a few kilobytes) when compared with the standard ROM.

FLASH

- FLASH memory is a variation of EEPROM technology – It combines the re-programmability of EEPROM and the high capacity of standard ROMs.
- FLASH is the latest ROM technology.
  - Most popular ROM technology used in today's embedded designs.
- FLASH memory is organised as sectors (blocks) or pages.
- FLASH memory stores information in an array of floating gate MOSFET transistors.
- The erasing of memory can be done at sector level or page level without affecting the other sectors or pages.
- Each sector/page should be erased before re-programming.
- The typical erasable capacity of FLASH is 1000 cycles.
- E.g.: W27C512 from WINBOND is an example of 64KB FLASH memory.

Non-Volatile RAM (NVRAM)

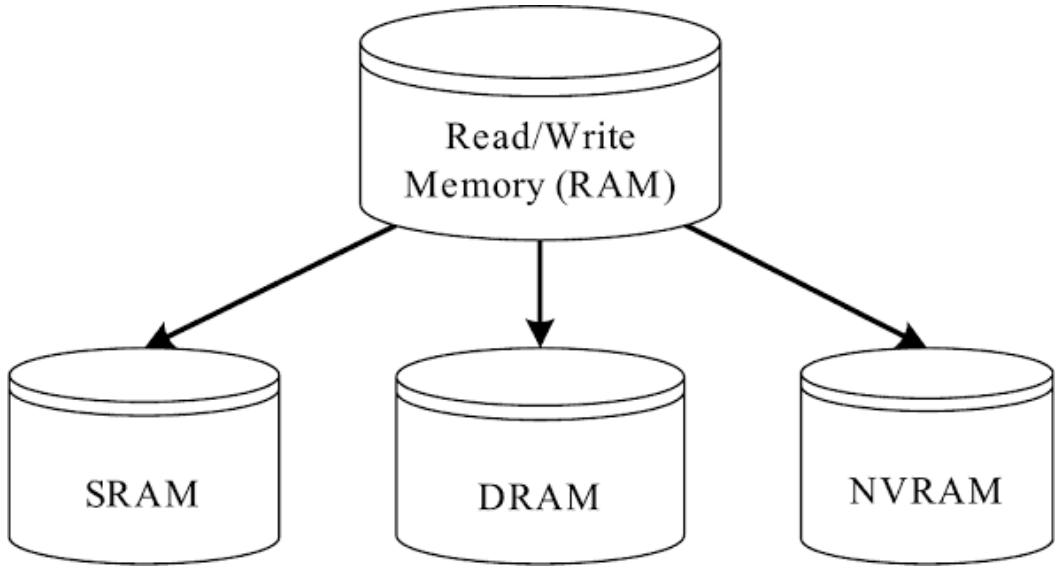
- Non-volatile RAM is a random access memory with battery backup.
- It contains static RAM based memory and a minute battery for providing supply to the memory in the absence of external power supply.
- The memory and battery are packed together in a single package.
- The life span of NVRAM is expected to be around 10 years.
- E.g.: DS1644 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 – when the power is turned off, all the contents are destroyed.
- RAM is a direct access memory – 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.

Read-Write Memory/Random Access Memory (RAM) (continued)

- RAM generally falls into three categories: Static RAM (SRAM), Dynamic RAM (DRAM) and Non-Volatile RAM (NVRAM).



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.
  - Fast due to its resistive networking and switching capabilities.
- In typical implementation, an SRAM cell (bit) is realised using six transistors (or 6 MOSFETs).
  - Four of the transistors are used for building the latch (flip-flop) part of the memory cell and two for controlling the access.

Static RAM (SRAM) (continued)

- In its simplest representation an SRAM cell can be visualised as shown in the figure below:

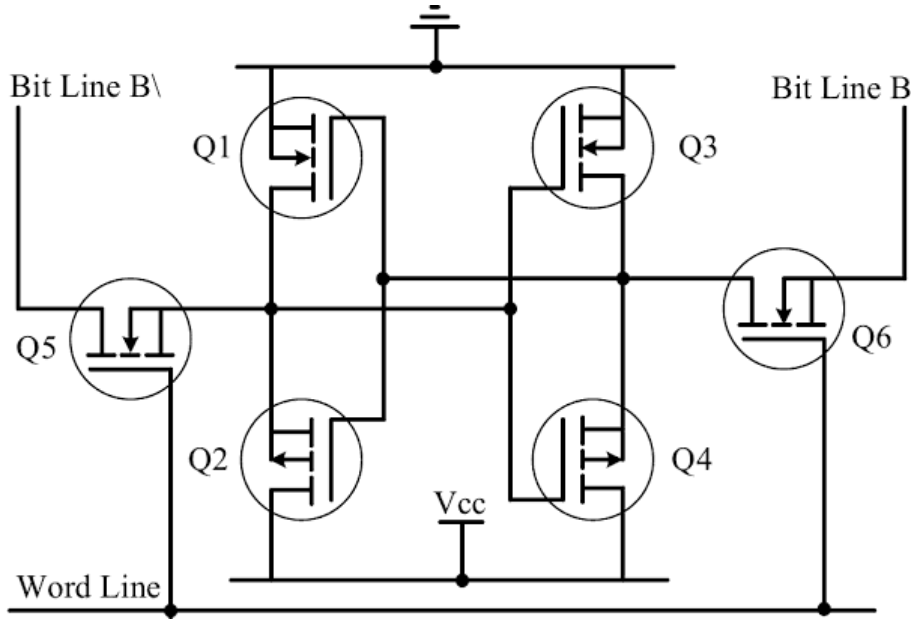


Fig: SRAM cell implementation

Static RAM (SRAM) (continued)

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

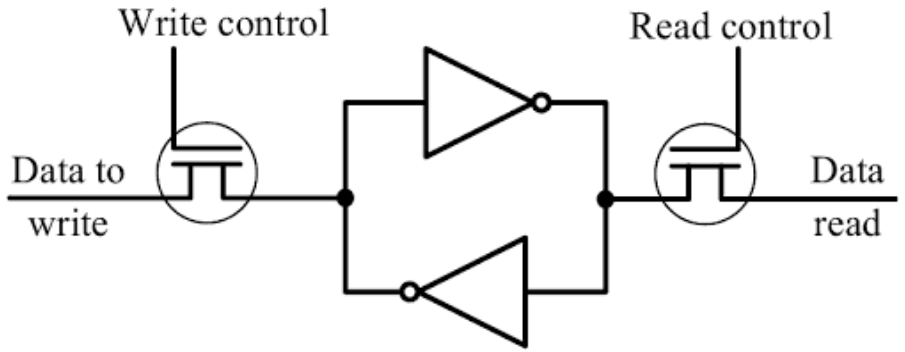


Fig: Visualisation of SRAM cell

Static RAM (SRAM) (continued)

- The access to the memory cell is controlled by 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.

Dynamic RAM (DRAM)

- Dynamic RAM stores data in the form of charge.
- They are made up of MOS transistor gates.
- Advantages – high density and low cost compared to SRAM.
- Disadvantage – 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.

Dynamic RAM (DRAM) (continued)

- Figure below 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.

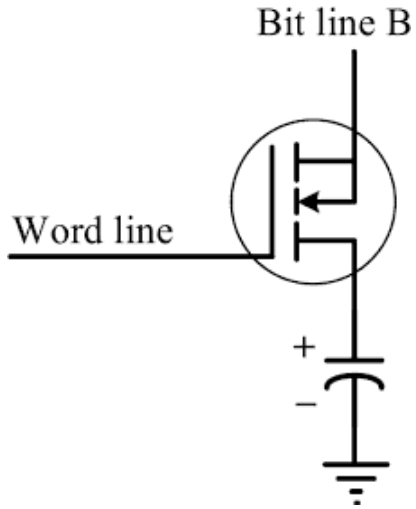


Fig: DRAM cell implementation

SRAM vs DRAM

- Table given below summarises the relative merits and demerits of SRAM and DRAM technology.

SRAM Cell	DRAM Cell
Made up of 6 CMOS transistors (MOSFET)	Made up of a MOSFET and a capacitor
Doesn't require refreshing	Requires refreshing
Low capacity (Less dense)	High capacity (Highly dense)
More expensive	Less expensive
Fast in operation. Typical access time is 10ns	Slow in operation due to refresh requirements. Typical access time is 60ns. Write operation is faster than read operation.

Memory According to the Type of Interface

- The interface (connection) of memory with the processor/controller can be of various types.
  - Parallel interface
  - Serial interface like I2C
  - SPI (Serial peripheral interface)
  - 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 ROM is very slow (120 to 200 ns) compared to the execution from a RAM(40 to 70 ns).
  - RAM access is about three times as fast as ROM access.
- Memory Shadowing 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.
  - 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 according to it during system boot up and it is time consuming.

Memory Shadowing (continued)

- In memory shadowing, a RAM is included behind the logical layer of BIOS at its same address as a shadow to the BIOS.
- The first step that happens during the boot up is copying the BIOS to the shadowed RAM and write protecting the RAM then disabling the BIOS reading.
  - 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 the applications designed to run on top of it
  - Data memory for holding variables and temporary data during task execution
  - 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.
  - Lot of factors need to be considered when selecting the type and size of memory for embedded system.

Memory Selection for Embedded Systems (continued)

- 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.
- 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 of FLASH memory and a few bytes of EEPROM (if required) for designing the system. Hence there is no need for external memory at all.
  - A PIC microcontroller device which satisfies the I/O and memory requirements can be used in this case.

Memory Selection for Embedded Systems (continued)

- 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.
- 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, extra RAM and ROM are needed for running user applications.

Memory Selection for Embedded Systems (continued)

- There are two parameters for representing a memory:
  1. Size of the memory chip – Memory density expressed in terms of number of memory bytes per chip.
     - Memory chips come in standard sizes like 512 bytes, 1024 bytes (1 kilobyte), 2048 bytes (2 kilobytes), 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1024KB(1 megabytes), etc.
     - While selecting a memory size, the address range supported by the processor must also be considered. For example, for a processor/controller with 16 bit address bus, the maximum number of memory locations that can be addressed is 2^{16} = 65536 bytes = 64KB
     - 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.
  2. Word size of the memory – The number of memory bits that can be read/written together at a time.
     - Word size can be 4, 8, 12, 16, 24, 32, etc.
     - The word size supported by the memory chip must match with the data bus width of the processor/controller.

Memory Selection for Embedded Systems (continued)

- FLASH memory is the popular choice for ROM 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 – NAND FLASH and NOR FLASH.
- NAND FLASH is a high-density low cost non-volatile storage memory, while NOR FLASH is less dense and slightly expensive.
- 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.
- It is a good practice to use a combination of NOR and NAND memory for storage memory requirements, where NAND can be used for storing the program code and/or data like the data captured in a camera device.
  - NAND FLASH doesn't support XIP and if NAND FLASH is used for storing program code, a DRAM can be used for copying and executing the program code.
  - NOR FLASH supports XIP and it can be used as the memory for bootloader or for even storing the complete program code.

Memory Selection for Embedded Systems (continued)

- 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 EEPROM 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.

Memory Selection for Embedded Systems (continued)

- For embedded systems with low power requirements like portable devices, choose low power memory devices.
- Certain embedded devices may be targeted for operating at extreme environmental conditions like high temperature, high humid area, etc.
  - Select an industrial grade memory chip in place of the commercial grade chip for such devices.

Sensors and Actuators

Sensors and 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.
- 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.

Sensors and Actuators (continued)

- A sensor is a transducer device that converts energy from one form to another for any measurement or control purpose.
  - E.g.: Temperature sensor, magnetic hall effect sensor, humidity sensor, etc.
- An actuator is a form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion).
  - Actuator acts as an output device.
  - E.g.: Stepper motor

The I/O Subsystem

- The I/O subsystem of the embedded system facilitates the interaction of the embedded system with the external world.
- The interaction happens through the sensors and actuators connected to the input and output ports respectively of the embedded system.
- The sensors may not be directly interfaced to the input ports, instead they may be interfaced through signal conditioning and translating systems like ADC, optocouplers, etc.

Light Emitting Diode (LED)

- Light Emitting Diode (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.
  - E.g.: 'Device ON', 'Battery low' or 'Charging of battery' conditions
- 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 is connected to +ve terminal of the supply voltage and cathode to the -ve terminal of supply voltage.
- The current flowing through the LED must be limited to a value below the maximum current that it can conduct.
  - A resister is used in series to limit the current through the LED.
- The ideal LED interfacing circuit is shown in the figure.

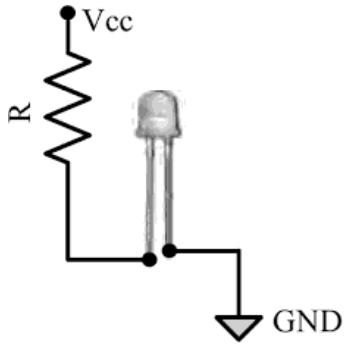


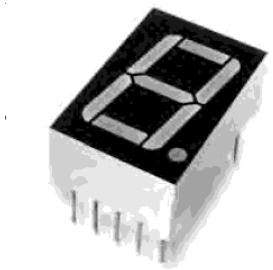
Fig: LED interfacing

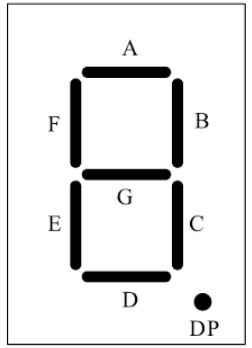
Light Emitting Diode (LED) (continued)

- 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.
    - 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 the processor/controller and the anode to the supply voltage through a current limiting resistor.
    - The LED is turned on when the port pin is at logic Low (Logic '0’).
    - Here the port pin 'sinks' current.

7-Segment LED Display

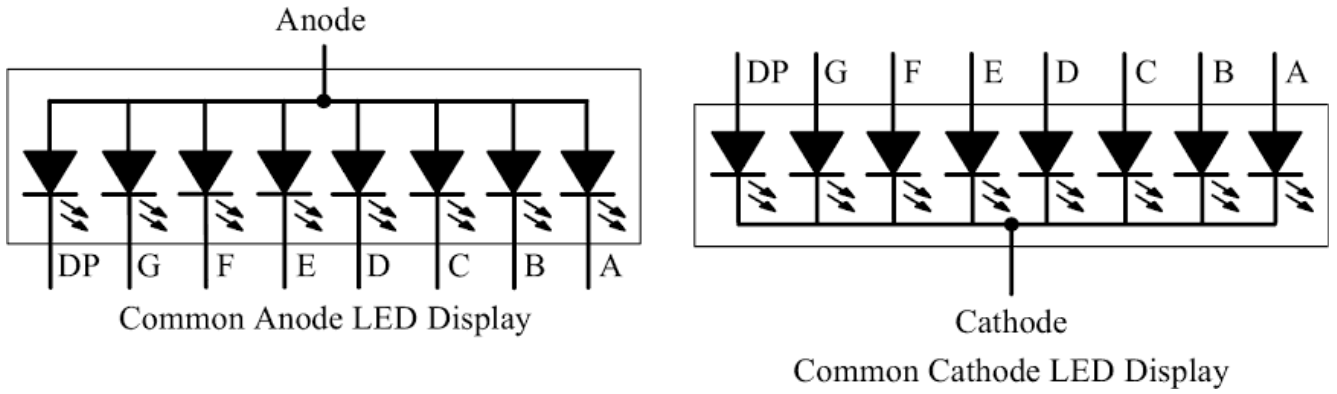
- The 7-segment LED display is an output device for displaying alpha numeric characters.
- It contains 7 LED segments arranged in a special form used for displaying alpha numeric characters and 1 LED used for representing 'decimal point' in decimal number display.
- The LED segments are named A to G and the decimal point LED segment is named as DP.
- The LED segments A to G and DP should be lit accordingly to display numbers and characters.





7-Segment LED Display (continued)

- The 7-segment LED displays are available in two different configurations, namely; Common Anode and Common Cathode.
- In the common anode configuration, the anodes of the 8 segments are connected commonly whereas in the common cathode configuration, the cathodes of 8 LED segments are connected commonly.
- Figure illustrates the Common Anode and Cathode configurations.



7-Segment LED Display (continued)

- Based on the configuration of the 7-segment LED unit, the LED segment's anode or cathode is connected to the port of the processor/controller in the order 'A' segment to the least significant port pin and DP segment to the most significant port pin.
- The current flow through each of the LED segments should be limited to the maximum value supported by the LED display unit.
  - The typical value is 20mA.
  - The current can be limited by connecting a current limiting resistor to the anode or cathode of each segment.
- 7-segment LED display is used in low cost embedded applications like Public telephone call monitoring devices, point of sale terminals, etc.

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.
- Figure illustrates the functioning of an optocoupler device.

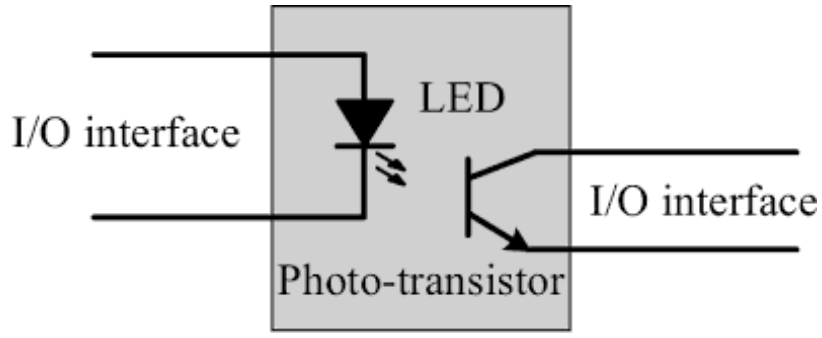


Fig: An optocoupler device

Optocoupler (continued)

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

Optocoupler (continued)

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

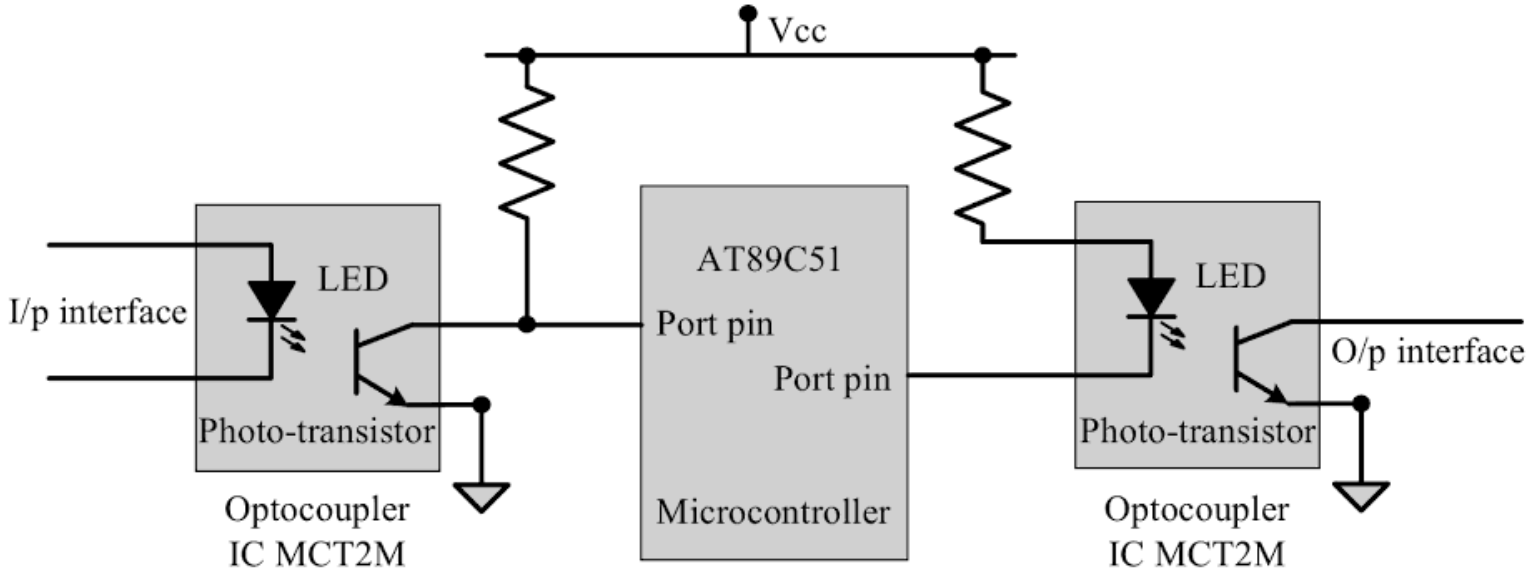


Fig: Optocoupler in Input and Output circuit

Relay

- Relay is an electro-mechanical device.
- In embedded application, the Relay unit acts as dynamic path selector 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.

Relay (continued)

- Relays are available in different configurations.
- Figure given below illustrates the widely used relay configurations for embedded applications.

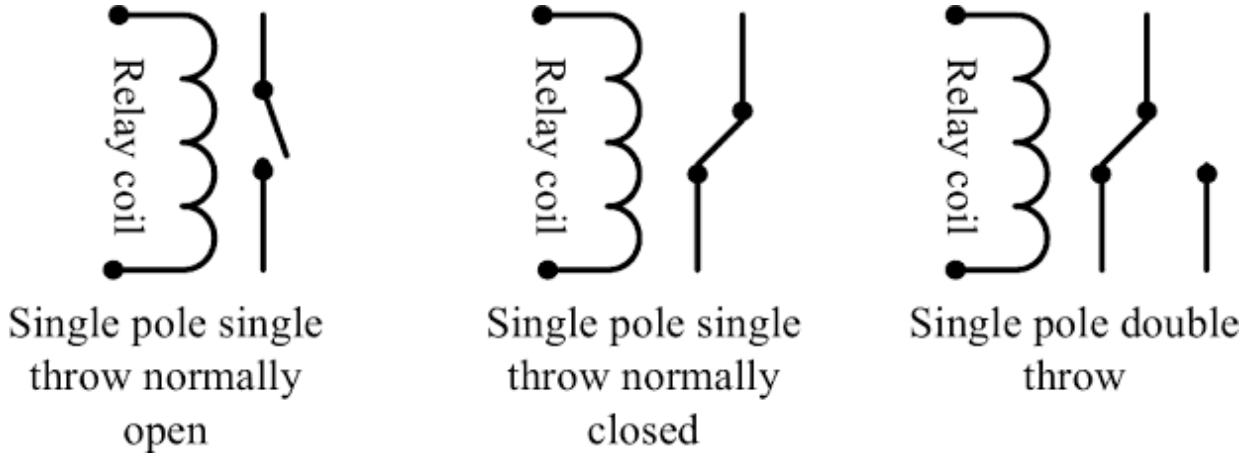


Fig: Relay configurations

Relay (continued)

- The Single Pole Single Throw configuration has only one path for information flow.
- The path is either open or closed in normal condition.
  - For Normally Open Single Pole Single Throw relay, the circuit is normally open and it becomes closed when the relay is energised.
  - For Normally Closed Single Pole Single Throw relay, the circuit is normally closed and it becomes open when the relay is energised.
- For Single Pole Double Throw configuration, there are two paths for information flow and they are selected by energising or de-energising the relay.

Relay (continued)

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

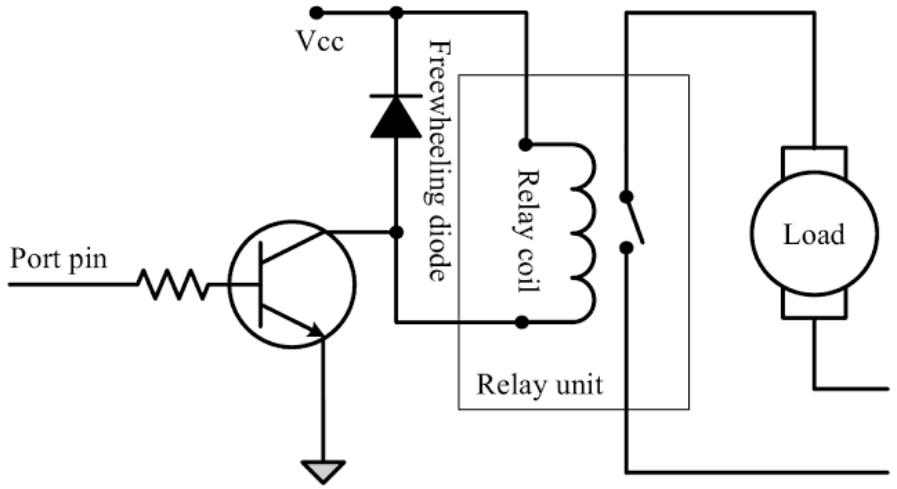


Fig: Transistor based Relay driving circuit

- Most of the industrial relays are bulky and require high voltage to operate.
- Special relays called 'Reed' relays are available for embedded application requiring switching of low voltage DC signals.

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 support 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

- It 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 Switch (continued)

- Push button is used for generating a momentary pulse.
- In embedded applications, 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.
- Figure illustrates how the push button can be used for generating 'LOW' and 'HIGH' pulses.

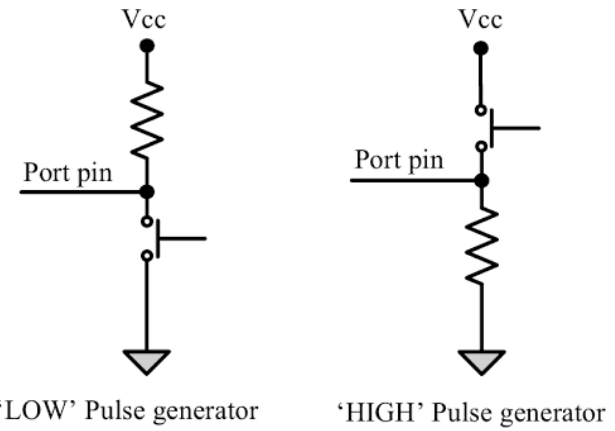


Fig: Push button switch configurations

Communication Interface

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:
  - Onboard Communication Interface (Device/board level communication interface)
    - E.g.: Serial interfaces like I2C, SPI, UART, 1-Wire, etc and parallel bus interface.
  - External Communication Interface (Product level communication interface)
    - E.g.: Wireless interfaces like Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS, etc. and wired interfaces like RS-232C/RS-422/RS-485, USB, Ethernet IEEE 1394 port, Parallel port, CF-II interface, SDIO, PCMCIA, etc.

Onboard Communication Interfaces

- An embedded system is a combination of different types of components (chips/devices) arranged on a printed circuit board (PCB).
- Onboard Communication Interface refers to the different communication channels/buses for interconnecting the various integrated circuits and other peripherals within the embedded system.
- E.g.: Serial interfaces like I2C, SPI, UART, 1-Wire, etc and parallel bus interface

Inter Integrated Circuit (I2C) Bus

- The Inter Integrated Circuit Bus (I2C or I^2C Pronounced 'I square C') is a synchronous bi-directional half duplex two wire serial interface bus.
  - (Half duplex - one-directional communication at a given point of time)
- 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:
  - Serial Clock (SCL line) – responsible for generating synchronisation clock pulses
  - Serial Data (SDA line) – responsible for transmitting the serial data across devices

Inter Integrated Circuit (I2C) Bus (continued)

- 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' or 'Slave'.
  - The 'Master' device is responsible for controlling the communication by initiating/terminating data transfer, sending data and generating necessary synchronisation 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 synchronisation clock signal is generated by the 'Master' device only.
- I2C supports multi masters on the same bus.

Inter Integrated Circuit (I2C) Bus (continued)

- The following bus interface diagram illustrates the connection of master and slave devices on the I2C bus.

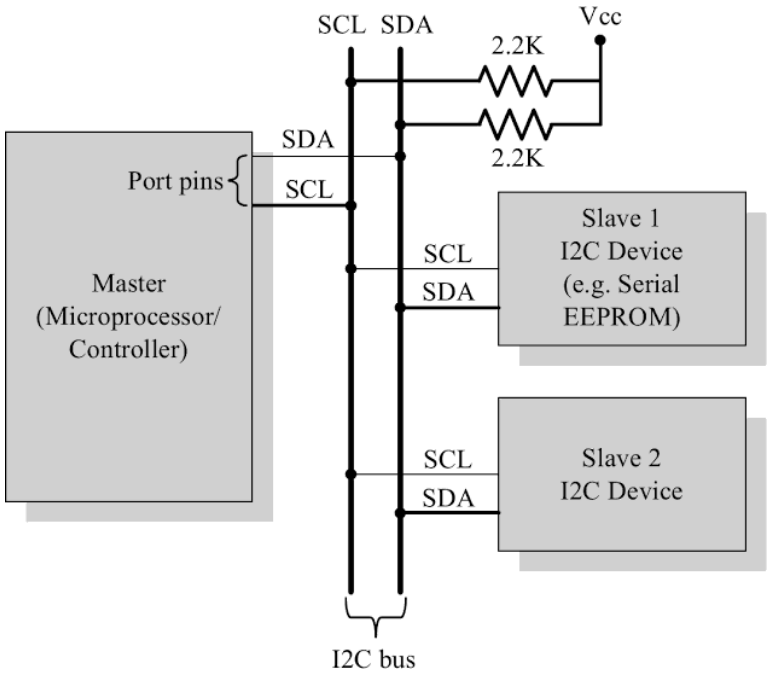


Fig: I2C Bus Interfacing

Inter Integrated Circuit (I2C) Bus (continued)

- The I2C bus interface is built around an input buffer and an open drain or collector transistor.
- When the bus is in the idle state, the open drain/collector transistor will be in the floating state and the output lines (SDA and SCL) switch to the 'High Impedance' state.
- For proper operation of the bus, both the bus lines should be pulled to the supply voltage (+5 V for TTL family and +3.3V for CMOS family devices) using pull-up resistors.
  - The typical value of resistors used in pull-up is 2.2K.
  - With pull-up resistors, the output lines of the bus in the idle state will be 'HIGH'
- The address of a I2C device is assigned by hardwiring the address lines of the device to the desired logic level.
  - Done at the time of designing the embedded hardware.

Inter Integrated Circuit (I2C) Bus (continued)

- 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 synchronising 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

Inter Integrated Circuit (I2C) Bus (continued)

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)

Inter Integrated Circuit (I2C) Bus (continued)

- I2C bus supports three different data rates:
  - Standard mode (Data rate up to 100kbits/sec (100 kbps))
  - Fast mode (Data rate up to 400kbits/sec (400 kbps))
  - High speed mode (Data rate up to 3.4Mbits/sec (3.4 Mbps))

Serial Peripheral Interface (SPI) Bus

- The Serial Peripheral Interface Bus (SPI) is a synchronous 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.
  - There can be more than one masters, but only one master device can be active at any given point of time.
- 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 (SCLK) – Signal line carrying the clock signals
  - Slave Select (SS) – Signal line for slave device select. It is an active low signal

Serial Peripheral Interface (SPI) Bus (continued)

- The bus interface diagram shown in the figure illustrates the connection of master and slave devices on the SPI bus.

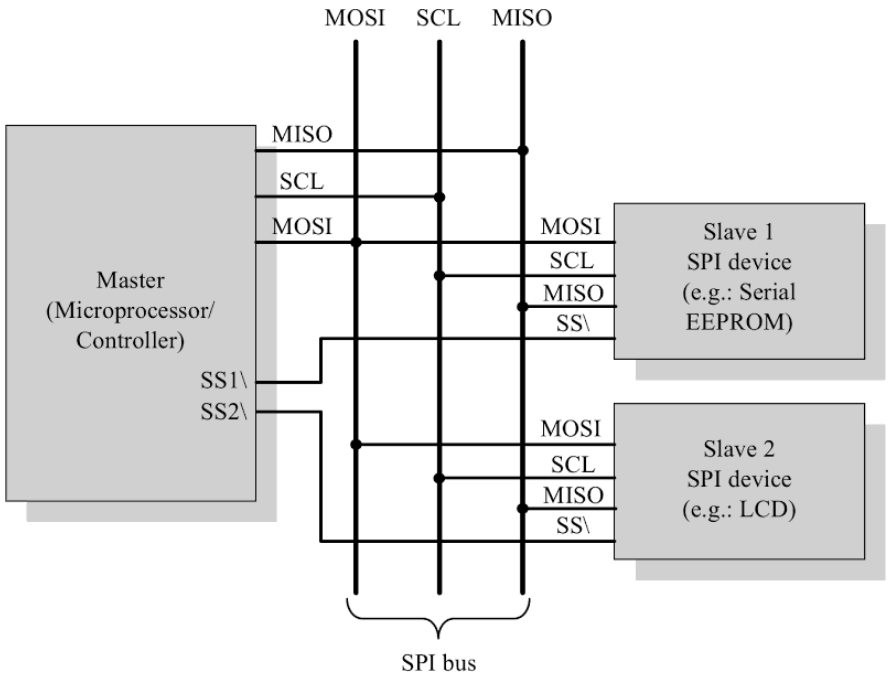


Fig: SPI Bus Interfacing

Serial Peripheral Interface (SPI) Bus (continued)

- 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.
- The serial data transmission through SPI bus is fully configurable.
  - SPI devices contain a certain set of registers for holding these configurations.
  - The control register holds the various configuration parameters like master/slave selection for the device, baud rate selection for communication, clock signal control, etc.
  - The status register holds the status of various conditions for transmission and reception.

Serial Peripheral Interface (SPI) Bus (continued)

- 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.
- It doesn't require a clock signal to synchronise the transmitting end and receiving end for transmission.
- Instead it relies upon the pre-defined agreement between the transmitting device and receiving device.

Universal Asynchronous Receiver Transmitter (UART) (continued)

- The serial communication settings (Baudrate, 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.

Universal Asynchronous Receiver Transmitter (UART) (continued)

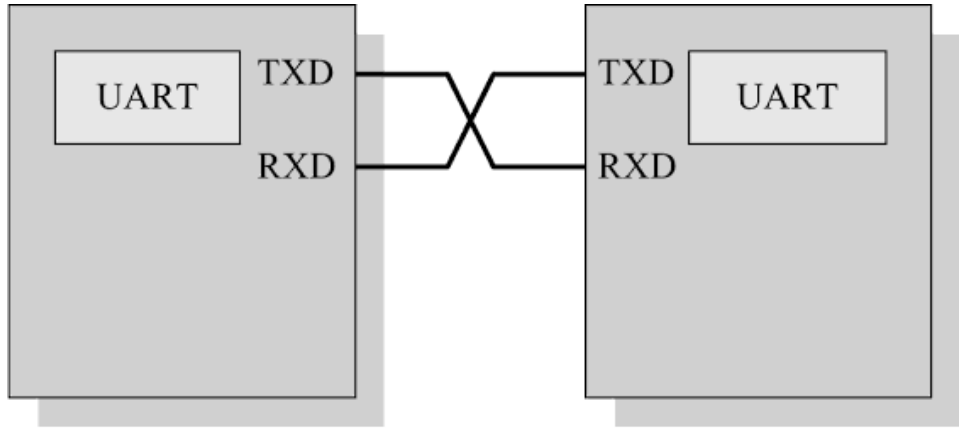
- The 'start' bit informs the receiver that a data byte is about to arrive.
- The receiver device starts polling its 'receive line' as per the baud rate settings.
  - If the baud rate is 'x' bits per second, the time slot available for one bit is 1/x seconds.
- The receiver unit polls the receiver line 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
  - In the case of 8 bits/byte, the byte is formed with the received 8 bits with the first received bit as the LSB and last received data bit as MSB.

Universal Asynchronous Receiver Transmitter (UART) (continued)

- For proper communication, the 'Transmit line' of the sending device should be connected to the 'Receive line' of the receiving device.
- 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.
- Nowadays most of the microprocessors/controllers are available with integrated UART functionality and they provide built-in instruction support for serial data transmission and reception.

Universal Asynchronous Receiver Transmitter (UART) (continued)

- Figure illustrates the UART interfacing.



TXD: Transmitter line
RXD: Receiver line

Fig: UART Interfacing

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 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.

1-Wire Interface (continued)

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

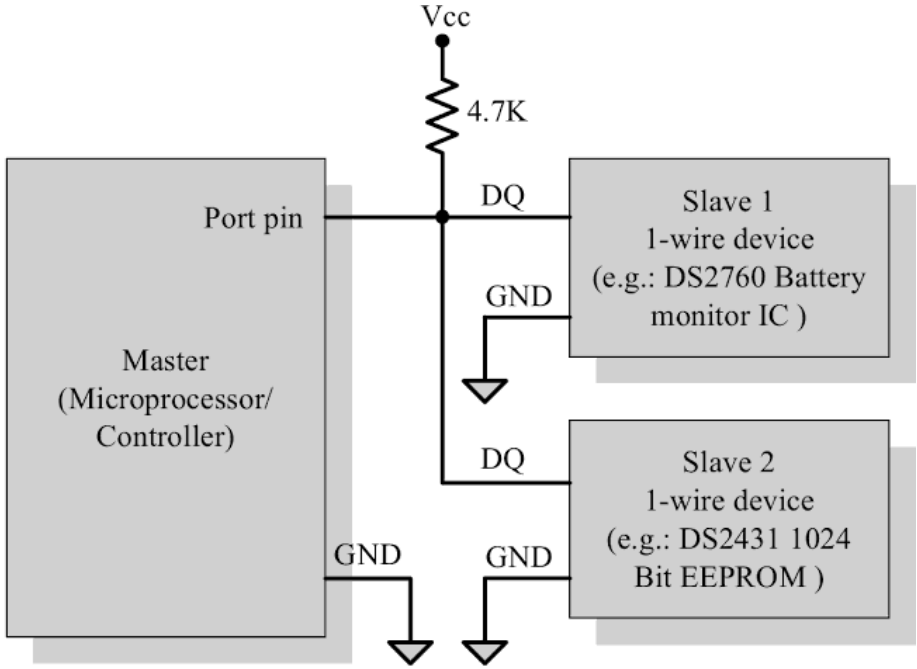


Fig: 1-Wire Interface Bus

1-Wire Interface (continued)

- Every 1-wire device contains a globally unique 64bit 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.

1-Wire Interface (continued)

- 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.

1-Wire Interface (continued)

- All communication over the 1 -wire bus is master initiated.
- The communication over the 1-wire bus is divided into timeslots of 60 microseconds.
- The 'Reset' pulse occupies 8 time slots. For starting a communication, the master asserts the reset pulse by pulling the 1-wire bus 'LOW' for at least 8 time slots (480 µs).
- If a 'slave' device is present on the bus and is ready for communication it should respond to the master with a 'Presence' pulse, within 60 µs of the release of the 'Reset' pulse by the master.
- The slave device(s) responds with a 'Presence' pulse by pulling the 1-wire bus 'LOW' for a minimum of 1 time slot (60 µs).

1-Wire Interface (continued)

- For writing a bit value of 1 on the 1-wire bus, the bus master pulls the bus for 1 to 15 µs and then releases the bus for the rest of the time slot.
  - A bit value of ‘0' is written on the bus by master pulling the bus for a minimum of 1 time slot (60 µs) and a maximum of 2 time slots (120 µs).
- To Read a bit from the slave device, the master pulls the bus 'LOW' for 1 to 15 µs.
  - If the slave wants to send a bit value ‘1' in response to the read request from the master, it simply releases the bus for the rest of the time slot.
  - If the slave wants to send a bit value '0', it pulls the bus 'LOW' for the rest of the time slot.

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.

Parallel Interface (continued)

- 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).

Parallel Interface (continued)

- The bus interface diagram shown in the figure illustrates the interfacing of devices through parallel interface.

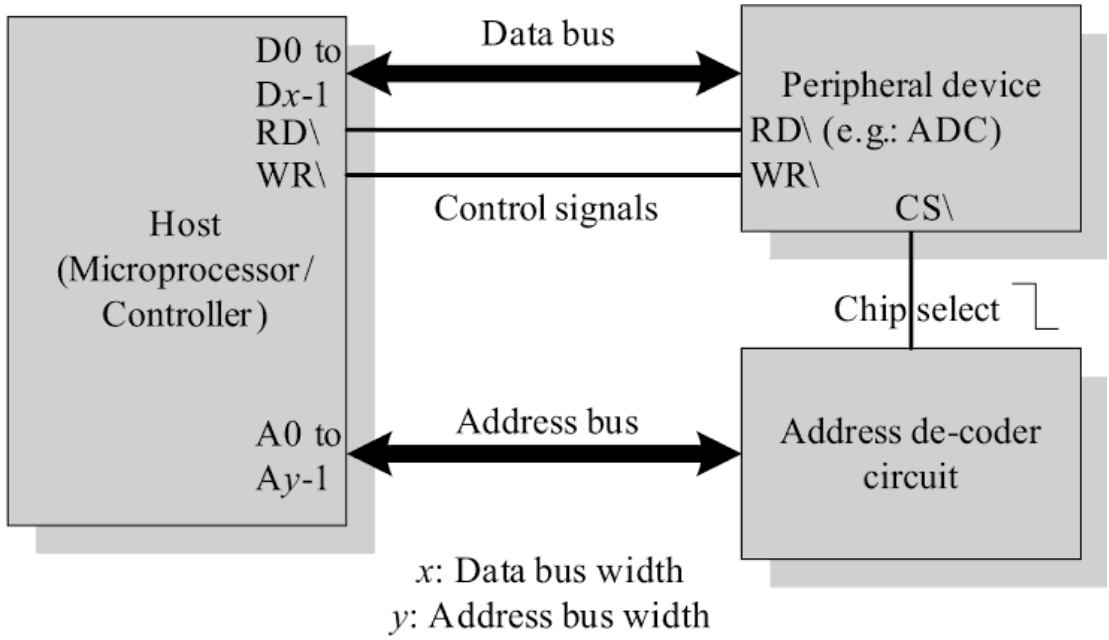


Fig: Parallel Interface Bus

Parallel Interface (continued)

- 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.

External Communication Interfaces

- External Communication Interface refers to the different communication channels/buses used by the embedded system to communicate with the external world.
- E.g.: RS-232 C & RS-485, Universal Serial Bus (USB), IEEE 1394 (Firewire), Infrared (IR), Bluetooth (BT), Wi-Fi, ZigBee, GPRS, etc.

RS-232 C & RS-485

- RS-232 C (Recommended Standard number 232, revision C) is a legacy, full duplex, wired, asynchronous serial communication interface.
- The RS-232 interface was 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 -25 V.
  - In EIA standard, logic ‘0’ is known as 'Space' and logic ‘1’ as ‘Mark’.

RS-232 C & RS-485 (continued)

- 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
  - DB-25: 25-Pin connector.

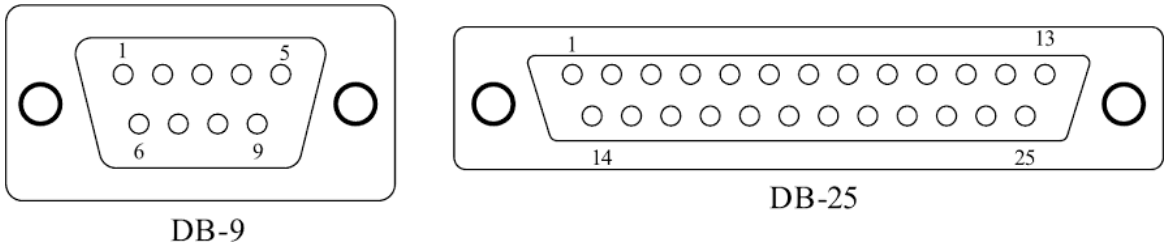


Fig: DB-9 and DB-25 RS-232 Connector Interface

RS-232 C & RS-485 (continued)

- The pin details for the two connectors are explained in the following table:

Pin Name	Pin no: (For DB-9 Connector)	Pin no: (For DB-25 Connector)	Description
TXD	3	2	Transmit Pin for Transmitting Serial Data
RXD	2	3	Receive Pin for Receiving Serial Data
RTS	7	4	Request to send.
CTS	8	5	Clear To Send
DSR	6	6	Data Set Ready
GND	5	7	Signal Ground
DCD	1	8	Data Carrier Detect
DTR	4	20	Data Terminal Ready
RI	9	22	Ring Indicator
FG		1	Frame Ground
SDCD		12	Secondary DCD
SCTS		13	Secondary CTS
STXD		14	Secondary TXD
TC		15	Transmission Signal Element Timing
SRXD		16	Secondary RXD
RC		17	Receiver Signal Element Timing
SRTS		19	Secondary RTS
SQ		21	Signal Quality detector
NC		9	No Connection
NC		10	No Connection
NC		11	No Connection
NC		18	No Connection
NC		23	No Connection
NC		24	No Connection
NC		25	No Connection

Pin Name	Pin no: (For DB-9 Connector)	Pin no: (For DB-25 Connector)	Description
TXD	3	2	Transmit Pin for Transmitting Serial Data
RXD	2	3	Receive Pin for Receiving Serial Data
RTS	7	4	Request to send.
CTS	8	5	Clear To Send
DSR	6	6	Data Set Ready
GND	5	7	Signal Ground
DCD	1	8	Data Carrier Detect
DTR	4	20	Data Terminal Ready
RI	9	22	Ring Indicator
FG		1	Frame Ground
SDCD		12	Secondary DCD
SCTS		13	Secondary CTS
STXD		14	Secondary TXD
TC		15	Transmission Signal Element Timing
SRXD		16	Secondary RXD
RC		17	Receiver Signal Element Timing
SRTS		19	Secondary RTS
SQ		21	Signal Quality detector
NC		9	No Connection
NC		10	No Connection
NC		11	No Connection
NC		18	No Connection
NC		23	No Connection
NC		24	No Connection
NC		25	No Connection

RS-232 C & RS-485 (continued)

- RS-232 is a point-to-point communication interface and the devices involved in RS-232 communication are called 'Data Terminal Equipment (DTE)' and 'Data Communication Equipment (DCE)’.
- If no data flow control is required, only TXD and RXD signal lines and ground line (GND) are required for data transmission and reception.
  - The RXD pin of DCE should be connected to the TXD pin of DTE and vice versa for proper data transmission.
- If hardware data flow control is required for serial transmission, various control signal lines of the RS-232 connection are used appropriately.

RS-232 C & RS-485 (continued)

- The Request To Send (RTS) and Clear To Send (CTS) signals co-ordinate the communication between DTE and DCE.
  - Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line.
- The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data.
- The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link.
  - DTR should be in the activated state before the activation of DSR.
- The Data Carrier Detect (DCD) control signal is used by the DCE to indicate the DTE that a good signal is being received.
- Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line.

RS-232 C & RS-485 (continued)

- As per the EIA standard RS-232 C supports baudrates up to 20Kbps (Upper limit 19.2 Kbps)
  - The commonly used baudrates by devices are 300bps, 1200bps, 2400bps, 9600bps, 11.52Kbps and 19.2Kbps.
  - 9600 is the popular baudrate setting used for PC communication.
- The maximum operating distance supported by RS-232 is 50 feet at the highest supported baudrate.
- Embedded devices contain a UART for serial communication and they generate signal levels conforming to TTL/CMOS logic.
  - A level translator IC like MAX 232 from Maxim Dallas semiconductor is used for converting the signal lines from the UART to RS-232 signal lines for communication.

RS-232 C & RS-485 (continued)

- 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 100Kbps 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 (USB) is a wired high speed serial bus for data communication.
- The first version of USB (USB 1.0) was released in 1995.
- 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 host can support connections up to 127, including slave peripheral devices and other USB hosts.

Universal Serial Bus (USB) (continued)

- Figure illustrates the star topology for USB device connection.

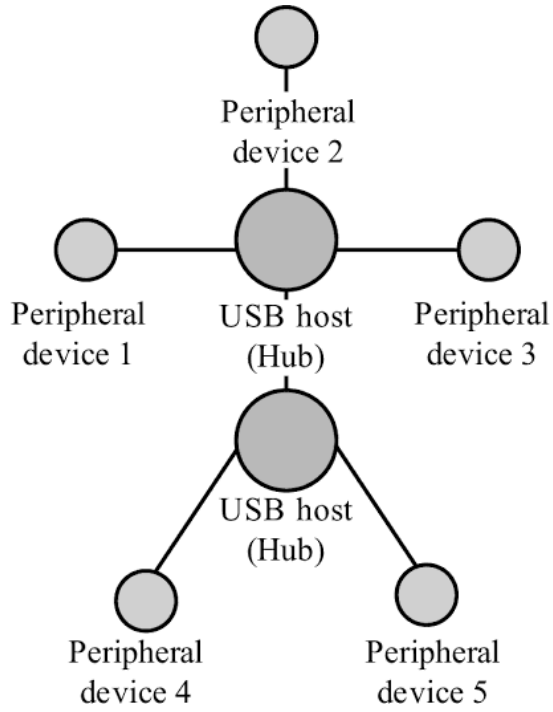


Fig: USB Device Connection topology

Universal Serial Bus (USB) (continued)

- 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:
  - Open Host Control Interface (OHCI)
  - Universal Host Control Interface (UHCI)

Universal Serial Bus (USB) (continued)

- The physical connection between a USB peripheral device and master device is established with a USB cable.
- The USB cable supports communication distance of up to 5 metres.
- The USB standard uses two different types of connector at the ends of the USB cable 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.

Universal Serial Bus (USB) (continued)

- 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

Universal Serial Bus (USB) (continued)

Type A Connector

Type B Connector

Type C Connector

Universal Serial Bus (USB) (continued)

- 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 mA at 5 V.
  - It is sufficient to operate low power devices.
- Mini and Micro USB connectors are available for small form factor devices like portable media players.
- Each USB device contains a Product ID (PID) and a Vendor ID (VID).
  - Embedded into the USB chip by the USB device manufacturer.
  - The VID for a device is supplied by the USB standards forum.
  - PID and VID are essential for loading the drivers corresponding to a USB device for communication.

Universal Serial Bus (USB) (continued)

- USB supports four different types of data transfers:
- Control transfer : Used by USB system software to query, configure and issue commands to the USB device.
- Bulk transfer : Used for sending a block of data to a device.
  - Supports error checking and correction.
  - Transferring data to a printer is an example for bulk transfer.
- Isochronous data transfer : Used for real-time data communication.
  - Data is transmitted as streams in real-time.
  - Doesn't support error checking and re-transmission 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 : 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.

Universal Serial Bus (USB) (continued)

- USB.ORG is the standards body for defining and controlling the standards for USB communication.
- Presently USB supports different data rates:
  - Low-Speed (LS) - 1.5Mbps – USB 1.0
  - Full-Speed (FS) - 12Mbps – USB 1.0
  - High-Speed (HS) - 480Mbps – USB 2.0
  - SuperSpeed (SS) - 5Gbps – USB 3.0
  - SuperSpeed+ (SS+) - 10Gbps – USB 3.1, 20 Gbps – USB 3.2

IEEE 1394 (Firewire)

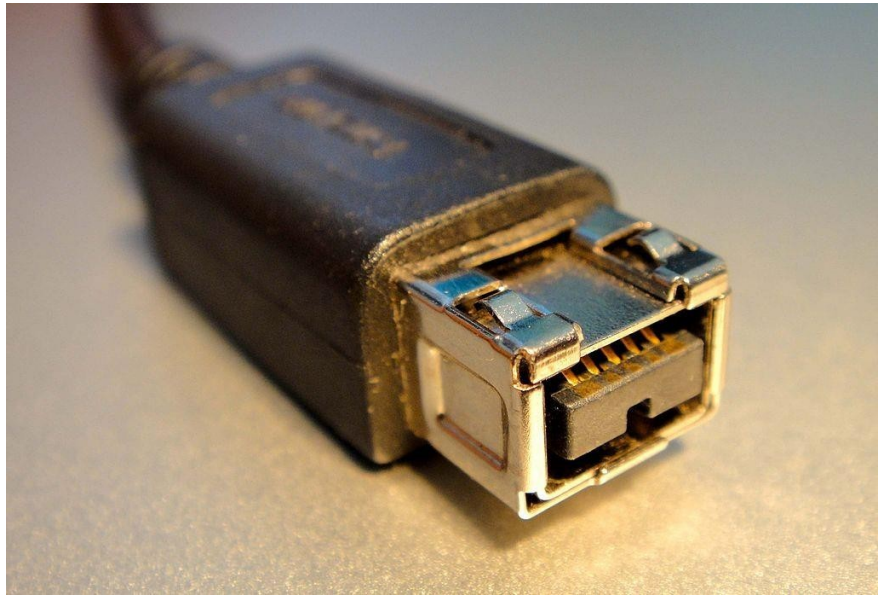
- IEEE 1394 is a wired, isochronous high speed serial communication bus.
- It is also known as High Performance Serial Bus (HPSB).
- The research on 1394 was started by Apple Inc. in 1985 and the standard for this was coined by IEEE.
- The implementation of 1394 is available from various players with different names:
  - Firewire is the implementation from Apple Inc
  - i.LINK is the implementation from Sony Corporation
  - Lynx is the implementation from Texas Instruments

IEEE 1394 (Firewire) (continued)

- 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.
- The 1394 standard supports a data rate of 400 to 3200 Mbits/second.
- The IEEE 1394 uses differential data transfer.
  - It increases the noise immunity.
- The interface cable supports 3 types of connectors, namely; 4-pin connector, 6-pin connector (alpha connector) and 9 pin connector (beta connector).
- The 6 and 9 pin connectors carry power also to support external devices.
  - It can supply unregulated power in the range of 24 to 30V.

IEEE 1394 (Firewire) (continued)

4-pin and 6-pin Connectors



9-pin Connector

IEEE 1394 (Firewire) (continued)

- The table given below illustrates the pin details for 4, 6 and 9 pin connectors.

Pin name	Pin no: (4 Pin Connector)	Pin no: (6 Pin Connector)	Pin no: (9 Pin Connector)	Description
Power		1	8	Unregulated DC supply. 24 to 30V
Signal Ground		2	6	Ground connection
TPB–	1	3	1	Differential Signal line for Signal line B
TPB+	2	4	2	Differential Signal line for Signal line B
TPA–	3	5	3	Differential Signal line for Signal line A
TPA+	4	6	4	Differential Signal line for Signal line A
TPA(S)			5	Shield for the differential signal line A. Normally grounded
TPB(S)			9	Shield for the differential signal line B. Normally grounded
NC			7	No connection

IEEE 1394 (Firewire) (continued)

- 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.
- 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 (IrDA) is a serial, half duplex, line of sight based wireless technology for data communication between devices.
- It is in use from the olden days of communication and you may be very familiar with it.
  - E.g.: The remote control of TV, VCD player, etc. works on Infrared.
- Infrared communication technique uses infrared waves of the electromagnetic spectrum for transmitting the data.
- It supports point-point and point-to-multipoint communication, provided all devices involved in the communication are within the line of sight.
- The typical communication range for IrDA lies in the range 10 cm to 1 m.
- The range can be increased by increasing the transmitting power of the IR device.

Infrared (IrDA) (continued)

- IR supports data rates ranging from 9600bits/second to 16Mbps.
- Depending on the speed of data transmission IR is classified into:
  - Serial IR (SIR) – supports data rates ranging from 9600bps to 115.2kbps.
  - Medium IR (MIR) – supports data rates of 0.576Mbps and 1.152Mbps.
  - Fast IR (FIR) – supports data rates up to 4Mbps.
  - Very Fast IR (VFIR) – supports high data rates up to 16Mbps.
  - Ultra Fast IR (UFIR) – targeted to support a data rate up to 100Mbps.

Infrared (IrDA) (continued)

- 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 remote 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.

Infrared (IrDA) (continued)

- Infrared Data Association (IrDA) is the regulatory body responsible for defining and licensing the specifications for IR data communication.
- IR communication 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 IR communication
  - Protocol part is responsible for defining the rules of communication.
- The physical link works on the wireless principle making use of Infrared for communication.
- The IrDA specifications include the standard for both physical link and protocol layer.
- The IrDA control protocol contains implementations for Physical Layer (PHY), Media Access Control (MAC) and Logical Link Control (LLC).

Infrared (IrDA) (continued)

- IrDA is a popular interface for file exchange and data transfer in low cost devices.
- IrDA was the prominent communication channel in mobile phones before Bluetooth's existence.

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.
- It supports a data rate of up to 1Mbps and a range of approximately 30 feet for data communication.

Bluetooth (BT) (continued)

- Bluetooth communication 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 protocol part is responsible for defining the rules of 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.

Bluetooth (BT) (continued)

- The rules governing the Bluetooth communication is implemented in the 'Bluetooth protocol stack'.
  - The Bluetooth communication IC holds the 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.

Bluetooth (BT) (continued)

- Bluetooth is the favourite choice for short range data communication in handheld embedded devices.
- Bluetooth technology is very popular among cell phone users as they are the easiest communication channel for transferring ringtones, music files, pictures, media files, etc. between neighbouring Bluetooth enabled phones.
- The Bluetooth standard specifies the minimum requirements that a Bluetooth device must support for a specific usage scenario.
- The Generic Access Profile (GAP) defines the requirements for detecting a Bluetooth device and establishing a connection with it.
  - All other specific usage profiles are based on GAP.
  - Serial Port Profile (SPP) for serial data communication, File Transfer Profile (FTP) for file transfer between devices, Human Interface Device (HID) for supporting human interface devices like keyboard and mouse are examples for Bluetooth profiles.
- The specifications for Bluetooth communication is defined and licensed by the standards body 'Bluetooth Special Interest Group (SIG)'.

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 it supports Internet Protocol (IP) based communication.
- It is essential to have device identities in a multipoint 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 (continued)

- 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 2.4 GHz or 5 GHz of radio spectrum and they co-exist with other ISM band devices like Bluetooth.

Wi-Fi (continued)

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

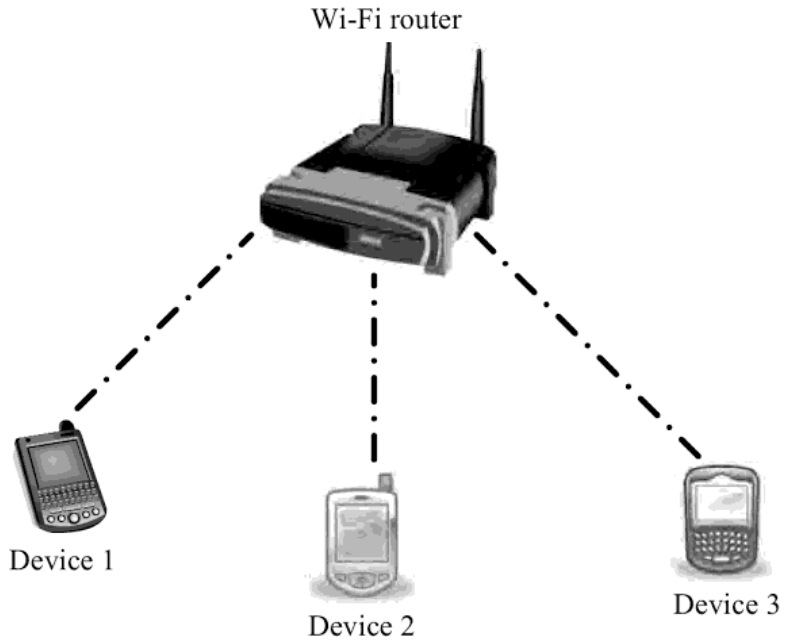


Fig: Wi-Fi Network

Wi-Fi (continued)

- For communicating with devices over a Wi-Fi network, the device when its Wi-Fi radio is turned ON, searches the available Wi-Fi network in its vicinity and lists out the Service Set Identifier (SSID) of the available networks.
- If the network is security enabled, a password may be required to connect to a particular SSID.
- Wi-Fi employs different security mechanisms like Wired Equivalency Privacy (WEP), Wireless Protected Access (WPA), etc. for securing the data communication.
- Wi-Fi supports data rates ranging from 1 Mbps to 1.73 Gbps depending on the standards (802.11a/b/g/n) and access/modulation method.
- Depending on the type of antenna and usage location (indoor/outdoor), Wi-Fi offers a range of 100 to 300 feet.

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 metres and a data rate of 20 to 250 Kbps.

ZigBee (continued)

- 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.

ZigBee (continued)

- The diagram shown in figure gives an overview of ZC, ZED and ZR in a ZigBee network.

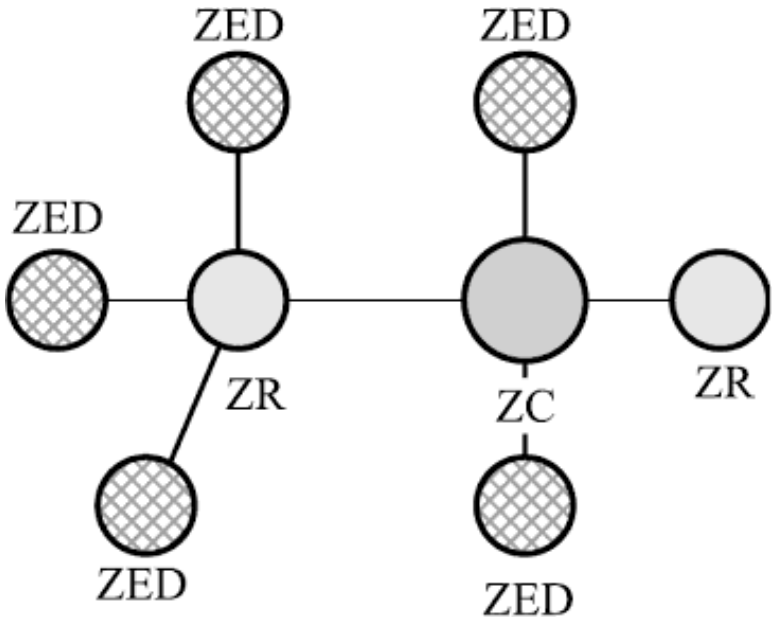


Fig: A ZigBee network model

ZigBee (continued)

- ZigBee is primarily targeting application areas like home & industrial automation, energy management, home control/security, medical/patient tracking, logistics & asset tracking and sensor networks & active RFID.
- Automatic Meter Reading (AMR), smoke detectors, wireless telemetry, HVAC control, heating control, lighting controls, environmental controls, etc. are examples for applications which can make use of the ZigBee technology.
- The specifications for ZigBee is developed and managed by the ZigBee Alliance, a non-profit consortium of leading semiconductor manufacturers, technology providers, OEMs and end-users worldwide.

General Packet Radio Service (GPRS)

- General Packet Radio Service (GPRS) 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 171.2 kbps.
- In GPRS communication, the radio channel is concurrently shared between several users instead of dedicating a radio channel to a cell phone user.

General Packet Radio Service (GPRS) (continued)

- 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), Long Term Evolution (LTE), etc. which offers higher bandwidths for communication.

Embedded Firmware

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:
  1. Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (IDE).
     - The IDE will contain an editor, compiler, linker, debugger, simulator, etc. IDES are different for different family of processors/controllers.
     - For example, Keil µVision 4 IDE is used for all family members 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.

Embedded Firmware (continued)

- The program written in high level language or assembly code 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 Assembly language based programming technique is used, the utilities supplied by the processor/controller vendors can be used to convert the source code into 'HEX File'.
  - Also third party tools are available, which may be of free of cost, for this conversion.

Embedded Firmware (continued)

- For a beginner in the embedded software field, it is strongly recommended to use the high level language based development technique.
  - Writing codes in a high level language is easy
  - The code written in high level language is highly portable
    - The same code can be used 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.
  - 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.

Embedded Firmware (continued)

- 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 he should carry an instruction set reference manual with him.
- A programmer using assembly language technique writes the program according to his view and taste.
- Often he 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 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.

Embedded Firmware (continued)

- Two types of control algorithm design exist in embedded firmware development:
  - The first type of control algorithm development is known as the infinite loop or 'super loop' based approach, where the control flow runs from top to bottom and then jumps back to the top of the program in a conventional procedure.
    - It is similar to the *while (1) { };* based technique in C.
  - The second method deals with splitting the functions to be executed into tasks and running these tasks using a scheduler which is part of a General Purpose or Real Time Embedded Operating System (GPOS/RTOS).

Other System Components

Other System Components

- 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.
- E.g.: Watchdog timer, Reset IC (or passive circuit), brown-out protection IC (or passive circuit), etc.
- 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.

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 or active low.
- Since the processor operation is synchronised to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilise before the internal reset state starts.

Reset Circuit (continued)

- 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.

Reset Circuit (continued)

- 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.

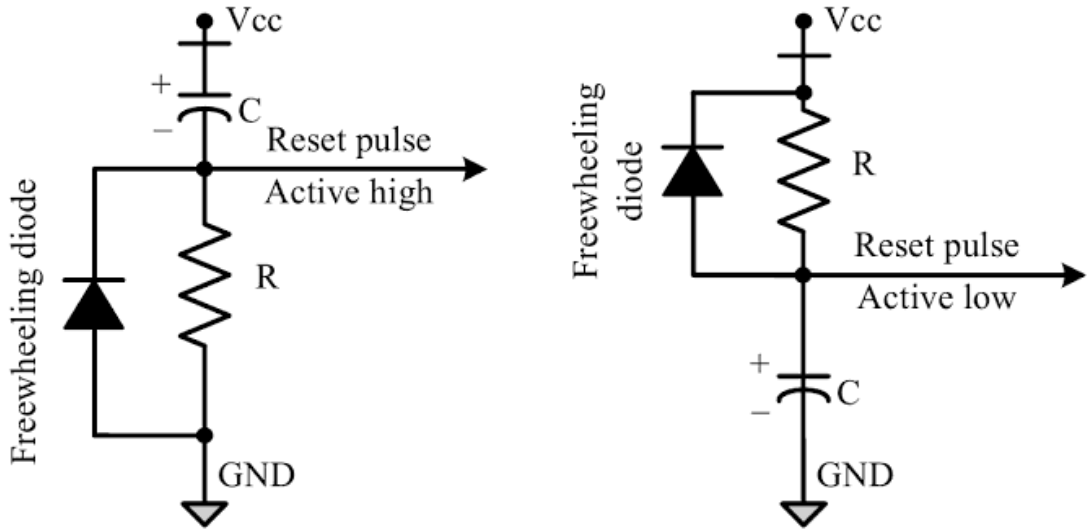


Fig: RC based reset circuit

Brown-out Protection Circuit

- Brown-out protection circuit prevents the processor/controller from unexpected program execution behaviour 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 behaviour may not be predictable if the supply voltage falls below the recommended operating voltage.
  - It may lead to situations like data corruption.

Brown-out Protection Circuit (continued)

- A brown-out protection circuit holds the processor/controller in reset state, when the 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 doesn't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs.

Brown-out Protection Circuit (continued)

- Figure illustrates a brown-out circuit implementation using Zener diode and transistor for processor/controller with active low Reset logic.
- The Zener diode D_Z and transistor Q forms the heart of this 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 R1, R2, and R3 can be selected based on the electrical characteristics of the transistor in use.
- Microprocessor Supervisor ICs like DS1232 from Maxim also provides Brown-out protection.

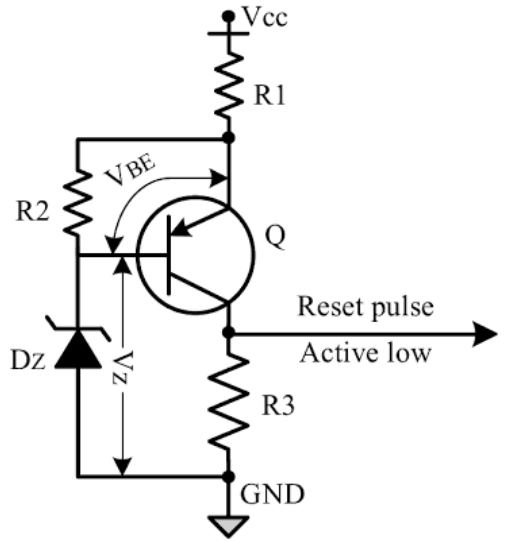


Fig: Brown-out protection circuit with Active low output

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 sync with a clock signal.
- The oscillator unit of the embedded system is responsible for generating the precise clock for the processor.
  - Analogous to the heart in living beings which produces heart beats.
- Certain processors/controllers integrate a built-in oscillator unit and simply require an external ceramic resonator/quartz crystal for producing the necessary clock signals.

Oscillator Unit (continued)

- Quartz crystals and ceramic resonators are equivalent in operation, however they possess physical difference.
- A quartz crystal is normally mounted in a hermetically sealed metal case with two leads protruding out of the case.
- Certain devices may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally.
  - Quartz crystal Oscillators are available in the form of chips and they can be used for generating the clock pulses in such cases.
- 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.

Oscillator Unit (continued)

- 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 accuracy of the crystal oscillator or ceramic resonator is normally expressed in terms of +/-ppm (Parts per million).

Oscillator Unit (continued)

- Figure illustrates the usage of quartz crystal/ceramic resonator and external oscillator chip for clock generation.

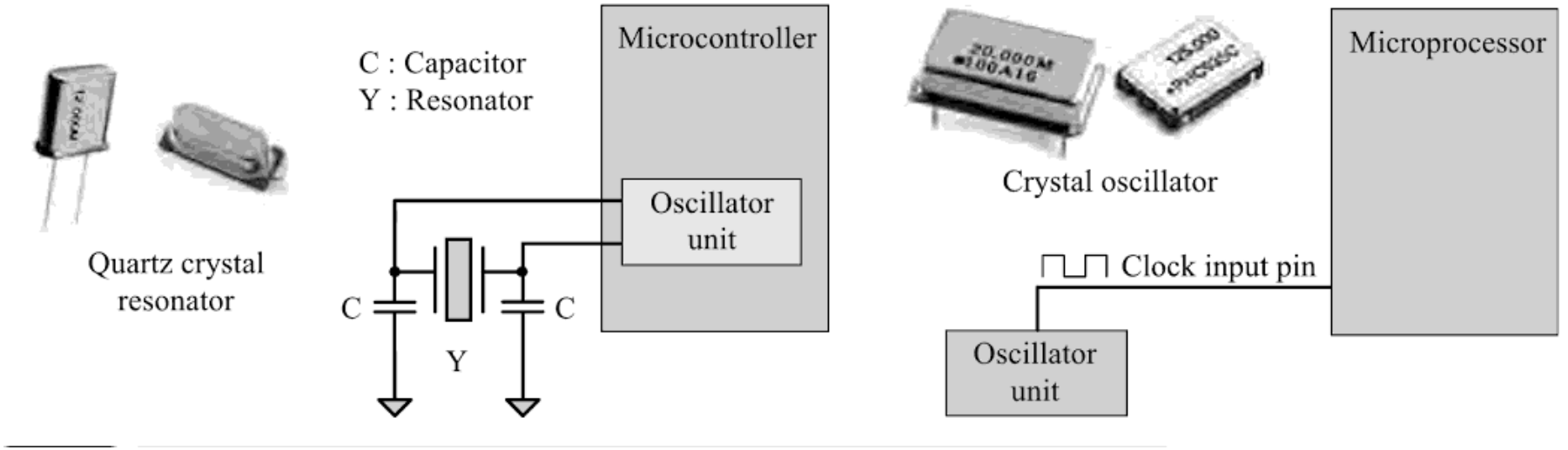


Fig: Oscillator circuitry using quartz crystal and quartz crystal oscillator

Real-Time Clock (RTC)

- Real-Time Clock (RTC) 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.

Real-Time Clock (RTC) (continued)

- For Operating System based embedded devices, a timing reference is essential for synchronising 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 updation, 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

- A watchdog timer, or simply a watchdog, is a hardware timer for monitoring the firmware execution and resetting the system processor/microcontroller when the program execution hangs up.
- 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.

Watchdog Timer (continued)

- 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 (which is susceptible to execution hang up) 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 timer you can reset the count by writing a 0 (for an up counting watchdog timer) to the watchdog timer register.

Watchdog Timer (continued)

- Most of the processors implement watchdog as a built-in component and provides status register to control the watchdog timer (like enabling and disabling watchdog functioning) and watchdog timer register for writing the count value.
- 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 DS1232 integrates a hardware watchdog timer in it.
- In modern systems running on embedded operating systems, the watchdog can be implemented in such a way that when a watchdog timeout occurs, an interrupt is generated instead of resetting the processor.
- The interrupt handler for this handles the situation in an appropriate fashion.

Watchdog Timer (continued)

- Figure illustrates the implementation of an external watchdog timer based microprocessor supervisor circuit for a small scale embedded system.

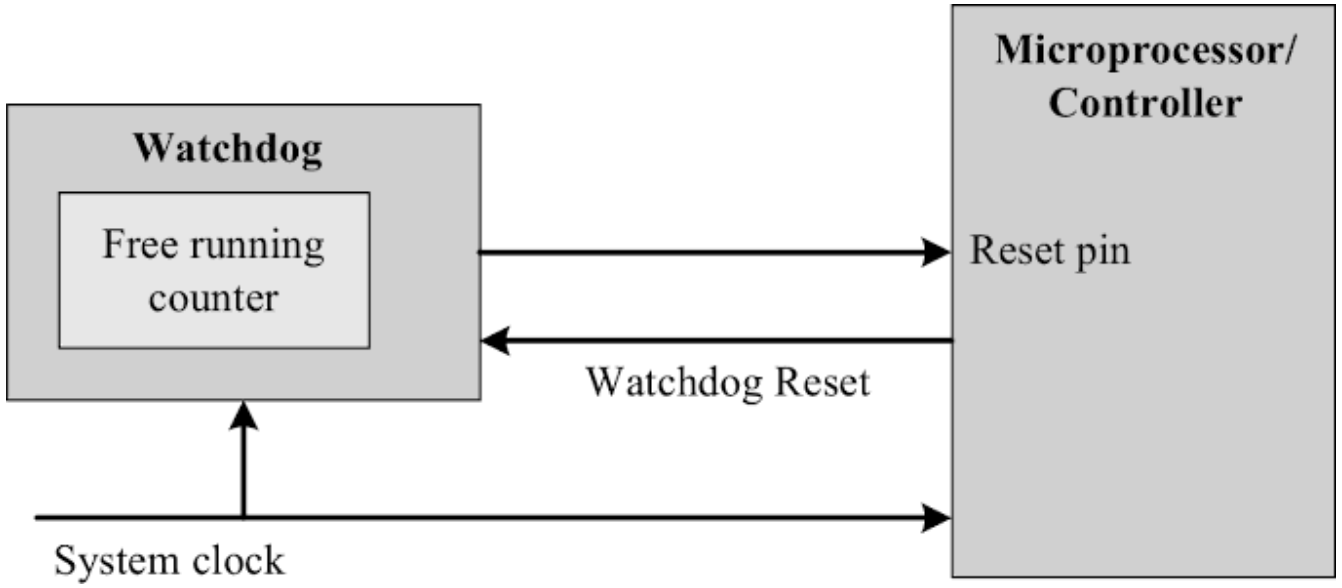


Fig: Watchdog timer for firmware execution supervision

PCB and Passive Components

- Printed Circuit Board (PCB) is the backbone of every embedded system.
- After finalising the components and the inter-connection among them, a schematic design is created and according to the schematic, the PCB is fabricated.
- PCB acts as a platform for mounting all the necessary components as per the design requirement.
- Also it acts as a platform for testing the embedded firmware.
- Apart from the subsystems mentioned already, passive electronic components like resistor, capacitor, diodes, etc. are also found on the board.
  - They are the co-workers of various chips contained in the embedded hardware.
  - They are very essential for the proper functioning of your embedded system.
  - For example for providing a regulated ripple-free supply voltage to the system, a regulator IC and spike suppressor filter capacitors are very essential.

References

1. Shibu K V, ***"Introduction to Embedded Systems"***, Tata McGraw Hill, 2009.
2. Raj Kamal, ***"Embedded Systems: Architecture and Programming",*** Tata McGraw Hill, 2008.

MODULE – 4

Embedded System Design Concepts

Characteristics and Quality Attributes of Embedded Systems

Characteristics of Embedded Systems

- Embedded systems possess certain specific characteristics.
  - 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

Characteristics of Embedded Systems (continued)

1. Application and Domain Specific
   - Each embedded system has certain functions to perform and they are developed in such a manner to do the intended functions only.
   - They cannot be used for any other purpose.
     - For example, the embedded control unit of a microwave oven cannot be replaced with an air conditioner's embedded control unit, because the embedded control units of microwave oven and air conditioner are specifically designed to perform certain specific tasks.
     - Also an embedded control unit developed for a particular domain, say telecom, cannot be replaced with another control unit designed to serve another domain like consumer electronics.

Characteristics of Embedded Systems (continued)

2. Reactive and Real Time
   - Embedded systems are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system.
   - 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.
   - Embedded systems produce changes in output in response to the changes in the input.
     - So they are generally referred as Reactive Systems.

Characteristics of Embedded Systems (continued)

- Real Time System operation means the timing behaviour of the system should be deterministic.
  - The system should respond to requests or tasks in a known amount of time.
- 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.

Characteristics of Embedded Systems (continued)

3. Operates in Harsh Environment
   - 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.
     - 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.

Characteristics of Embedded Systems (continued)

4. Distributed
   - The term ***distributed*** means that embedded systems may be a part of larger systems.
   - Many numbers of such distributed embedded systems form a single large embedded control unit.
     - For example, an automatic vending machine.
       - It 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).
       - It contains a card reader embedded unit, responsible for reading and validating the user's ATM card, transaction unit for performing transactions, a currency counter for dispatching/vending currency to the authorised person and a printer unit for printing the transaction details.
       - We can visualise 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.

Characteristics of Embedded Systems (continued)

5. Small Weight and Size
   - 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 sized and low weight products.

Characteristics of Embedded Systems (continued)

6. 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 minimise 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 the system bulky.
   - 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 application.
     - The more the power consumption the less is the battery life.

Quality Attributes of Embedded Systems

- Quality attributes are the 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 quality attributes in any embedded system development are broadly classified into two:
  - Operational Quality Attributes
  - 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 operational quality attributes are:
  1. Response
  2. Throughput
  3. Reliability
  4. Maintainability
  5. Security
  6. Safety

Operational Quality Attributes (continued)

1. Response
   - *Response* is a measure of quickness of the system.
   - It gives you an idea about how fast the 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 within this particular timeline.

Operational Quality Attributes (continued)

2. Throughput
   - *Throughput* deals with the efficiency of a system.
   - Throughput can be 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 the 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.

Operational Quality Attributes (continued)

3. Reliability
   - *Reliability* is a measure of how much percentage you can rely upon the proper functioning of the system or what is the percentage susceptibility of the system to failures.
   - System reliability is defined using two terms:
     - **Mean Time Between Failures (MTBF)**
       - Gives the frequency of failures in hours/weeks/months.
     - **Mean Time To Repair (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 the order of minutes.

Operational Quality Attributes (continued)

4. Maintainability

- *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 check-up.
- Reliability and maintainability are considered as two complementary disciplines.
- A more reliable system means a system with less corrective maintainability requirements and vice versa.
- Maintainability can be broadly classified into two categories:
  - **Scheduled or Periodic Maintenance (preventive maintenance)**
    - For example, replacing the cartridge of a printer after each 'n' number of printouts to get quality prints.
  - **Maintenance to unexpected failures (corrective maintenance)**
    - For example, repairing the printer if the paper feeding part fails.

Operational Quality Attributes (continued)

- Maintainability is also an indication of the availability of the product for use.
- In any embedded system design, the ideal value for availability is expressed as

$$A_i = \frac{MTBF}{MTBF + MTTR}$$

where

A_i = Availability in the ideal condition

$MTBF$ = Mean Time Between Failures

$MTTR$ = Mean Time To Repair

Operational Quality Attributes (continued)

Numerical Example 1

- The Mean Time Between Failure (MTBF) of an embedded product is 4 months and the Mean Time To Repair (MTTR) of the product is 2 weeks. What is the availability of the product?
- Solution:

Given $MTBF$ = 4 months = 120 days

and $MTTR$ = 2 weeks = 14 days

We know that $A_i = \frac{MTBF}{MTBF+MTT}$

$A_i = \frac{120}{120+14} = \frac{120}{134}$

$A_i = 0.8955$ or 89.55%

Operational Quality Attributes (continued)

Numerical Example 2

- The availability of an embedded product is 90%. The Mean Time Between Failure (MTBF) of the product is 30 days. What is the Mean Time To Repair (MTTR) in days/hours for the product?
- Solution:

Given $A_i = 90\% = 0.9$

and MTBF = 30 days

We know that $A_i = \frac{MTBF}{MTBF+MTT}$

$$MTBF + MTTR = \frac{MTBF}{A_i}$$

$$MTTR = \frac{MTBF}{A_i} - MTBF$$

$$MTTR = \frac{30}{0.9} - 30$$

$$MTTR = 3.33\ days\ or\ 80\ hours$$

Operational Quality Attributes (continued)

5. Security
   - Confidentiality, Integrity, and Availability are the three major measures of information security.
     - Confidentiality deals with the protection of data and application from unauthorised disclosure.
     - Integrity deals with the protection of data and application from unauthorised modification.
     - 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 a user name and password to access into a particular person's profile—This is 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 levels 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.

Operational Quality Attributes (continued)

6. Safety
   - Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products.
   - The breakdown of an embedded system may occur due to a hardware failure or a firmware failure.
   - Safety analysis is 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 non-operational quality attributes are:
  1. Testability & Debug-ability
  2. Evolvability
  3. Portability
  4. Time-to-prototype and market
  5. Per unit and total cost

Non-Operational Quality Attributes (continued)

1. Testability & Debug-ability
   - *Testability* 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 behaviour 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.

Non-Operational Quality Attributes (continued)

2. Evolvability
   - *Evolvability* is referred as the non-heritable variation (in Biology)
   - 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.

Non-Operational Quality Attributes (continued)

3. Portability
   - *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 environments, target processors/controllers and embedded operating systems.
   - 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 (e.g. Intel x86) to a different target processor (say Hitachi SH3 processor).

Non-Operational Quality Attributes (continued)

- If the firmware is written in a high level language like ‘C’, it is very easy to port the firmware
  - It as only few target processor-specific functions which can be replaced with the ones for the new target processor and re-compiling the program for the new target processor-specific settings.
  - The program then needs to be re-compiled to generate the new target processor-specific machine codes.
- If the firmware is written in Assembly Language for a particular family of processor (say x86 family), the portability is poor.
  - It is very difficult to translate the assembly language instructions to the new target processor specific language.

Non-Operational Quality Attributes (continued)

4. Time-to-Prototype and Market
   - *Time-to-market* is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial 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.
     - Competitor might release their product before you do.
     - The technology used might have superseded with a new technology.

Non-Operational Quality Attributes (continued)

- Product prototyping helps a lot in reducing time-to-market.
- 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.

Non-Operational Quality Attributes (continued)

5. Per Unit Cost and Revenue
   - *Cost* is a factor which is closely monitored by both end user and product manufacturer.
   - 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.
   - The budget and total system cost should be properly balanced to provide a marginal profit.

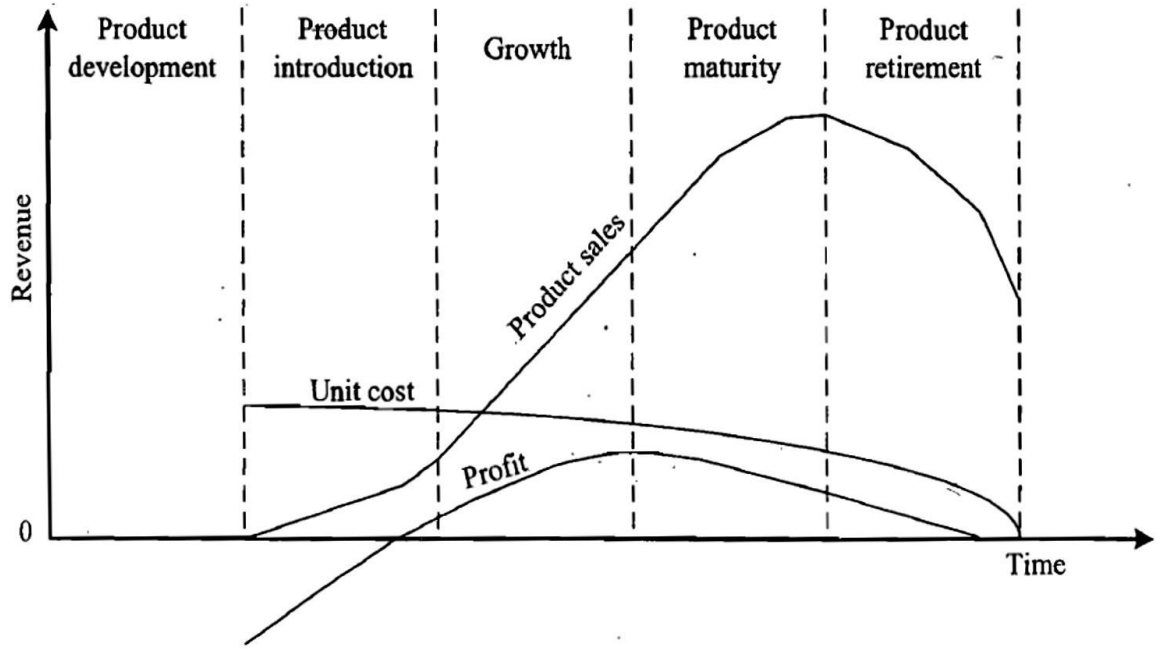
Non-Operational Quality Attributes (continued)

- The product life cycle of every embedded product has different phases:

1. Design and Development Phase:
   - The product idea generation, prototyping, Roadmap definition, actual product design and development are the activities carried out during this phase.
   - There is only investment and no returns.
1. Product Introduction Phase:
   - Once the product is ready to sell, it is introduced to the market.
   - 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.
2. Growth Phase
   - The product grabs high market share.
3. Maturity Phase:
   - The growth and sales will be steady and the revenue reaches at its peak.
4. Product Retirement/Decline Phase:
   - 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.

Non-Operational Quality Attributes (continued)

- 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.



Product Life Cycle (PLC) curve

Non-Operational Quality Attributes (continued)

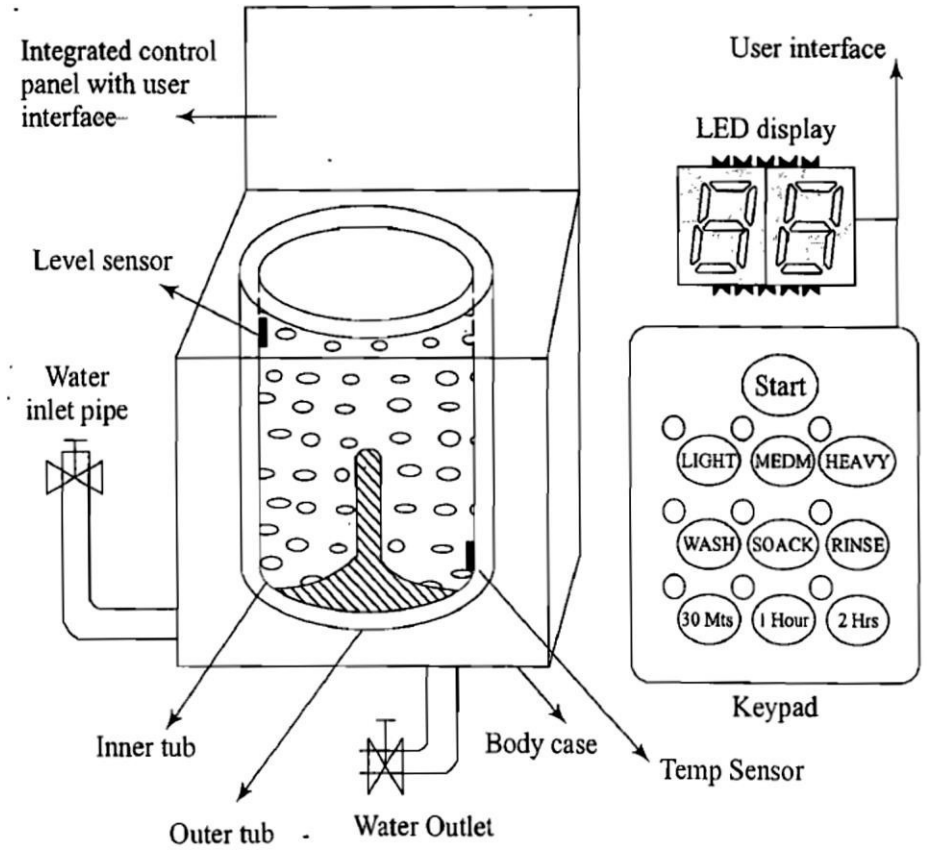
- From the graph, it is clear that the total revenue increases from the product introduction stage to the product maturity stage.
- The revenue peaks at the maturity stage and starts falling in the decline/retirement Stage.
- The unit cost is very high during the introductory stage.
  - A typical example is cell phone; if you buy a new model of cell phone during its launch time, the price will be high and you will get the same model with a very reduced price after three or four months of its launching).
- The profit increases with increase in sales and attains a steady value and then falls with a dip in sales.
- You can see a negative value for profit during the initial period.
- It is because during the product development phase there is only investment and no returns.
- Profit occurs only when the total returns exceed the investment and operating cost.

Embedded Systems – Application and Domain Specific

Washing Machine – Application-Specific Embedded System

- 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.
  - All these components can be seen in a washing machine.

Washing Machine – Application-Specific Embedded System (continued)



Washing Machine – Functional Block Diagram

Washing Machine – Application-Specific Embedded System (continued)

- The actuator part of the washing machine consists of a motorised 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 microprocessor/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.

Washing Machine – Application-Specific Embedded System (continued)

Top Loading
Washing Machine

Front Loading
Washing Machine

Washing Machine – Application-Specific Embedded System (continued)

- 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.

Washing Machine – Application-Specific Embedded System (continued)

- 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'.*
- 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.

Washing Machine – Application-Specific Embedded System (continued)

- 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.

Washing Machine – Application-Specific Embedded System (continued)

Integrated Control Panel of a Washing Machine

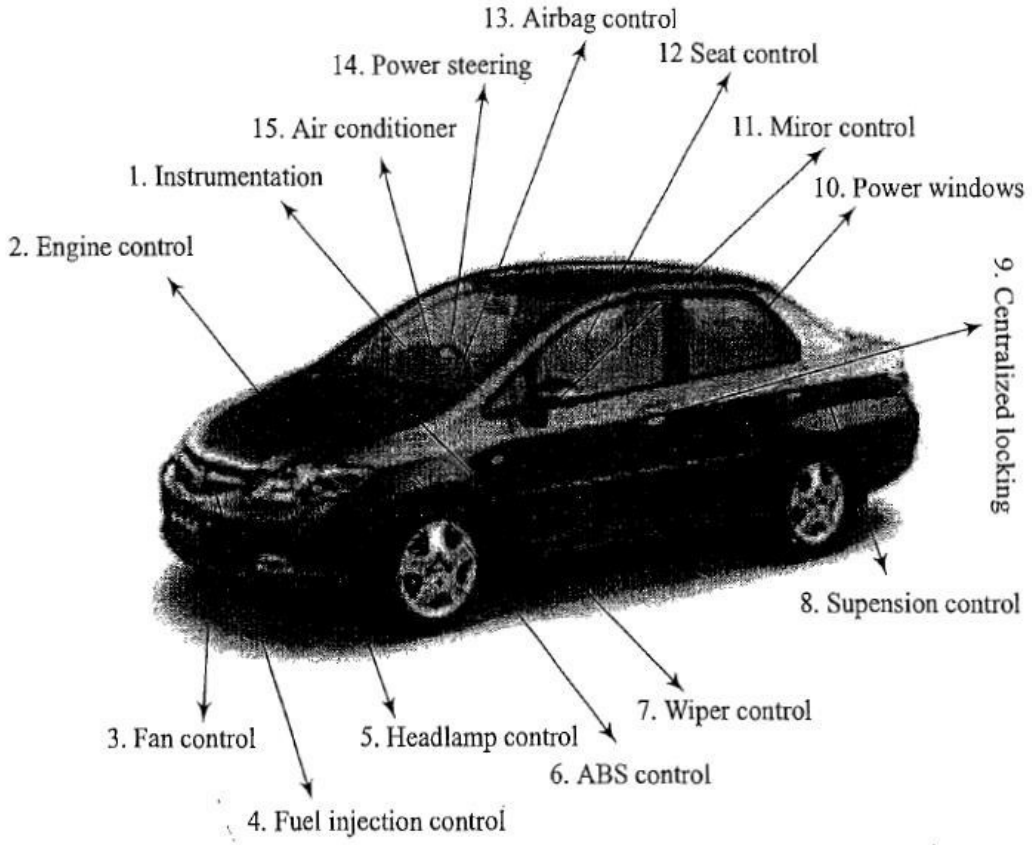
Washing Machine – Application-Specific Embedded System (continued)

- 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 namely *Wash, Spin and Rinse*, cloth type selector namely *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, namely, water temperature sensor, water level sensor, etc. and actuator interface including motor control for agitator and tub movement control, inlet water flow control, etc.

Automotive – Domain-Specific Embedded System

- The major application domains of embedded systems are consumer, industrial, automotive, telecom, etc.
  - Telecom and automotive industry holds a big market share.
- Figure below gives an overview of the various types of electronic control units employed automotive applications.

Automotive – Domain-Specific Embedded System (continued)



Embedded System in the Automotive Domain

Inner Workings 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 vehicle like Mercedes S and BMW 7 may contain 75 to 100 numbers of embedded controllers.

Inner Workings of Automotive Embedded Systems (continued)

- Government regulations on fuel economy, environmental factors and emission standards and increasing customer demands on safety, comfort and infotainment forces the automotive manufactures to opt for sophisticated embedded control units within the vehicle.
- The first embedded system used in automotive application was the microprocessor based fuel injection system introduced by Volkswagen 1600 in 1968.

Inner Workings of Automotive Embedded Systems (continued)

- The electronic control units (ECUs) used in the automotive embedded industry can be broadly classified into two:
  - High-speed Electronic Control Units (HECUs):
    - These 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):
    - These 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.
- Different types of serial interface buses are:
  - Controller Area Network (CAN) Bus
  - Local Interconnect Network (LIN) Bus
  - Media-Oriented System Transport (MOST) Bus

Automotive Communication Buses (continued)

- Controller Area Network (CAN) Bus
  - CAN Bus was originally proposed by Robert Bosch, pioneer in the Automotive embedded solution providers.
  - It supports medium speed (ISO11519-class B with data rates up to 125 Kbps) and high speed (IS011898 class C with data rates up to 1 Mbps) 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.

Automotive Communication Buses (continued)

- Local Interconnect Network (LIN) Bus
  - 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.

Automotive Communication Buses (continued)

- Media-Oriented System Transport (MOST) Bus
  - MOST Bus is targeted for automotive audio/video equipment interfacing.
  - It is a multimedia fibre-optic point-to-point network implemented in a star, ring or daisy-chained topology over optical fibre cables.
  - 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 visualised in three verticals namely, *silicon providers, tools and platform providers* and *solution providers*.
- Silicon Providers
  - They 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 Analog Devices, Xilinx, Atmel, Maxim/Dallas, NXP Semiconductors, Renesas, Texas Instruments, Fujitsu, Infineon, NEC, etc.

Key Players of the Automotive Embedded Market (continued)

- Tools and Platform Providers
  - They are manufacturers and suppliers of various kinds of development tools and Real Time Embedded Operating Systems for developing and debugging different control unit related applications.
  - Tools fall into two categories, namely embedded software application development tools and embedded hardware development tools.
    - Sometimes the silicon suppliers provide the development suite for application development using their chip.
    - Some third party suppliers may also provide development kits and libraries.
  - Some of the leading suppliers of tools and platforms in automotive embedded applications are ENEA, The MathWorks, MATLAB, Keil Software, Lauterbach, ARTiSAN, Microsoft, etc.

Key Players of the Automotive Embedded Market (continued)

- Solution Providers
  - They 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 Bosch Automotive, DENSO Automotive, Infosys Technologies, Delphi, etc.

Hardware Software Co-Design and Program Modelling

Hardware Software Co-Design

- In the traditional embedded system development approach, the hardware software partitioning is done at an early stage.
  - 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.

Hardware Software Co-Design (continued)

- During the co-design process, the product requirements captured from the customer are converted into system level needs or processing requirements.
  - At this point of time it is not segregated as either hardware requirement or software requirement, instead it is specified as functional requirement.
- The system level processing requirements are then transferred into functions which can be simulated and verified against performance and functionality.
- The Architecture design follows the system design.
  - The partition of system level processing requirements into hardware and software takes place during the architecture design phase.
  - Each system level processing requirement is mapped as either hardware and/or software requirement.
  - The partitioning is performed based on the hardware-software trade-offs.

Hardware Software Co-Design (continued)

- The architectural design results in the detailed behavioural description of the hardware requirement and the definition of the software required for the hardware.
- The processing requirement behaviour is usually captured using computational models.
  - The models representing the software processing requirements are translated into firmware implementation using programming languages.

Fundamental Issues in Hardware Software Co-Design

- The fundamental issues in hardware software co-design are:
  - Selecting the Model
  - Selecting the Architecture
  - Selecting the Language
  - Partitioning System Requirements into Hardware and Software

Fundamental Issues in Hardware Software Co-Design (continued)

- 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 a variety 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 components is revealed and the designer has to switch to a model capable of capturing the system's structure.

Fundamental Issues in Hardware Software Co-Design (continued)

- 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.
  - The commonly used architectures in system design are 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.
    - 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.).

Fundamental Issues in Hardware Software Co-Design (continued)

- The controller architecture implements the finite state machine model using a state register and two combinational circuits.
  - The state register holds the present state and the combinational circuits implement the logic for next state and output.
- 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
    - The datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units.
  - Ports connect the datapath to multiple buses.
  - Most of the time the arithmetic units are connected in parallel with pipelining support for bringing high performance.

Fundamental Issues in Hardware Software Co-Design (continued)

- 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.
  - Normally the datapath is implemented in a chip and the I/O pins of the chip acts as the data input output ports for the chip resident data path.

Fundamental Issues in Hardware Software Co-Design (continued)

- 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.
    - e.g. Reading a register value and comparing it with a given value and then transfer the program execution to a new address location is done using the CJNE instruction for 8051 ISA).
  - 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.

Fundamental Issues in Hardware Software Co-Design (continued)

- The Reduced Instruction Set Computing (RISC) architecture uses instruction set representing simple operations.
  - It requires the execution of multiple RISC instructions to perform a complex operation.
  - The datapath 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.

Fundamental Issues in Hardware Software Co-Design (continued)

- 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 Elements.
    - The scheduling of the instruction execution and controlling of each PE is performed through a single controller.
    - The SIMD architecture forms the basis of re-configurable processor.
  - In MIMD architecture, the Processing Elements 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.

Fundamental Issues in Hardware Software Co-Design (continued)

- Selecting the Language
  - A programming language captures a 'Computational Model' and maps it into architecture.
  - There is no hard and fast rule to specify this 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.

Fundamental Issues in Hardware Software Co-Design (continued)

- Partitioning System Requirements into Hardware and Software
  - From an implementation perspective, 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

- The commonly used computational models in embedded system design are:
  - Data Flow Graph Model
  - Control Data Flow Graph Model
  - State Machine Model
  - Sequential Program Model
  - Concurrent/Communicating Process Model
  - Object-Oriented Model

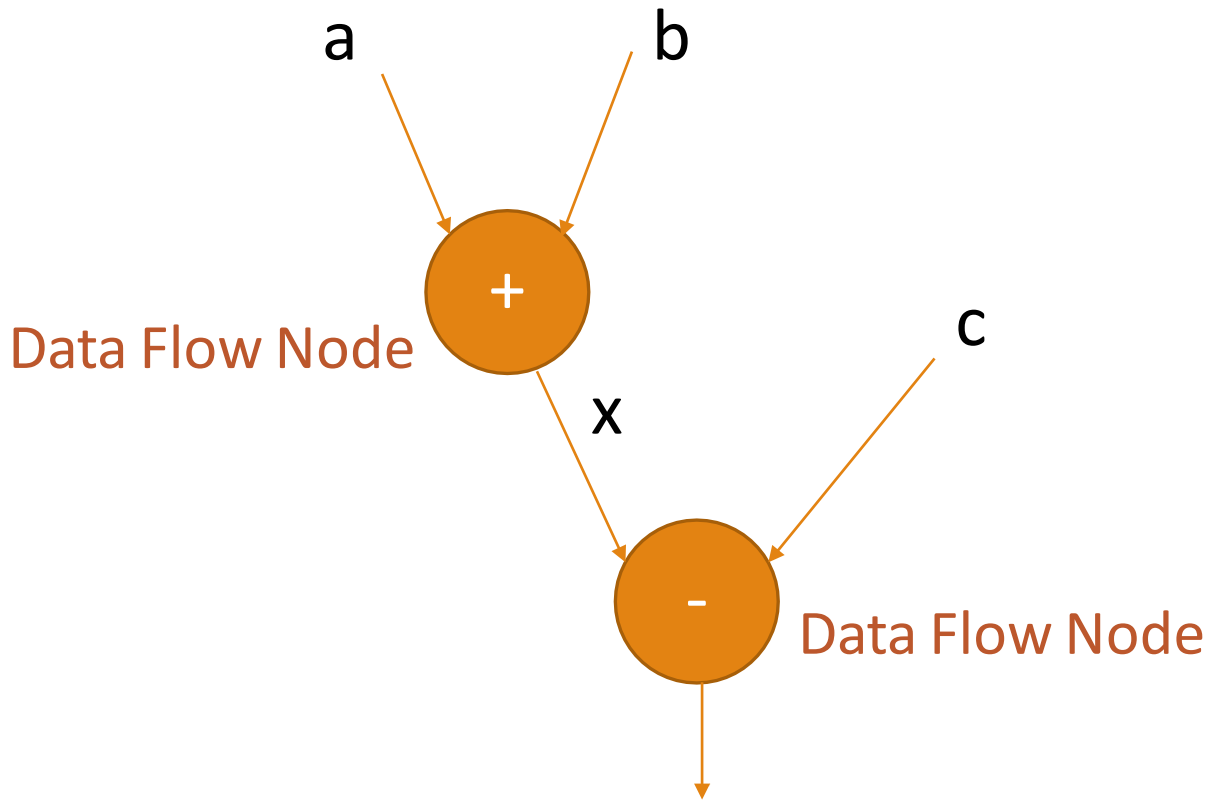
Data Flow Graph/Diagram (DFG) Model

- The Data Flow Graph (DFG) model translates the data processing requirements into a data flow graph.
- It is a data driven model in which the program execution is determined by data.
- This model emphasises on the data and operations on the data which transforms the input data to output data.
- Embedded applications which are computational intensive and data driven are modelled using the DFG model.
  - DSP applications are typical examples for it.

Data Flow Graph/Diagram (DFG) Model (continued)

- Data Flow Graph (DFG) 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.
- Suppose one of the functions in our application contains the computational requirement $x = a + b$ and $y = x - c$.
- Figure illustrates the implementation of a DFG model for implementing these requirements.

Data Flow Graph/Diagram (DFG) Model (continued)



Data Flow Graph (DFG) Model

Data Flow Graph/Diagram (DFG) Model (continued)

- 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.

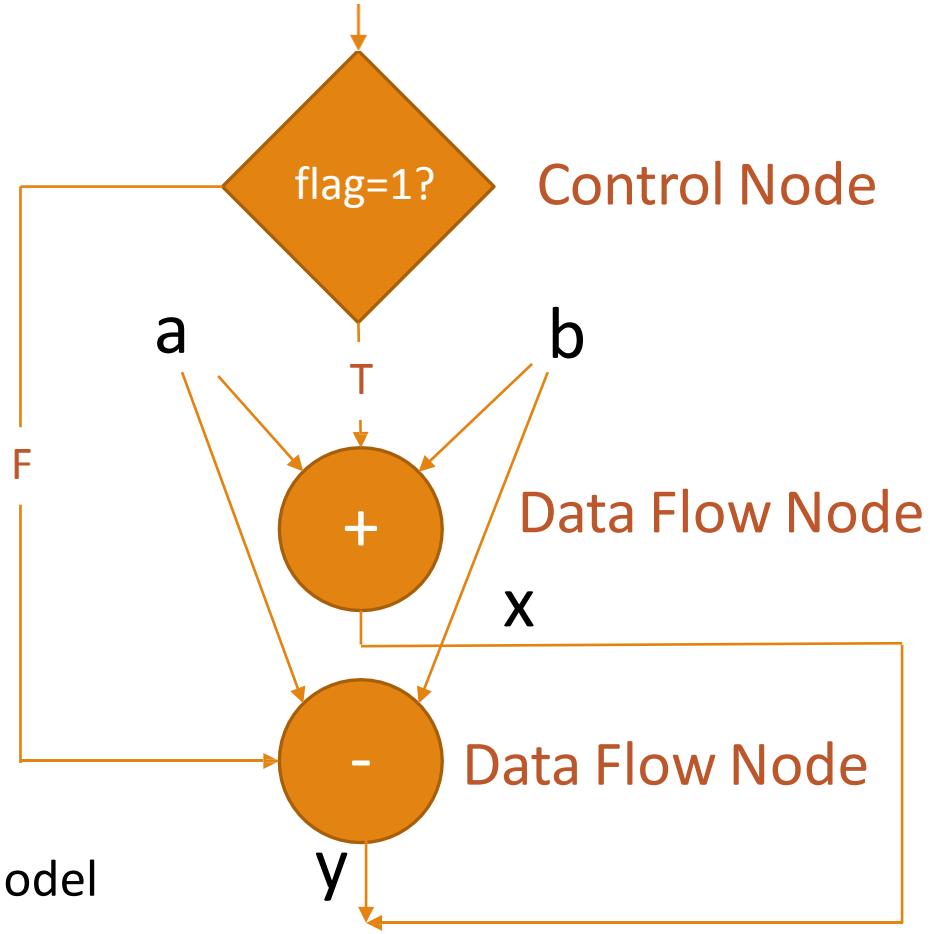
Control Data Flow Graph/Diagram (CDFG) Model

- The DFG model is a data driven model in which the execution is controlled by data and it doesn't involve any control operations (conditionals).
- The Control DFG (CDFG) model is used for modelling 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.

Control Data Flow Graph/Diagram (CDFG) Model (continued)

- Consider 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 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 modelled to a concurrent process model.
- The decision on which process is to be executed is determined by the control node.

Control Data Flow Graph/Diagram (CDFG) Model (continued)



Control Data Flow Graph (CDFG) Model

Control Data Flow Graph/Diagram (CDFG) Model (continued)

- A real world example for modelling the embedded application using CDFG is capturing and saving of the image to a format set by the user in a digital still camera.
- Here everything is data driven.
  - Analog Front End converts the CCD sensor generated analog signal to Digital Signal
  - The data from ADC is stored 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 modelling reactive or event-driven embedded systems whose processing behaviour are dependent on state transitions.
  - Embedded systems used in the control and industrial applications are typical examples for event driven systems.
- The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'.
  - *State* 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.

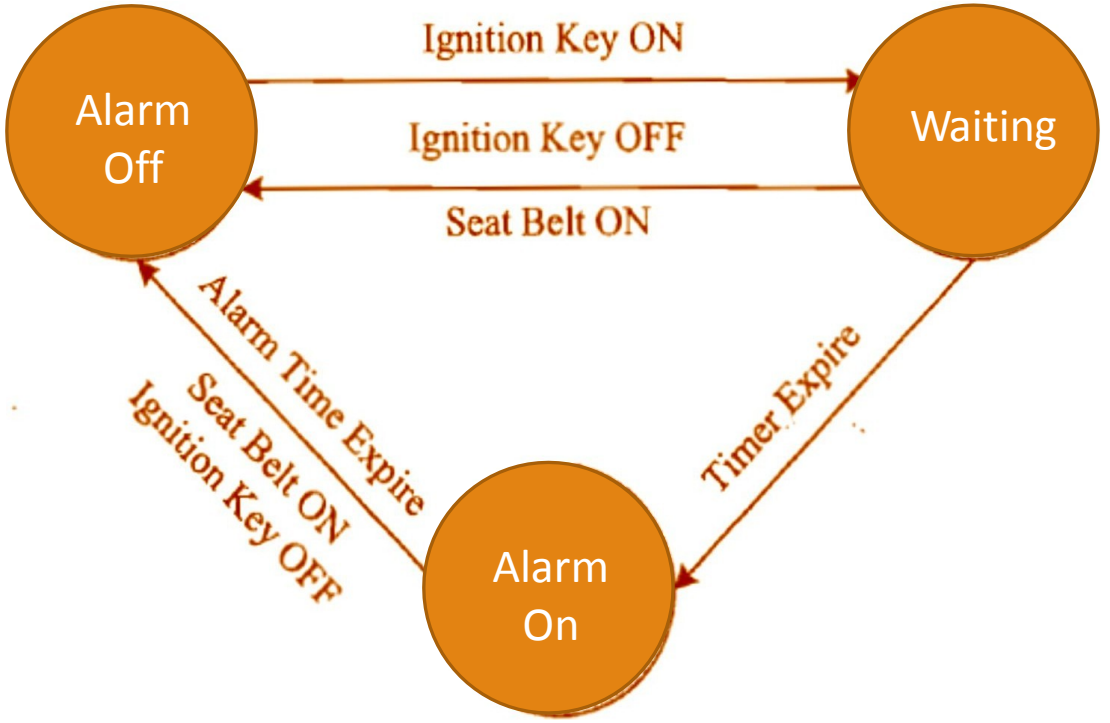
Finite State Machine (FSM) Model

- A Finite State Machine (FSM) model is one in which the number of states are finite.
  - 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.
  1. 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.
  2. The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Finite State Machine (FSM) Model (continued)

- Here the states are
  - 'Alarm Off’
  - 'Waiting’
  - 'Alarm On’
- The events are
  - 'Ignition Key ON’
  - 'Ignition Key OFF’
  - 'Timer Expire’
  - 'Alarm Time Expire’
  - 'Seat Belt ON’
- Using the FSM, the system requirements can be modeled as given in figure.

Finite State Machine (FSM) Model (continued)

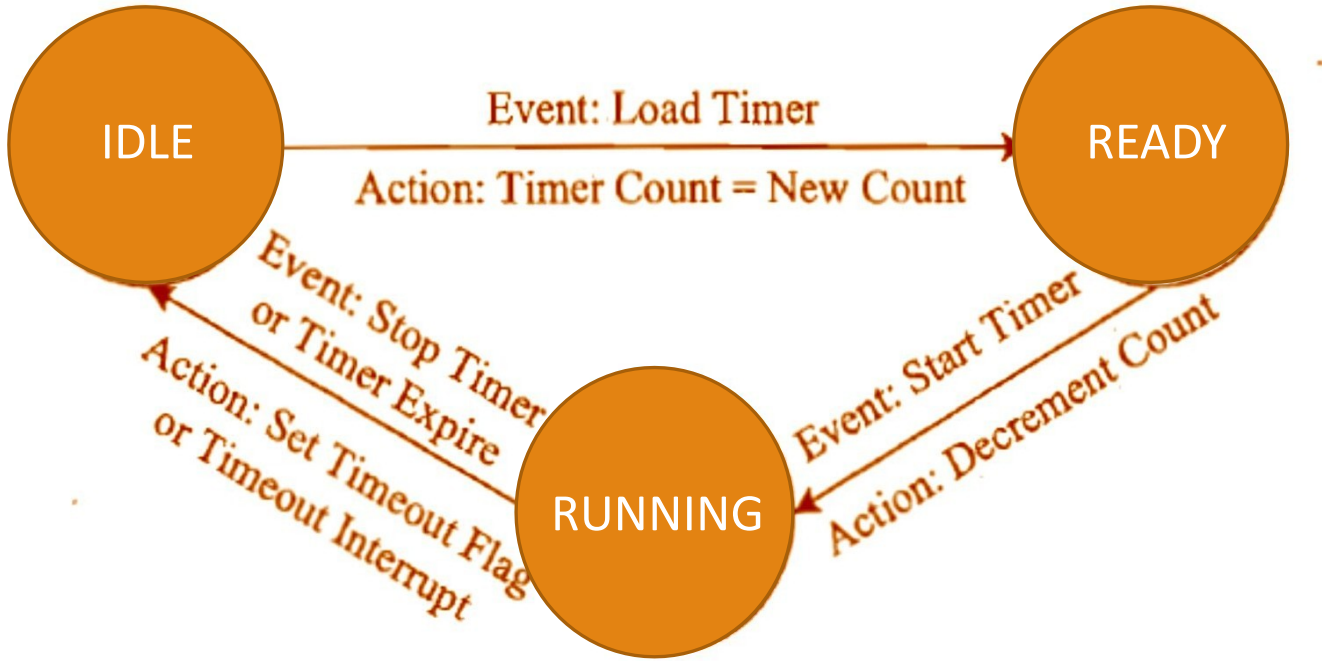


FSM Model for Automatic Seat Belt Warning System

Finite State Machine (FSM) Model (continued)

- 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 modelled as shown in the figure.

Finite State Machine (FSM) Model (continued)



FSM Model for Timer

Finite State Machine (FSM) Model (continued)

- 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.

FSM Model - Example 1

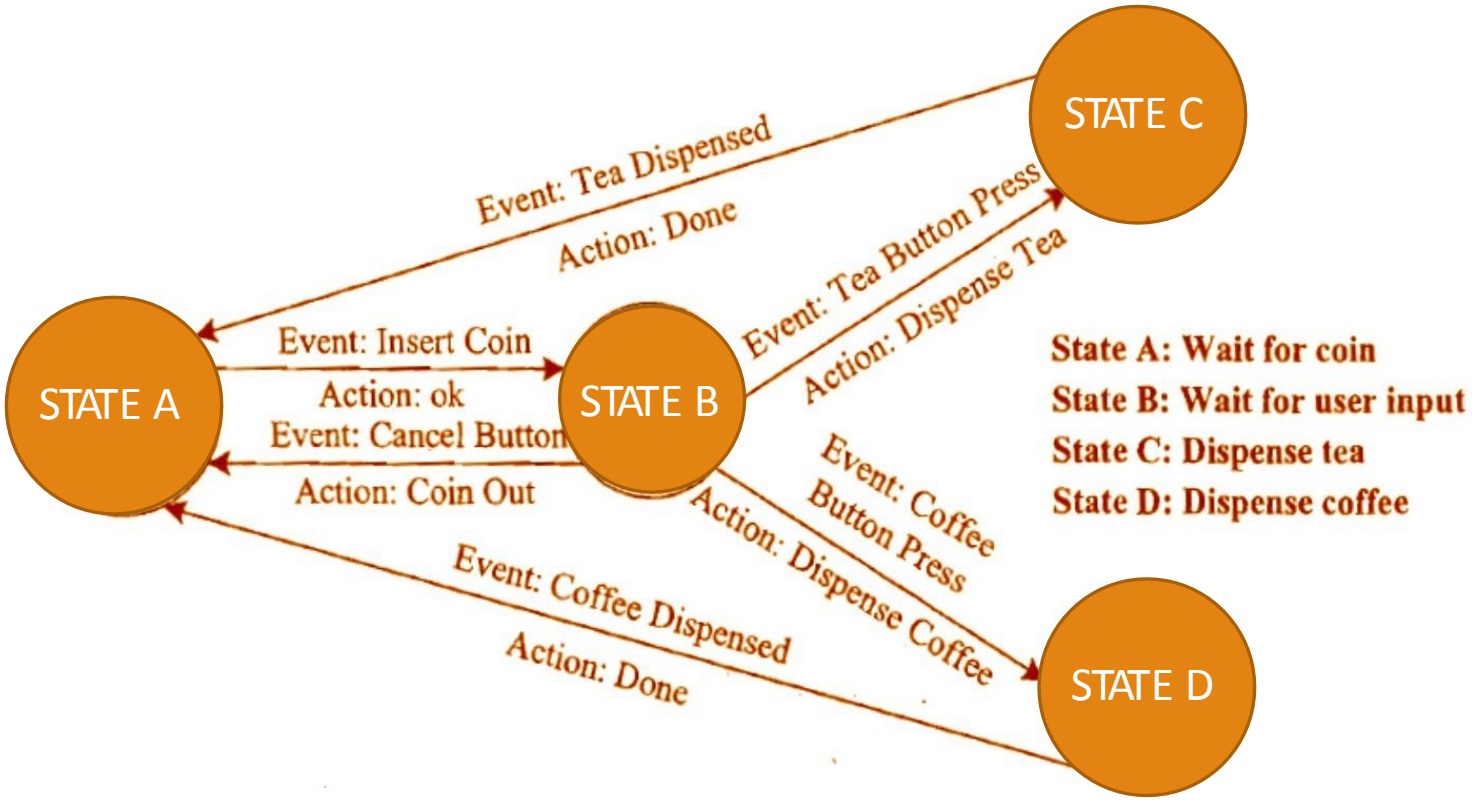
Design an automatic tea/coffee vending machine based on FSM model for the following requirement.

- 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.

Solution

- The FSM Model contains four states namely,
  - 'Wait for coin’
  - 'Wait for User Input’
  - 'Dispense Tea'
  - 'Dispense Coffee'

FSM Model - Example 1 (continued)



FSM Model for Automatic Tea/Coffee Vending Machine

FSM Model - Example 1 (continued)

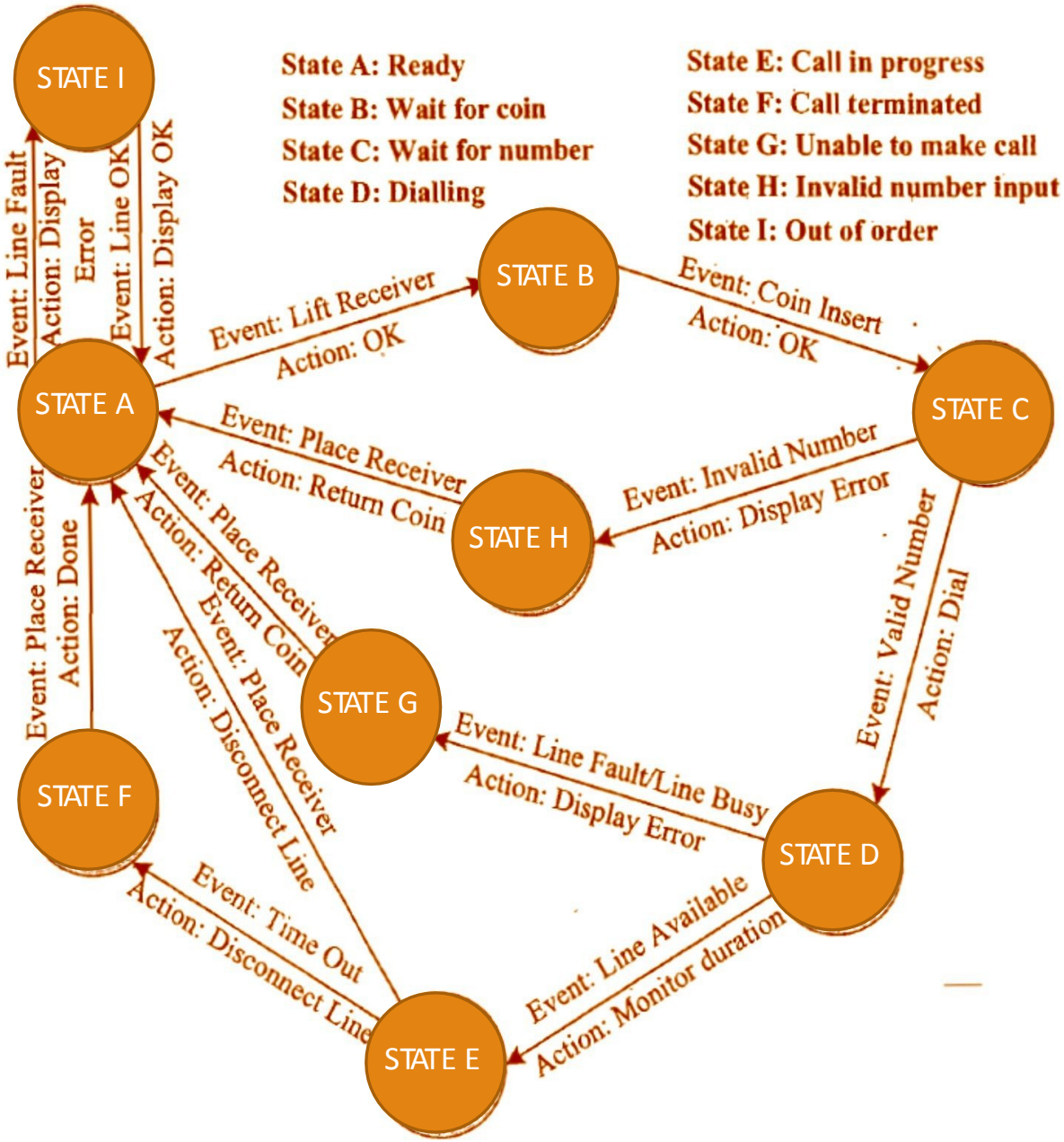
- The event 'Insert Coin' (5 rupee coin insertion), transitions the state to 'Wait for User Input’.
- The system stays in this state until a user input is received from the buttons 'Cancel', 'Tea' or 'Coffee' (Tea and Coffee are the drink select button).
- If the event triggered in 'Wait State' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin’.
- If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press, the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively.
- Once the coffee/tea vending is over, the respective states transition back to the 'Wait for Coin' state.

FSM Model - 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.

FSM Model for
Coin Operated Telephone
System



Sequential Program Model

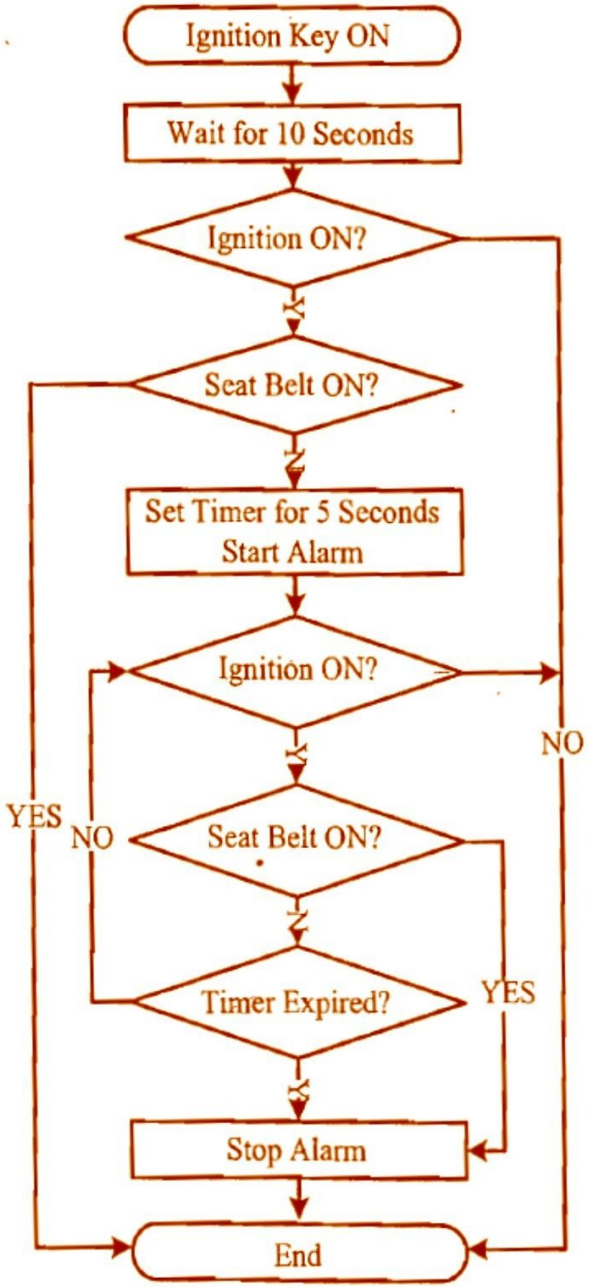
- In the Sequential Program Model, the functions or processing requirements are executed in sequence.
  - It is same as the conventional procedural programming.
- Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
- Finite State Machines (FSMs) and Flow Charts are used for modelling sequential program.
  - The FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow.

Sequential Program Model (continued)

- 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()==ON)&&(timer_expire()==NO));
   stop_alarm();
   }
  }
 }
}
```

Figure illustrates the flow chart approach for modelling the 'Seat Belt Warning' system explained in the FSM modelling section.



Sequential Program Model for Seat Belt Warning System

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 utilisation, 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 by switching the task execution, when the subtask under execution goes to a wait or sleep mode.
- However, concurrent processing model requires additional overheads in task scheduling, task synchronisation and communication.

Concurrent/Communicating Process Model (continued)

- As an example, consider the implementation of the 'Seat Belt Warning' system using 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)

Concurrent/Communicating Process Model (continued)

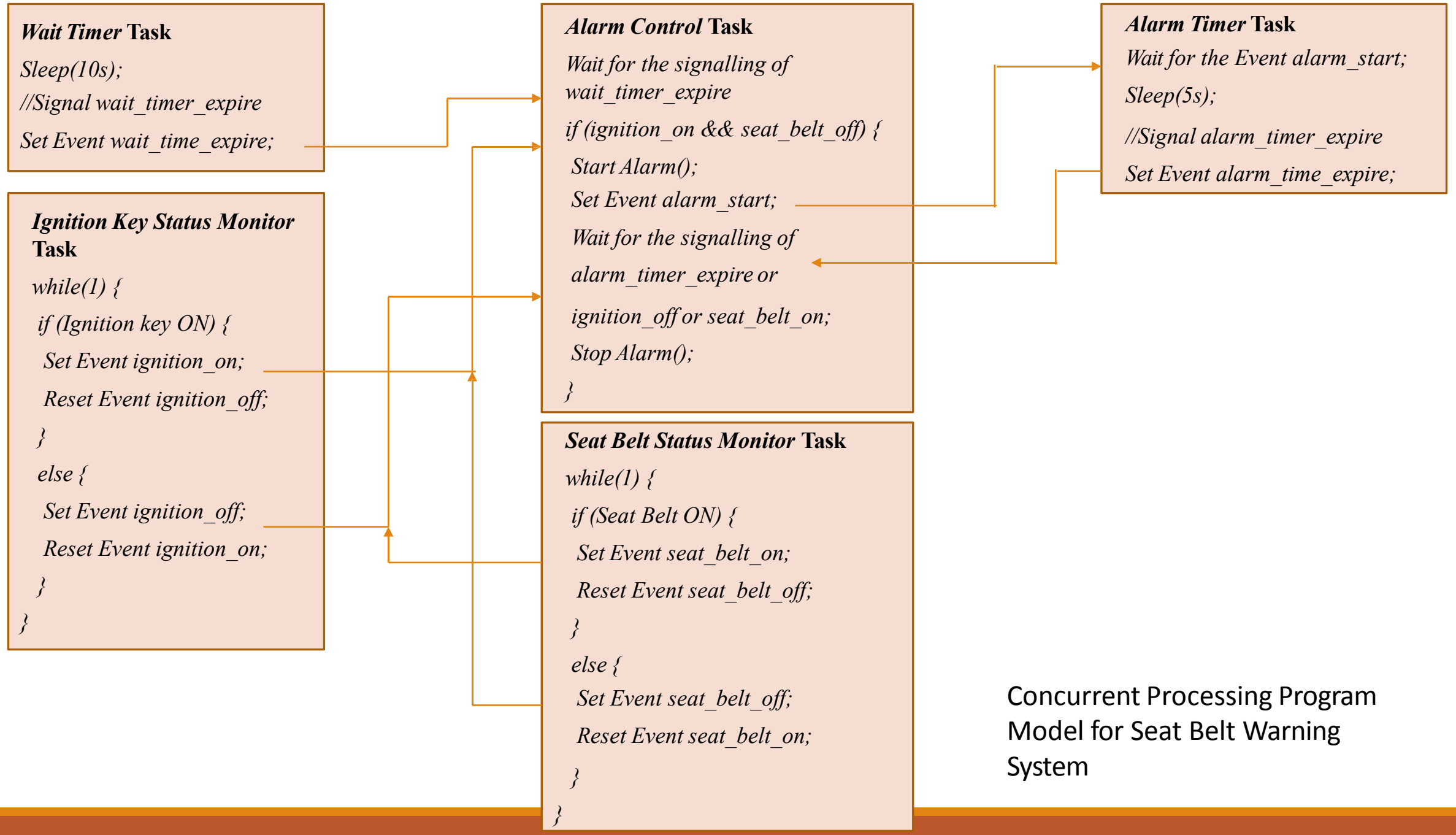
- The tasks cannot be executed them randomly or sequentially.
- We need to synchronise their execution through some mechanism.
  - For example, the alarm control task is executed only when the wait timer is expired and if the ignition key is in the ON state and seat belt is in the OFF state.
- We will use events to indicate these scenarios.
- The *wait_timer_expire* event is associated with the timer task event and it will be in the reset state initially and it is set when the timer expires.
- Similarly, events *ignition_on* and *ignition_off* are associated with the task ignition key status monitoring and the events *seat_belt_on* and *seat_belt_off* are associated with the task seat belt status monitoring.

Concurrent/Communicating Process Model (continued)

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 modelling 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 modelling, *object is an entity used for representing or modelling a particular piece of the system*.
  - Each object is characterised by a set of unique behaviour and state.

Object-Oriented Model (continued)

- 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 behaviour through member functions.
- The member variables and member functions of a class can be private, public or protected.
  - Private member variables and functions are accessible only within the class, 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

Introduction to Embedded Firmware Design

- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements.
- Firmware is considered as the master brain of the embedded system.
- Imparting intelligence to an Embedded system is a one time process and it can happen at any stage.
  - It can be immediately after the fabrication of the embedded hardware or at a later stage.
- For most of the embedded products, the embedded firmware is stored at a permanent memory (ROM) and they are non-alterable by end users.
  - Some of the embedded products used in the Control and Instrumentation domain are adaptive.

Introduction to Embedded Firmware Design (continued)

- 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.
- Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modelling tools.
- Once the program model is created, the next step is the implementation of the tasks and actions by capturing the model using a language which is understandable by the target processor/controller.

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:
  - Super Loop Based Approach (Conventional Procedural Based Design)
  - Embedded Operating System (OS) Based Approach

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.
- It 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.
- In a multiple task based system, each task is executed in serial in this approach.

Super Loop Based Approach (continued)

- The firmware execution flow for this will be
  1. Configure the common parameters and perform initialisation for various hardware components memory, registers, etc.
  2. Start the first task and execute it
  3. Execute the second task
  4. Execute the next task
  5. :
  6. :
  7. Execute the last defined task
  8. Jump back to the first task and follow the same flow

Super Loop Based Approach (continued)

- 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 visualise 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();
 }
}
```

Super Loop Based Approach (continued)

- Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation.
  - This repetition is achieved by using an infinite loop.
  - Hence the name 'Super loop based approach’.
- 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.

Super Loop Based Approach (continued)

- Advantage of Super Loop Based Approach:
  - It doesn't require an operating system
    - There is no need for scheduling which task is to be executed and assigning priority to each task.
    - 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 Approach (continued)

- Applications of Super Loop Based Approach:
  - This type of design is deployed in low-cost embedded products and products where response time is not time critical.
  - Some embedded products demands this type of approach if some tasks itself are sequential.
  - 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.
    - It should strictly follow a specified sequence and the combination of these series of tasks constitutes a single task-namely data read/write.

Super Loop Based Approach (continued)

- 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.
- The keyboard scanning and display updating happens at a reasonably high rate.
- 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.
- It is not economical to embed an OS into low cost products and it is an utter waste to do so if the response requirements are not crucial.

Super Loop Based Approach (continued)

- Drawbacks of Super Loop Based Approach:
  - Any failure in any part of a single task will affect the total system.
    - If the program hangs up at some point while executing a task, it will remain there forever and ultimately the product stops functioning.
      - Watch Dog Timers (WDTs) can be used to overcome this, but this, in turn, may cause additional hardware cost and firmware overheads.
  - 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, 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.
      - Interrupts can be used for external events requiring real time attention.

Embedded Operating System (OS) Based Approach

- The Embedded 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.

Embedded Operating System (OS) Based Approach (continued)

- 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.
  - Examples of Embedded products using Microsoft Windows XP OS are Personal Digital Assistants (PDAs), Hand held devices/Portable devices and Point of Sale (POS) terminals.
- Use of GPOS in embedded products merges the demarcation of Embedded Systems and general computing systems in terms of OS.
- For developing applications on top of the OS, the OS supported APIs are used.
- Similar to the different hardware specific drivers, OS based applications also require 'Driver software' for different hardware present on the board to communicate with them.

Embedded Operating System (OS) Based Approach (continued)

- Real Time Operating System (RTOS) based design approach is employed in embedded products demanding Real-time response.
  - RTOS responds 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 CE', 'pSOS', 'VxWorks', 'ThreadX', 'MicroC/OS-II', 'Embedded Linux', 'Symbian',* etc. are examples of RTOS employed in embedded product development.
- Mobile phones, PDAs (Based on Windows CE/Windows Mobile Platforms), handheld devices, etc. are examples of 'Embedded Products' based on RTOS.
  - Most of the mobile phones are built around the popular RTOS 'Symbian'. *(sic)*

Embedded Firmware Development Languages

- For embedded firmware development, we can use either
  - a target processor/controller specific language (Generally known as Assembly language or low level language) or
  - a target processor/controller independent language (Like 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*'
  - 'Machine language' is a processor understandable language.
- Machine language is a binary representation and it consists of 1s and 0s.
- 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.***

Assembly Language Based Development

- *'Assembly language*' is the human readable notation of *'machine language*'
  - 'Machine language' is a processor understandable language.
- Machine language is a binary representation and it consists of 1s and 0s.
- 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.***

Assembly Language Based Development (continued)

- Assembly Language program was the most common type of programming adopted in the beginning of software revolution.
- Even today also almost all low level, system related, programming is carried out using assembly language.
- In particular, assembly language is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers.

Assembly Language Based Development (continued)

- 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.
- For example

```
MOV A, #30
```

Here `MOV A` is the Opcode and `#30` is the operand

- The same instruction when written in machine language will look like

```
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.

Assembly Language Based Development (continued)

- Each line of an assembly language program is split into four fields as given below

LABEL OPCODE OPERAND COMMENTS

- A 'LABEL' is an optional identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located.
- For example

```
DELAY: MOV R0,#255 ;Load Register R0 with 255
```

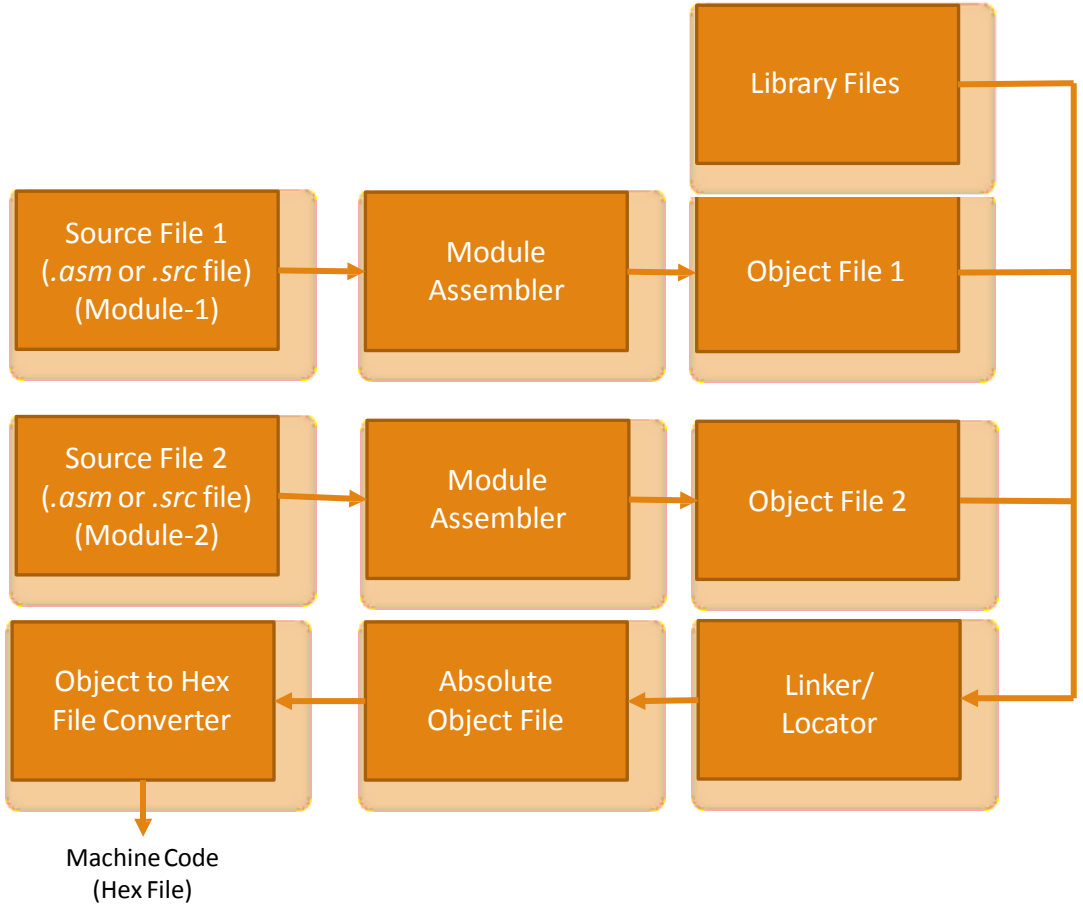
LABEL OPCODE OPERAND COMMENT

Assembly Language Based Development (continued)

- The Assembly language program written in assembly code is saved as *.asm* (Assembly file) file or an *.src* (source) file (also *.s* file).
- Any text editor like ‘Notepad' or 'WordPad' from Microsoft or the text editor provided by an Integrated Development (IDE) tool can be used for writing the assembly instructions.
- Similar to 'C' and other high level language programming, we can have multiple source files called modules in assembly language programming.
  - Each module is represented by an '*.asm*' or '*.src*' file.
  - This approach is known as 'Modular Programming’.
- Modular programming is employed when the program is too complex or too big.
  - In 'Modular Programming', the entire code is divided into submodules and each module is made re-usable.
  - Modular Programs are usually easy to code, debug and alter.

Assembly Language Based Development (continued)

Assembly language to machine language conversion process



Assembly Language Based Development (continued)

- **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.
    - 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 are illustrated in the figure.

Assembly Language Based Development (continued)

- 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 a re-locatable segment.
  - It can be placed at any code memory location and it is the responsibility. of the linker/locater to assign absolute address for this module.
- Each module can share variables and subroutines (functions) among them.
  - Keyword 'PUBLIC' and 'EXTRN' are used while accessing shared variables and subroutines.

Assembly Language Based Development (continued)

- **Library File Creation and Usage**
  - Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time.
    - Library files are generated with extension *'. lib'.*
  - When the linker processes a library, only those object modules in the library that are necessary to create the program are used.
  - Library file is some kind of source code hiding technique.
  - For example, *'LIB51'* from Keil 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.

Assembly Language Based Development (continued)

- **Linker and Locator**
  - 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 generates an absolute object module by extracting the object modules from the library, if any, and those *obj* files created by the assembler, which is generated by assembling the individual modules of a project.
  - It is the responsibility of the linker to link any external dependent variables or functions declared on various modules and resolve the external dependencies among the modules.
  - 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.

Assembly Language Based Development (continued)

- **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.
  - 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.
  - *'OH51'* 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

- Efficient Code Memory and Data Memory Usage (Memory Optimisation)
  - Since the developer is well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations.
  - This leads to less utilisation of code memory and efficient utilisation of data memory.
- High Performance
  - Optimised 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.

Advantages of Assembly Language Based Development (continued)

- 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 'hawkers' to reveal the technology behind 'Proprietary Products'.
  - Though most of the products employ code memory protection, if it may be possible to break the memory protection and read the code memory, it can easily be converted into assembly code using a dis-assembler program for the target machine.

Drawbacks of Assembly Language Based Development

- 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 organisation 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.
  - Also more lines of assembly code are required for performing an action which can be done with a single instruction in a high-level language like 'C'.

Drawbacks of Assembly Language Based Development (continued)

- Developer Dependency
  - Unlike high level languages, there is no common written rule for developing assembly language based applications.
  - 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 analyse this code, he/she also may not be able to understand what is done and why it is done.
    - Hence upgrading an assembly program or modifying it on a later stage is very difficult.

Drawbacks of Assembly Language Based Development (continued)

- Non-Portable
  - Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Intel x86 family of processors) and cannot be re-used for another target processors/controllers (Say ARM11 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

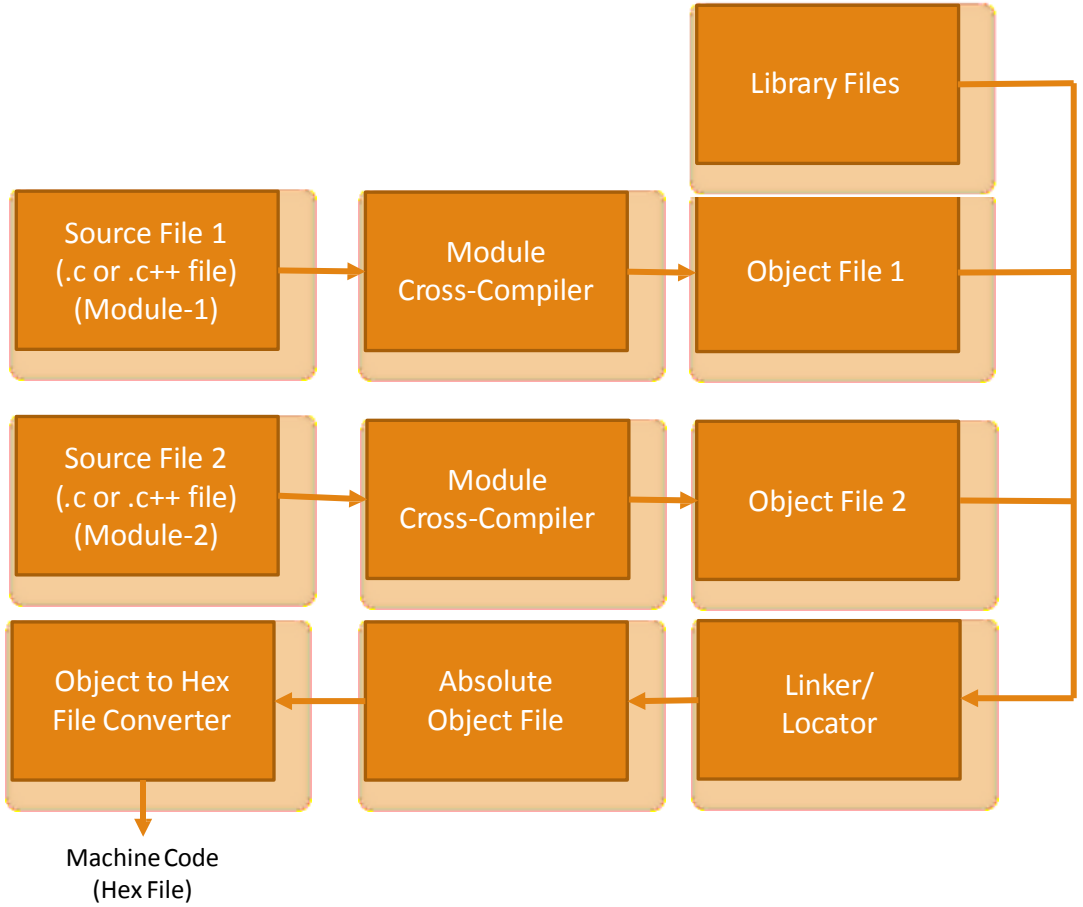
- Any high level language (like C, C++ or Java) with a supported cross-compiler for the target processor can be used for embedded firmware development.
- The most commonly used high level language for embedded firmware application development is '*C*'.
  - 'C' is well defined, easy to use high level 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.

High Level Language Based Development (continued)

- 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.
  - 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 figure.

High Level Language Based Development (continued)

High level language to machine language conversion process



High Level Language Based Development (continued)

- The program written in any of the high level languages 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 can be used for writing the program.
- Most of the high level languages support *modular programming* approach and hence we can have multiple source files called *modules* written in corresponding high level language.
- The source files corresponding to each module is represented by a file with corresponding language extension.

High Level Language Based Development (continued)

- Translation of high level source code to executable object code is done by a cross-compiler.
- Each high level language should have a cross-compiler for converting the high level source code into the target processor machine code.
  - C51 Cross-compiler from Keil software is an example for Cross-compiler used for 'C' language for the 8051 family of microcontroller.
- 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 organisation 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.
  - 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.

Advantages of High Level Language Based Development (continued)

- 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 code.
  - If the developer strictly adheres to the rules, the firmware will be 100% developer independent.

Advantages of High Level Language Based Development (continued)

- Portability
  - Target applications written in high level languages are converted to target processor/controller understandable format (machine codes) by a cross-compiler.
  - An application written in high level language for a particular target processor can 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 target processor/controller, provided, the cross-compiler has support for the processor/controller selected.
    - This makes applications written in high level language highly portable.
  - Little effort may be required in the existing code to replace the target processor specific files with new header files, register definitions with new ones, etc.
    - This is the major flexibility offered by high level language based design.

Limitations of High Level Language Based Development

- Poor Optimization by Cross-Compilers
  - Some cross-compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions.
  - Target images created by such compilers may be messy and non-optimised in terms of performance as well as code size.
    - For example, the task achieved by cross-compiler generated machine instructions from a high level language may be achieved through a lesser number of instructions if the same task is hand coded using target processor specific machine codes.
  - The time required to execute a task also increases with the number of instructions.
  - However modern cross-compilers are tending to adopt designs incorporating optimisation techniques for both code size and performance.

Limitations of High Level Language Based Development (continued)

- Not Suitable for Low Level Hardware
  - High level language based code snippets may not be efficient in accessing low level hardware where hardware access timing is critical (of the order of nano or micro seconds).
- High Investment Cost
  - 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:
  - Mixing Assembly Language with High Level Language
  - Mixing High Level Language with Assembly Language
  - Inline Assembly programming

Mixing Assembly Language with High Level Language

- Assembly routines are mixed with 'C' in situations where
  - the entire program is written in 'C' and the cross compiler in use do not have a built in support for implementing certain features like Interrupt Service Routine functions (ISR) *or*
  - if the programmer wants to take advantage of the speed and optimised 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 Assembly Language with High Level Language (continued)

- Mixing 'C' and Assembly is little complicated.
  - The programmer must be aware of how parameters are passed from the 'C' routine to Assembly and values are returned from assembly routine to 'C' and how 'Assembly routine' is invoked from the 'C' code.
- Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross-compiler dependent.

Mixing Assembly Language with High Level Language (continued)

- Consider an example Keil C51 cross compiler for 8051 controller.
- The steps for mixing assembly code with ‘C’ are:
  - Write a simple function in C that passes parameters and returns values the way you want your assembly routine to.
  - 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.
  - 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.
  - Rename the .SRC file to .A51 file.
  - 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 Assembly Language with High Level Language (continued)

- As an example consider the following sample code:

```
#pragma SRC
unsigned char my_assembly_func (unsigned int argument)
{
 return (argument + 1); // Insert dummy lines to access all args and
                        // retvals
}
```

- This C function on cross compilation generates the following assembly SRC file.
- The special compiler directive SRC generates the Assembly code corresponding to the 'C' function and each lines of the source code is converted to the corresponding Assembly instruction.

Mixing High Level Language 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 optimised code, optimal performance, efficient code memory utilisation 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'.

Mixing High Level Language with Assembly Language (continued)

- Most often the functions written in 'C' use parameter passing to the function and returns value/s to the calling functions.
- Parameters are passed to the function and values are returned from the function using CPU registers, stack memory and fixed memory.
- Its implementation is cross compiler dependent and it varies across cross compilers.

Mixing High Level Language with Assembly Language (continued)

- Consider an example for the Keil C51 cross compiler.
- C51 allows passing of a maximum of three arguments through general purpose registers R2 to R7.
- If the three arguments are *char* variables, they are passed to the function using registers R7, R6 and R5, respectively.
- If the parameters are *int* values, they are passed using register pairs (R7, R6), (R5, R4) and (R3, R2).
- If the number of arguments is greater than three, the first three arguments are passed through registers and rest is passed through fixed memory locations.

Mixing High Level Language with Assembly Language (continued)

- Return values are usually passed through general purpose registers.
- R7 is used for returning *char* value and register pair (R7, R6) is used for returning *int* value.
- The 'C' subroutine can be invoked from the assembly program using the subroutine call Assembly instruction.
- For example

```
LCALL _Cfunction
```

where *Cfunction* is a function written in 'C'

- The prefix _ informs the cross compiler that the parameters to the function are passed through registers.
- If the function is invoked without the _ prefix, it is understood that the parameters are passed through fixed memory locations.

Inline Assembly Programming

- Inline assembly is a technique for inserting target processor/controller specific Assembly instructions at any location of a source code written in high level language 'C'.
  - This avoids the delay in calling an assembly routine from a 'C' code.
- Special keywords are used to indicate that the start and end of Assembly instructions.
  - The keywords are cross-compiler specific.
  - C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.
  - For example:

```
#pragma asm
MOV A, #13H
#pragma endasm
```

Programming in Embedded C

- Whenever the conventional 'C' Language and its extensions are used for programming embedded systems, it is referred as ***'Embedded C'*** programming.
- Programming in 'Embedded C' is quite different from conventional Desktop application development using 'C' language for a particular OS platform.
- Desktop computers contain working memory in the range of Megabytes (Nowadays Giga bytes) and storage memory in the range of Giga bytes.
  - For a desktop application developer, the resources available are surplus in quantity and s/he can be very lavish in the usage of RAM and ROM and no restrictions are imposed at all.
  - This is not the case for embedded application developers.
- Almost all embedded systems are limited in both storage and working memory resources.
  - Embedded application developers should be aware of this fact and should develop applications in the best possible way which optimises the code memory and working memory usage as well as performance.
  - In other words, the hands of an embedded application developer are always tied up in the memory usage context.

‘C’ vs. ‘Embedded C’

- 'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support.
- 'C' offers a combination of the features of high level language and assembly and helps in hardware 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.
- 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 the supported cross-compiler for the particular Embedded 'C' language.
- A software program called 'Cross-compiler' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language).

Compiler vs. Cross-Compiler

- Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium).
- Here the operating system, the compiler program and the application making use of the source code run on the same target processor.
- The source code is converted to the target processor specific machine instructions.
- The development is platform specific (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 the software tools used in cross-platform development applications.
  - In cross-platform development, the compiler running on a particular target processor/OS converts the source code to 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.
  - 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.
- In embedded firmware application, whenever we use the term 'Compiler' it normally refers to the cross-compiler.

References

1. Shibu K V, ***"Introduction to Embedded Systems"***, Tata McGraw Hill, 2009.
2. Raj Kamal, ***"Embedded Systems: Architecture and Programming",*** Tata McGraw Hill, 2008.

MODULE – 5

RTOS and IDE for Embedded System Design

Operating System Basics

Operating System Basics

- The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.
- The OS manages the system resources and makes them available to the user applications/tasks on a need basis.
- The primary functions of an operating system are:
  - Make the system convenient to use
  - Organise and manage the system resources efficiently and correctly

Operating System Architecture

- Figure gives an insight into the basic components of an operating system and their interfaces with rest of the world.

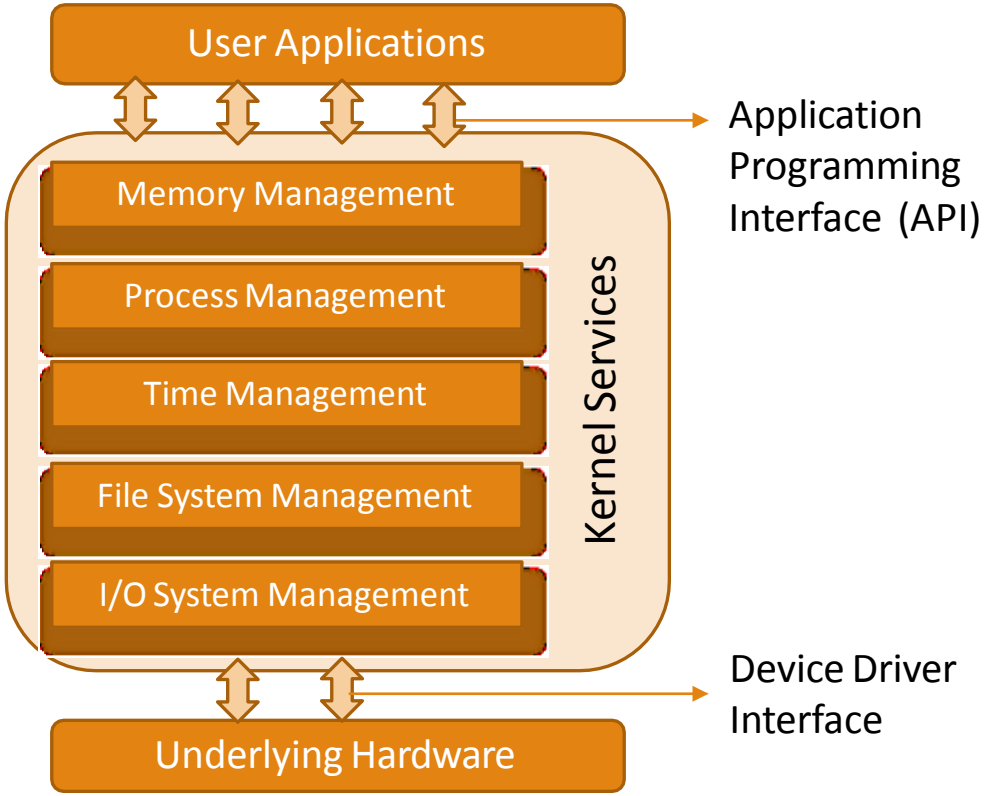


Fig: The Operating System Architecture

The Kernel

- The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services.
- Kernel acts as the abstraction layer between system resources and user applications.
- Kernel contains a set of system libraries and services.

The Kernel (continued)

- For a general purpose OS, the kernel contains different services for handling the following:
  - Process Management
  - Primary Memory Management
  - File System Management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection Systems
  - Interrupt Handler

Process Management

- Process management deals with managing the processes/tasks.
- Process management includes
  - Setting up the memory space for the process
  - Loading the process's code into the memory space
  - Allocating system resources
  - Scheduling and managing the execution of the process
  - Setting up and managing the Process Control Block (PCB)
  - Inter Process Communication and synchronisation
  - Process termination/deletion, etc.

Primary Memory Management

- The term *primary memory* refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored.
- The Memory Management Unit (MMU) of the kernel is responsible for
  - Keeping track of which part of the memory area is currently used by which process
  - Allocating and De-allocating memory space on a need basis (Dynamic memory allocation)

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.
- The 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 (e.g. Hard disk storage)
  - Providing automatic allocation of file space based on the amount of free space available
  - Providing a flexible naming convention for the files
- The various file system management operations are OS dependent.
  - For example, the kernel of Microsoft DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

I/O System (Device) Management

- Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.
- In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel.
- The kernel maintains a list of all the I/O devices of the system.
  - May be available in advance or updated dynamically as and when a new device is installed.
- The service Device Manager of the kernel is responsible for handling all I/O device related operations.
- The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service called device drivers.
- 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 since the main memory is volatile.
- In most of the systems, the secondary storage is kept in disks (Hard Disk).
- The secondary storage management service of kernel deals with
  - Disk storage allocation
  - Disk scheduling (Time interval at which the disk is activated to backup data)
  - Free Disk space management

Protection Systems

- Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions.
  - E.g. ‘Administrator’, ‘Standard’, ‘Restricted’ permissions in Windows XP.
- Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.
- In multiuser supported operating systems, one user may not be allowed to view or modify the whole or portions of another user's data or profile details.
- In addition, some application may not granted with permission to make use of some of the system resources.
  - This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler

- Kernel provides handler mechanism for all external/internal interrupts generated by the system.

Kernel Space and User Space

- The applications/services are classified into two categories:
  - User applications
  - Kernel applications
- Kernel Space is the memory space at which the kernel code is located
  - Kernel applications/services are kept in this contiguous area of primary (working) memory.
  - It is protected from the unauthorised access by user programs/applications.
- User Space is the memory area where user applications are loaded and executed.

Kernel Space and User Space (continued)

- The partitioning of memory into kernel and user space is purely OS dependent.
  - Some OS implement this kind of partitioning and protection whereas some OS do not segregate the kernel and user application code storage into two separate areas.
- In an operating system with virtual memory support, the user applications are loaded into its corresponding virtual memory space with *demand paging technique*.
  - The entire code for the user application need not be loaded to the main (primary) memory at once.
  - The user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis.
  - The act of loading the code into and out of the main memory is termed as *'Swapping'*.
  - Swapping happens between the main (primary) memory and secondary storage memory.

Monolithic Kernel and Microkernel

- The kernel forms the heart of an operating system.
- Different approaches are adopted for building an Operating System kernel.
- Based on the kernel design, kernels can be classified into
  - Monolithic Kernel
  - Microkernel

Monolithic Kernel

- In monolithic kernel architecture, all kernel services run in the kernel space.
- Here all kernel modules run within the same memory space under a single kernel thread.
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system.
- The major drawback of 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.

Monolithic Kernel (continued)

- The architecture representation of a monolithic kernel is given in the figure.

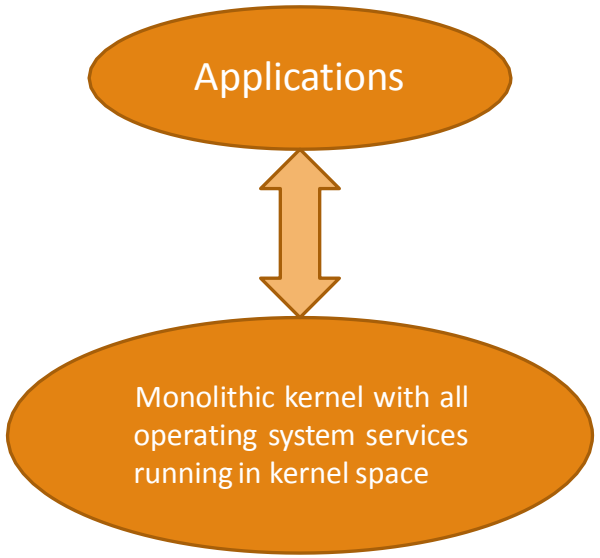


Fig: The Monolithic Kernel Model

Microkernel

- The microkernel design incorporates only the essential set of Operating System services into the kernel.
- The rest of the Operating System services are implemented in programs known as '*Servers*' which runs in user space.
- This provides a 'highly modular design and OS-neutral abstract to the kernel.
- Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel.
- Mach, QNX, Minix 3 kernels are examples for microkernel.

Microkernel (continued)

- The architecture representation of a microkernel is shown in the figure.

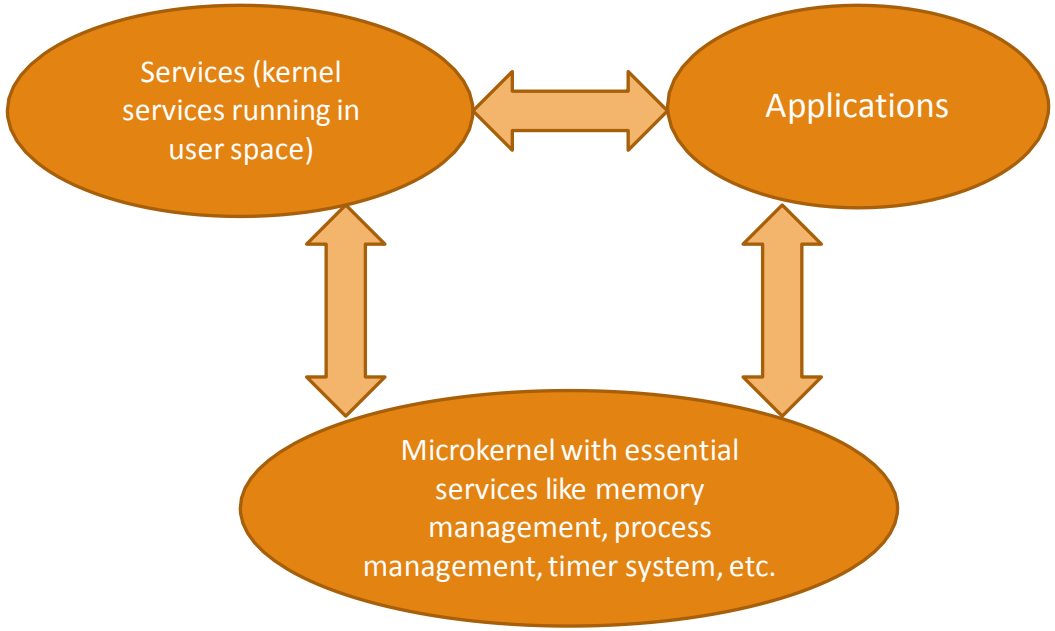


Fig: The Microkernel Model

Microkernel (continued)

- Microkernel based design approach offers the following benefits:
  - Robustness
    - If a problem is encountered in any of the services, which runs as *'Server'* application, the same can be reconfigured and re-started without the need for re-starting the entire OS.
    - Thus, this approach is highly useful for systems, which demands high *'availability'*.
    - Since the services which run as *'Servers'* are running on a different memory space, the chances of corruption of kernel services are ideally zero.
  - Configurability
    - Any service which runs as *'Server'* application can be changed without the need to restart the whole system.
    - This makes the system dynamically configurable.

Types of Operating Systems

Types of Operating Systems

- Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into different types.
  - General Purpose Operating System (GPOS)
  - Real-Time Operating System (RTOS)

General Purpose Operating System (GPOS)

- The operating systems which are deployed in general computing systems are referred as *General Purpose Operating Systems (GPOS)*.
- The kernel of such a GPOS is more generalised and it contains all kinds of services required for executing generic applications.
- General purpose operating systems are often quite non-deterministic in behaviour.
  - Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.
- GPOS are usually deployed in computing systems where deterministic behaviour is not an important criterion.
- Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed.
- Windows XP/MS-DOS etc. are examples for General Purpose Operating Systems.

Real-Time Operating System (RTOS)

- '*Real-Time*' implies deterministic timing behaviour.
  - Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time regardless the number of services.
- A Real-Time Operating System or RTOS implements policies and rules concerning time-critical allocation of a system's resources.
  - The RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
- Predictable performance is the hallmark of a well-designed RTOS.
- This is best achieved by the consistent application of policies and rules.
  - Policies guide the design of an RTOS.
  - Rules implement those policies and resolve policy conflicts.
- Windows CE, QNX, VxWorks, MicroC/OS-II, etc. are examples of Real-Time Operating Systems (RTOS).

The Real-Time Kernel

- The kernel of a Real-Time Operating System is referred as *Real-Time kernel*.
- The Real-Time kernel is highly specialised and it contains only the minimal set of services required for running the user applications/tasks.
- The basic functions of a Real-Time kernel are:
  - Task/Process Management
  - Task/Process Scheduling
  - Task/Process Synchronisation
  - Error/Exception Handling
  - Memory Management
  - Interrupt Handling
  - Time Management

The Real-Time Kernel (continued)

- 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
    - setting up a Task Control Block (TCB) for the task
    - task/process termination/deletion
  - A *Task Control Block (TCB)* is used for holding the information corresponding to a task.

The Real-Time Kernel (continued)

- 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***: 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***: 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 Real-Time Kernel (continued)

- The parameters and implementation of the TCB is kernel dependent.
- The TCB parameters vary across different kernels, based on the task management implementation.
- Task management service utilises the TCB of a task in the following way:
  - Creates a TCB for a task on creating a task
  - Delete/remove the TCB of a task when the task is terminated or deleted
  - Reads the TCB to get the state of a task
  - Update the TCB with updated parameters on need basis (e.g. on a context switch)
  - Modify the TCB to change the priority of the task dynamically

The Real-Time Kernel (continued)

- Task/Process Scheduling
  - Deals with sharing the CPU among various tasks/processes.
  - A kernel application called *'Scheduler'* handles the task scheduling.
  - *Scheduler* is nothing but an algorithm implementation, which performs the efficient and optimum scheduling of tasks to provide a deterministic behaviour.
- Task/Process Synchronisation
  - Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

The Real-Time Kernel (continued)

- Error/Exception Handling
  - Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks.
  - Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution, etc. are examples of errors/exceptions.
  - Errors/Exceptions can happen at the kernel level services or at task level.
    - Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception.
    - The OS kernel gives the information about the error in the form of a system call (API).
    - Watchdog timer is a mechanism for handling the timeouts for tasks.

The Real-Time Kernel (continued)

- Memory Management
  - RTOS makes use of '*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'*.
  - To achieve predictable timing and avoid the timing overheads, most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection.
    - RTOS kernels assume that the whole design is proven correct and protection is unnecessary.
    - Some commercial RTOS kernels allow memory protection as optional.

The Real-Time Kernel (continued)

- A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).
- In the *'block'* based memory allocation, 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.
- The *'block'* based memory allocation achieves deterministic behaviour with the trade of limited choice of memory chunk size and suboptimal memory usage.

The Real-Time Kernel (continued)

- Interrupt Handling
  - Deals with the handling of various types of interrupts.
  - Interrupts provide Real-Time behaviour to systems.
  - Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
  - Interrupts can be either *Synchronous* or *Asynchronous*.
  - **Synchronous interrupts:**
    - Occur in sync with the currently executing task.
    - Usually the software interrupts fall under this 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.

The Real-Time Kernel (continued)

- **Asynchronous interrupts:**
  - Occur at any point of execution of any task, and are not in sync with the currently executing task.
  - The interrupts generated by external devices (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, 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 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 interrupt can be enabled or disabled individually.
- Most of the RTOS kernel implements 'Nested Interrupts' architecture.
  - Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a high priority interrupt.

The Real-Time Kernel (continued)

- 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'* and 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 Real-Time Kernel (continued)

- 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

$$2^{32} \times 10^{-6}\ seconds = \frac{2^{32} \times 10^{-6}}{24 \times 60 \times 60}\ Days = \sim 0.0497\ Days = 1.19\ Hours$$

- If the 'Timer tick' interval is 1 millisecond, the system time register will reset in

$$2^{32} \times 10^{-3}\ seconds = \frac{2^{32} \times 10^{-3}}{24 \times 60 \times 60}\ Days = 49.7\ Days = \sim 50\ Days$$

The Real-Time Kernel (continued)

- The *'Timer tick'* interrupt is handled by the 'Timer Interrupt' handler of kernel.
- The *'Timer tick'* interrupt can be utilised 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 preempted by the 'Timer Interrupt' task.

Hard Real-Time

- Real-Time Operating Systems that strictly adhere to the timing constraints for a task are referred as *'Hard Real-Time'* systems.
  - They 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 loss and irrecoverable damages to the system/users.
- Hard Real-Time systems emphasise the principle '*A late answer is a wrong answer*'.
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems.
  - Any delay in the deployment of the air bags makes the life of the passengers under threat.

Hard Real-Time (continued)

- Hard Real-Time Systems does not implement the virtual memory model for handling the memory.
  - This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory.
- Most of the Hard Real-Time Systems are automatic and does not contain a *Human in the Loop (HITL)*.
  - The presence of *human in the loop* for tasks introduces unexpected delays in the task execution.

Soft Real-Time

- Real-Time Operating Systems that do not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as *'Soft Real-Time'* systems.
- Missing deadlines for tasks are acceptable for a Soft Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
- A Soft Real-Time system emphasises the principle *'A late answer is an acceptable answer, but it could have done bit faster'*.
- Soft Real-Time systems most often have a *human in the loop (HITL)*.
- Automatic Teller Machine (ATM) is a typical example for Soft-Real-Time System.
  - If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
- An audio-video playback system is another example for Soft Real-Time system.
  - No potential damage arises if a sample comes late by fraction of a second, for playback.

Tasks, Process and Threads

Task

- 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.
- Multiple instances of the same program can execute simultaneously.
- 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.
- A process is sequential in execution.

The Structure of a Process

- The concept of *'Process'* leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation of the CPU and other system resources.
- Concurrent execution is achieved through the sharing of CPU among the processes.
- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process.
- This can be visualised as shown in the figure.

The Structure of a Process (continued)

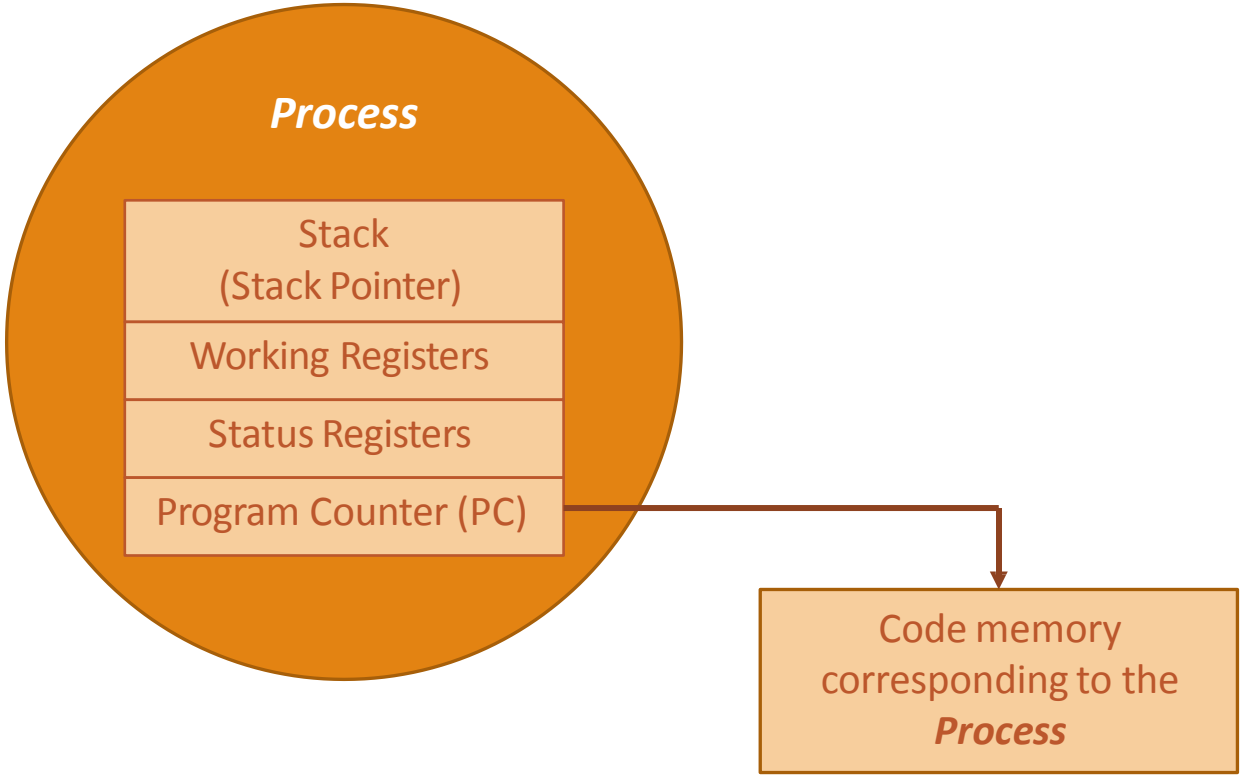


Fig: Structure of a Process

The Structure of a Process (continued)

- 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 the program counter register becomes mapped to the physical registers of the CPU.

The Structure of a Process (continued)

- From a memory perspective, the memory occupied by the process is segregated into three regions as shown in the figure:
  - Stack memory - holds all temporary data such as variables local to the process
  - Data memory - holds all global data for the process
  - Code memory - contains the program code (instructions) corresponding to the process

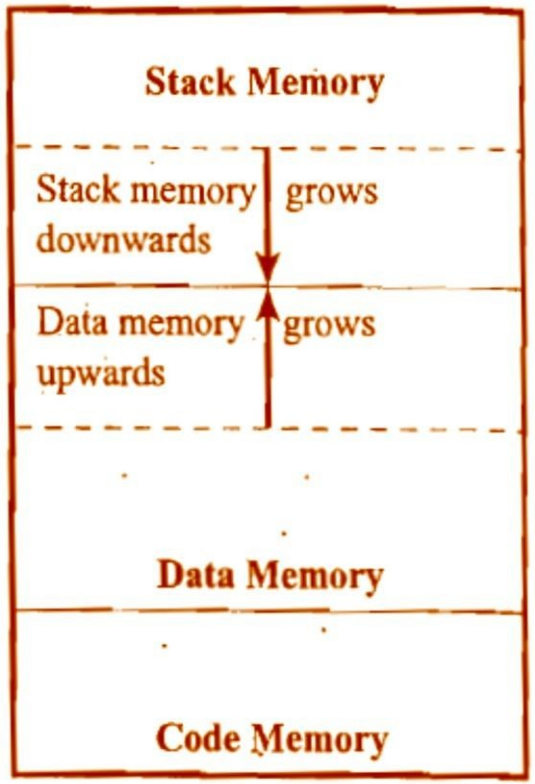


Fig: Memory Organisation of a Process

Process States and State Transition

- 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'*.
- Figure represents the various states and state transitions associated with a process.

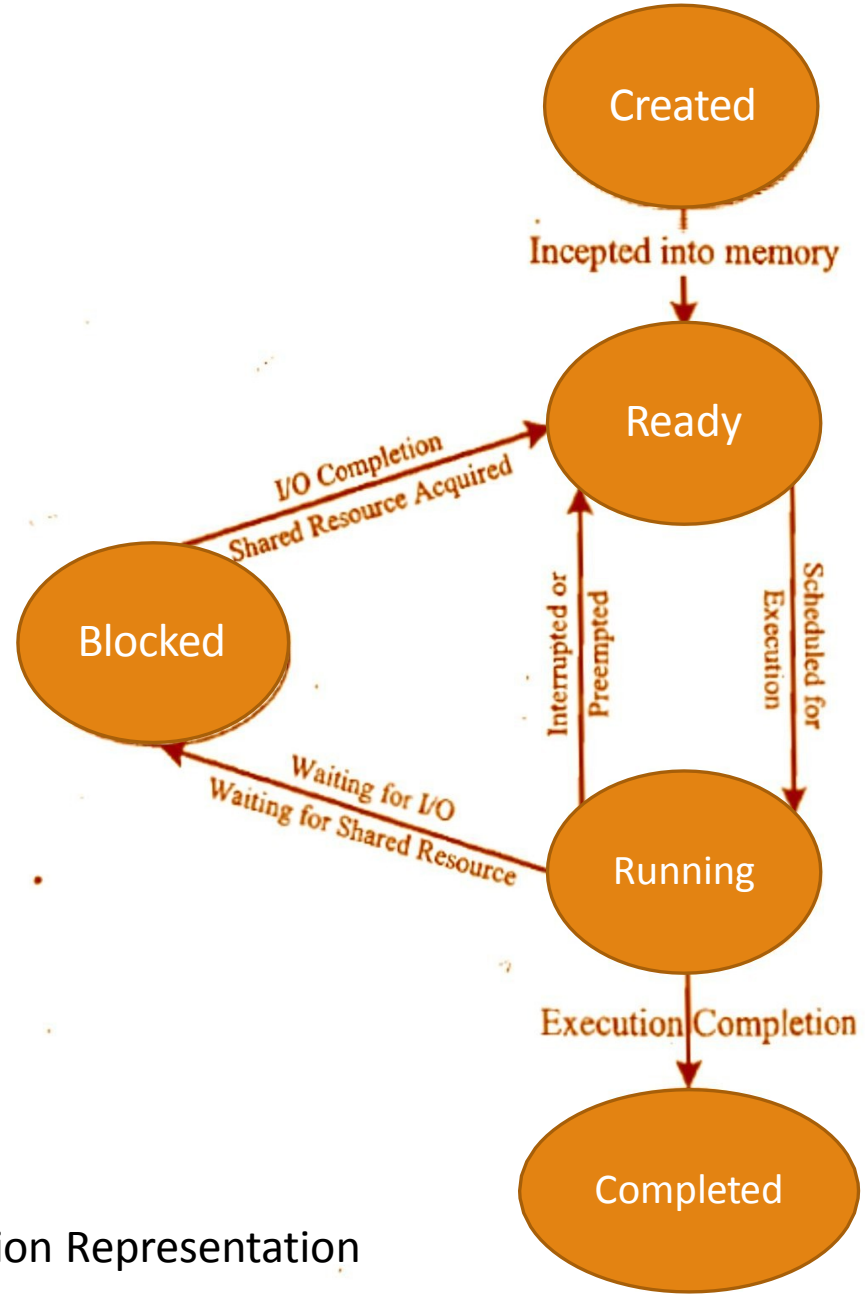


Fig: Process States and State Transition Representation

Process States and State Transition (continued)

- The state at which a process is being created is referred as 'Created State’.
  - The Operating System recognises a process in the 'Created State' but no resources are allocated to the process.
- The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as 'Ready State’.
  - At this stage, the process is placed in the 'Ready list' queue maintained by the OS.
- The state where in the source code instructions corresponding to the process is being executed is called 'Running State’.
  - Running State is the state at which the process execution happens.

Process States and State Transition (continued)

- '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 be 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
- A state where the process completes its execution is known as 'Completed State’.

Process Management

- Process management deals with
  - creation of a process
  - setting up the memory space for the process
  - loading the process's code into the memory space
  - allocating system resources
  - setting up a Process Control Block (PCB) for the process
  - process termination/deletion

Threads

- A *thread* is the primitive that can execute code.
- A *thread* is a single sequential flow of control within a process.
- *'Thread'* is also known as light-weight process.
- 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 figure.

Threads (continued)

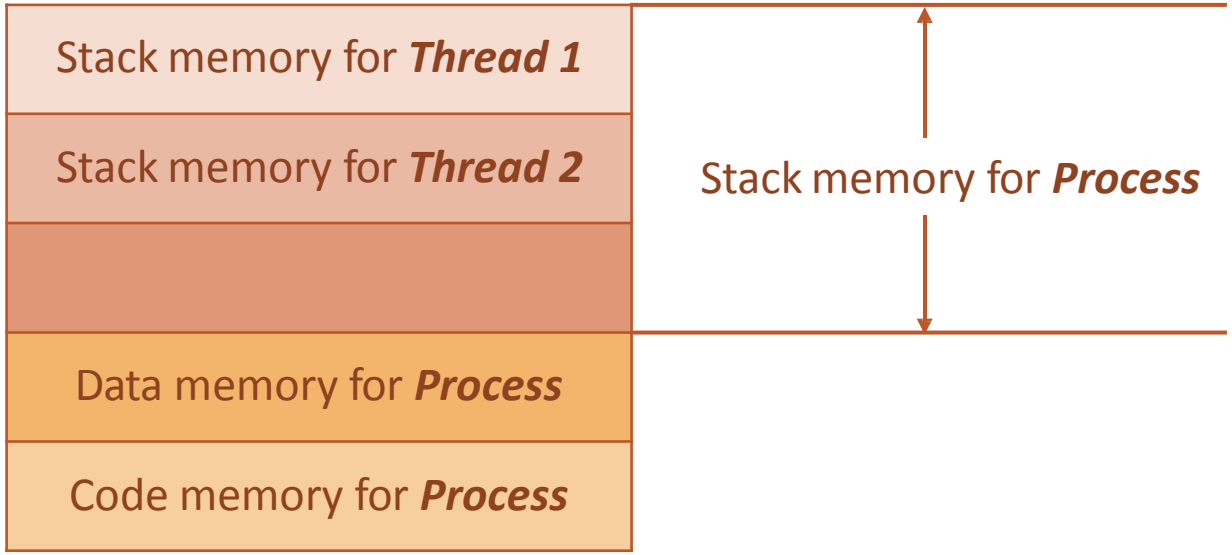


Fig: Memory organisation of a ***Process*** and its associated ***Threads***

The Concept of Multithreading

- A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc.
- If all the subfunctions of a task are executed in sequence, the CPU utilisation may not be efficient.
  - For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state.

The Concept of Multithreading (continued)

- Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different subfunctionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution.
  - This leads to more speedy execution of the process and the efficient utilisation of the processor time and resources.
- If 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 better visualised with the thread-process diagram, shown in the figure.

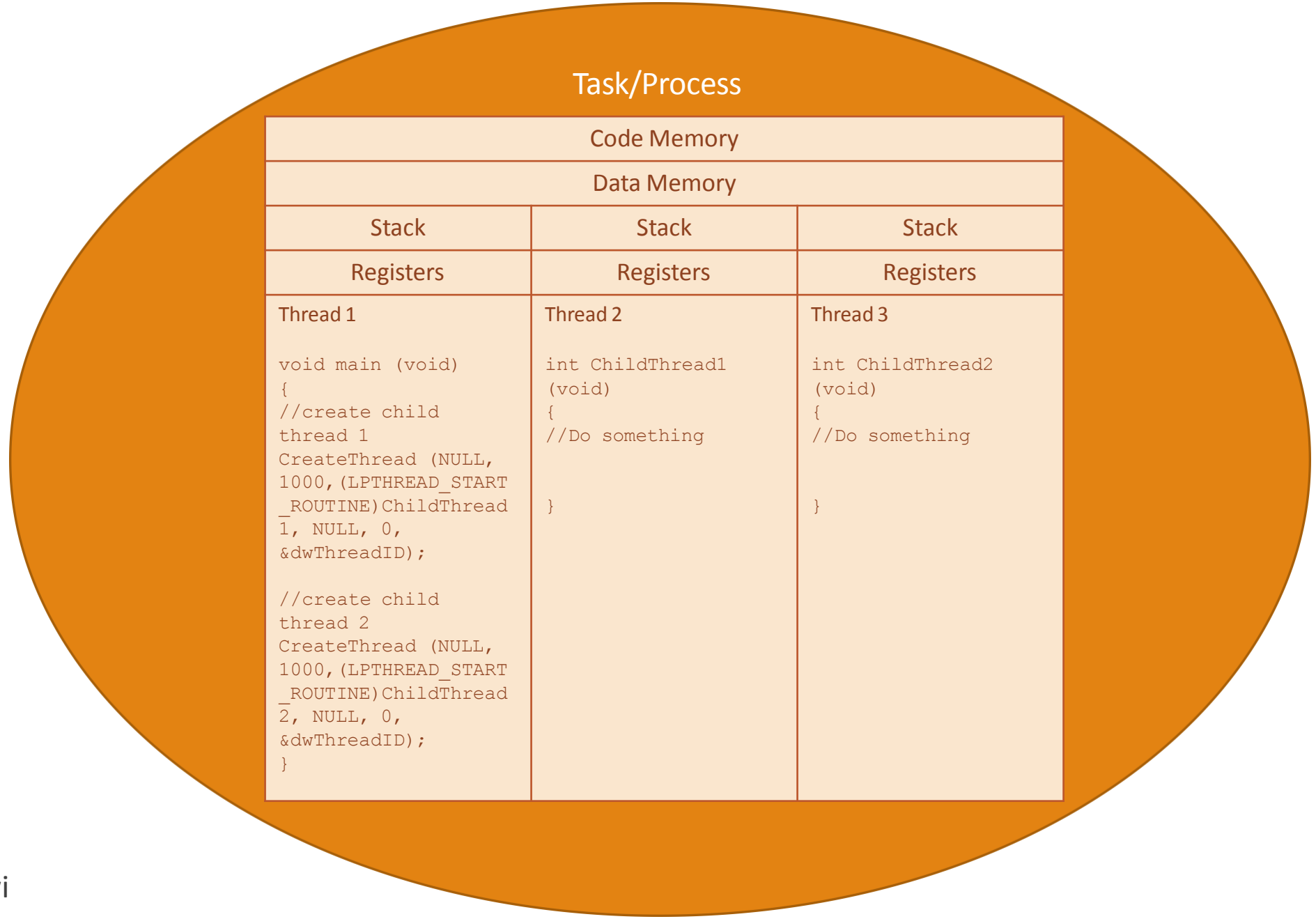


Fig: Process wi

The Concept of Multithreading (continued)

- Use of multiple threads to execute a process brings the following advantages:
  - **Better memory utilisation**
    - 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.
  - **Speedy execution of the process**
    - Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing.
  - **Efficient CPU utilisation**
    - The CPU is engaged all time.

Thread Standards

- Thread standards deal with the different standards available for thread creation and management.
  - These standards are utilised 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
  - Win32 Threads
  - Java Threads

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.

POSIX Threads (continued)

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t, *attribute,
void * (*start_function) (void *), void *arguments);
```

- This primitive creates a new thread for running the function *start_function*.
- Here *pthread_t* is the handle to the newly created thread and *pthread_attr_t* is the data type for holding the thread attributes.
- *'start_function'* is the function the thread is going to execute and *arguments* is the arguments for *'start_function'*.
- On successful creation of a *Pthread*, *pthread_create()* associates the Thread Control Block (TCB) corresponding to the newly created thread to the variable of type *pthread_t* (*new_thread ID* in our example).

POSIX Threads (continued)

```
int pthread_join(pthread_t new_thread, void * *thread_status);
```

- This primitive blocks the current thread and waits until the completion of the thread pointed by it (*new_thread* in this example).
- All the POSIX 'thread calls' returns an integer.
  - A return value of zero indicates the success of the call.

POSIX Threads - Example

- Write a multithreaded application to print 'Hello I'm in main thread" from the main thread and "Hello I'm in new thread" 5 times each, using the *pthread_create()* and *pthread_join()* POSIX primitives.

```
//Assumes the application is running on an OS where POSIX library is available
#include<pthread.h>
#include<stdlib.h>
#include<stdio.h>
//*************************************************************************
//New thread function for printing “Hello I’m in new thread”
void *new_thread(void *thread_args)
{
 int i,j;
 for(j=0; j<5; j++)
 {
  printf(“Hello I’m in new thread\n”);
  for(i=0; i<10000; i++);              //Wait for some time. Do nothing.
 }
 return NULL;
}
```

```
//*************************************************************************
//Start of main thread
int main (void)
{
 int i,j;
 pthread_t tcb;
//Create the new thread for executing new_thread function
 if (pthread_create(&tcb, NULL, new_thread, NULL))
 {
  //New thread creation failed
  printf("Error in creating new thread\n");
  return -1;
 }
 for(j=0; j<5; j++)
 {
  printf("Hello I'm in main thread\n");
  for(i=0; i<10000; i++);              //Wait for some time. Do nothing.
 }
 if (pthread_join(tcb, NULL))
 {
  //Thread join failed
  printf("Error in Thread join\n");
  return -1;
 }
return 1;
}
```

POSIX Threads (continued)

- The termination of a thread can happen in different ways:
  - Natural termination:
    - The thread completes its execution and returns to the main thread through a simple *return* or by executing the *pthread_exit()* call.
  - Forced termination:
    - This can be achieved by the call *pthread_cancel()* or through the termination of the main thread with *exit* or *exec* functions.
    - *pthread_cancel()* call is used by a thread to terminate another thread.

Thread Pre-emption

- Thread pre-emption is the act of pre-empting the currently running thread.
  - It means, stopping the currently running thread temporarily.
- Thread pre-emption is performed for sharing the CPU time among all the threads.
- The execution switching among threads is known as 'Thread context switching’.
- Thread context switching is dependent on the Operating system's scheduler and the type of the thread.

Types of Threads

- User Level Threads
  - User level threads do not have kernel/Operating System support and they exist solely in the running process.
  - Even if a process contains multiple user level threads, the OS treats 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.
  - In summary, user level threads of a process are non-preemptive at thread level from OS perspective.
  - The execution switching (thread context switching) happens 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.

Types of Threads (continued)

- Kernel Level Threads
  - 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.
  - In summary, kernel level threads are pre-emptive.
  - 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.

Thread Binding Models

- There are many ways for binding user level threads with system/kernel level threads.
- ***Many-to-One Model***
  - Here, many user level threads are mapped to a single kernel thread.
  - In this model, 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.
  - The *'PThread' example* is an illustrative example for application with Many-to-One thread model.

Thread Binding Models (continued)

- ***One-to-One Model***
  - Here, each user level thread is bonded to a kernel/system level thread.
  - Windows XP/NT/2000 and Linux threads are examples for One-to-One thread models.
  - The modified *'PThread'* example is an illustrative example for application with One-to-One thread model.
- ***Many-to-Many Model***
  - In this model, many user level threads are allowed to be mapped to many kernel threads.
  - Windows NT/2000 with *ThreadFibre* package is an example for this.

Thread vs. 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 heal memory. It shares the data memory and heap memory with other threads of the same process.	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 thread (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 (share the total stack memory of the process).
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 lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also die.

Task Scheduling

Task Scheduling

- Multitasking involves the execution switching among the different tasks.
- There should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time.
- Determining which task/process is to be executed at a given point of time is known as *task/process scheduling*.
- Scheduling policies forms the guidelines for determining which task is to be executed when.
- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service.
- The kernel service/application, which implements the scheduling algorithm, is known as *'Scheduler'*.

Task Scheduling

- Based on the scheduling algorithm used, scheduling can be classified into:
  - Non-preemptive Scheduling
    - The currently executing task/process is allowed to run until it terminates or enters the '*Wait*' state waiting for an I/O or system resource.
  - Preemptive Scheduling
    - The currently executing task/process is preempted (stopped temporarily) and another task from the Ready queue is selected for execution.

Task Scheduling (continued)

- The process scheduling decision may take place when a process switches its state to
  1. '*Ready*' state from '*Running*' state
  2. '*Blocked/Wait*' state from '*Running*' state
  3. '*Ready*' state from '*Blocked/Wait*' state
  4. '*Completed*' state
- A process switches to 'Ready' state from the 'Running' state when it is preempted.
  - Hence, the type of scheduling in scenario 1 is pre-emptive.

Task Scheduling (continued)

- When a high priority process in the 'Blocked/Wait' state completes its I/O and switches to the 'Ready' state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive.
  - This is indicated by scenario 3.
- In preemptive/non-preemptive multitasking, the process relinquishes the CPU when it enters the ‘Blocked/Wait' state or the 'Completed' state and switching of the CPU happens at this stage.
- Scheduling under scenario 2 can be either preemptive or non-preemptive.
- Scheduling under scenario 4 can be preemptive, non-preemptive or co-operative.

Task Scheduling (continued)

- The selection of a scheduling criterion/algorithm should consider the following factors:
  - ***CPU Utilisation:***
    - The scheduling algorithm should always make the CPU utilisation high.
    - CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.
  - ***Throughput:***
    - This gives an indication of the number of processes executed per unit of time.
    - The throughput for a good scheduler should always be higher.
  - ***Turnaround Time (TAT):***
    - It is the amount of time taken by a process for completing its execution.
    - It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution.
    - The turnaround time should be minimal for a good scheduling algorithm.

Task Scheduling (continued)

- ***Waiting Time:***
  - It is the amount of time spent by a process in the 'Ready' queue waiting to get the CPU time for execution.
  - The waiting time should be minimal for a good scheduling algorithm.
- ***Response Time:***
  - It is the time elapsed between the submission of a process and the first response.
  - For a good scheduling algorithm, the response time should be as least as possible.

To summarise, a good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

Task Scheduling (continued)

- The various queues maintained by OS in association with CPU scheduling are:
  - ***Job Queue:***
    - Contains all the processes in the system.
  - ***Ready Queue:***
    - Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution.
    - The Ready queue is empty when there is no process ready for running.
  - ***Device Queue:***
    - Contains the set of processes, which are waiting for an I/O device.

Preemptive Scheduling

- In preemptive scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution.
  - Every task in the 'Ready' queue gets a chance to execute.
- When to pre-empt a task and which task is to be picked up from the 'Ready' queue for execution after preempting the current task is purely dependent on the scheduling algorithm.
- A task which is preempted by the scheduler is moved to the 'Ready' queue.
- The act of moving a 'Running' process/task into the 'Ready' queue by the scheduler, without the processes requesting for it is known as ‘Preemption’

Preemptive Scheduling Techniques

- Preemptive scheduling can be implemented in different approaches.
  - Time-based preemption
  - Priority-based preemption
- The various types of preemptive scheduling adopted in task/process scheduling are:
  - Preemptive Shortest Job First (SJF)/Shortest Remaining Time (SRT) Scheduling
  - Round Robin (RR) Scheduling
  - Priority Based Scheduling

Preemptive Shortest Job First (SJF)/Shortest Remaining Time (SRT) Scheduling

- In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.
- The preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process.
- If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution.
- Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.
  - Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling .

Preemptive SJF/SRT Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enter the ready queue together. A new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the SRT scheduling.

Preemptive SJF/SRT Scheduling – Example (continued)

Process ID	Remaining Time
P1	10 ms
P2	5 ms
P3	7 ms

'Ready' queue at 0 ms

P2 is scheduled

Process ID	Remaining Time
P1	10 ms
P2	3 ms
P3	7 ms
P4	2 ms

'Ready' queue at 2 ms

P2 is preempted
P4 is scheduled

Process ID	Remaining Time
P1	10 ms
P2	3 ms
P3	7 ms

'Ready' queue at 4 ms

P4 is completed
P2 is scheduled

Process ID	Remaining Time
P1	10 ms
P3	7 ms

'Ready' queue at 7 ms

P2 is completed
P3 is scheduled

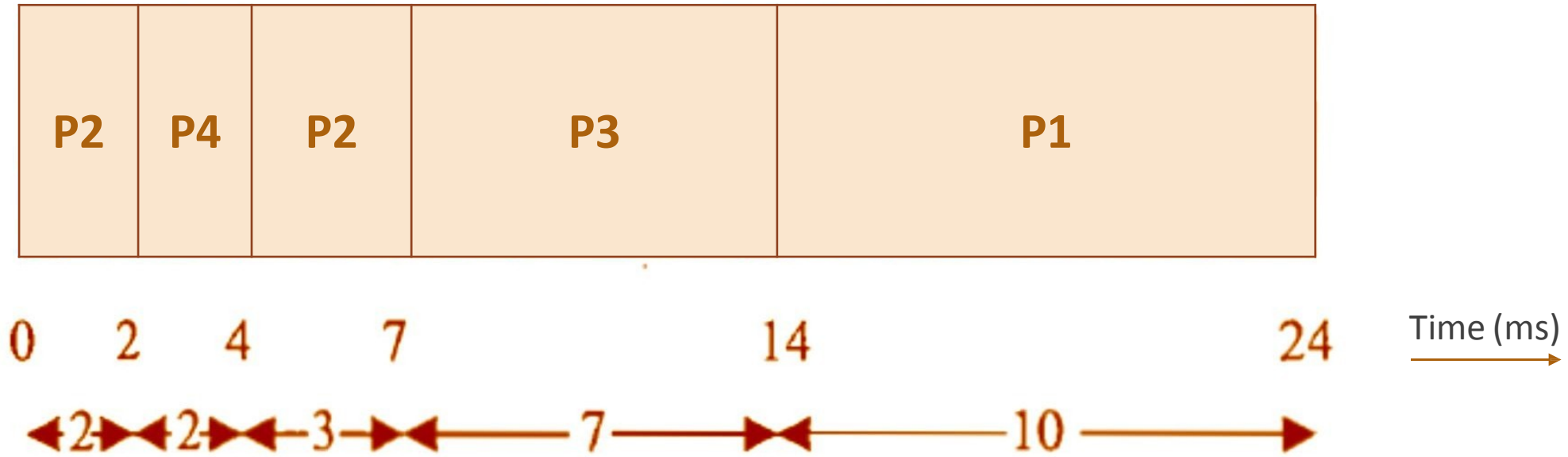
Process ID	Remaining Time
P1	10 ms

'Ready' queue at 14 ms

P3 is completed
P1 is scheduled

Preemptive SJF/SRT Scheduling – Example (continued)

- The execution sequence can be written as below:



Preemptive SJF/SRT Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P2 = 0\ ms + (4\ -\ 2)\ ms = 2\ ms$
  - $Waiting\ time\ for\ P4 = 0\ ms$
  - $Waiting\ time\ for\ P3 = 7\ ms$
  - $Waiting\ time\ for\ P1 = 14\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$
$$= \frac{2+0+7+14}{4}\ ms = \frac{23}{4} ms$$
$$= 5.75\ ms$$

Preemptive SJF/SRT Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 2\ ms + 5\ ms = 7\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P4 = 0\ ms + 2\ ms = 2\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 7\ ms + 7\ ms = 14\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 14\ ms + 10\ ms = 24\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{7+2+14+24}{4}\ ms = \frac{47}{4} ms$$

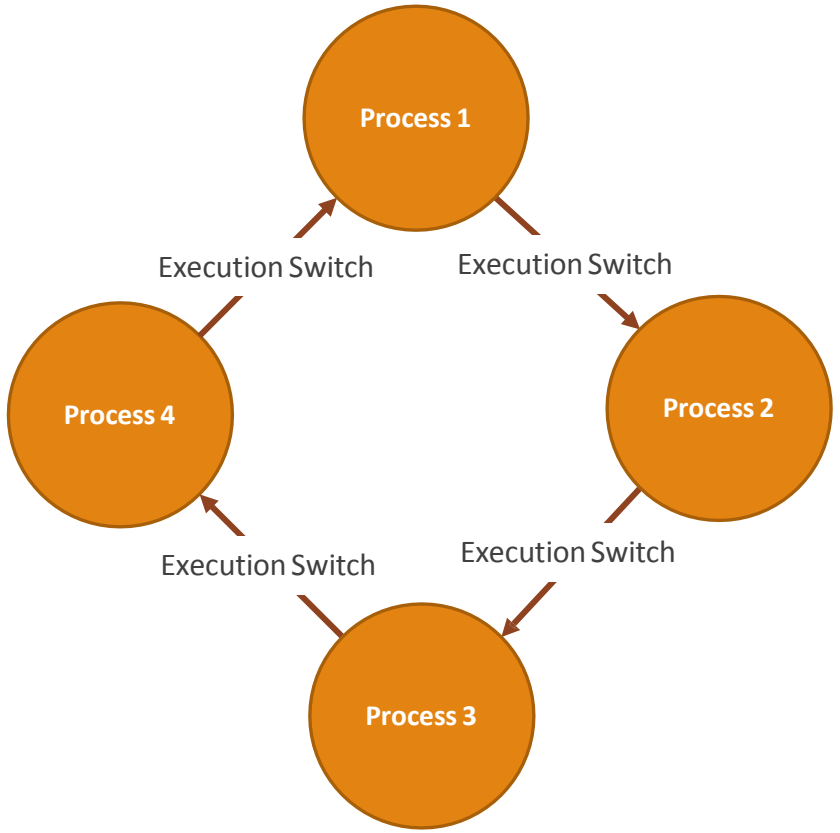
$$= 11.75\ ms$$

Round Robin (RR) Scheduling

- In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot.
  - 'Round Robin' brings the message "Equal chance to all".
- The execution starts with picking up the first process in the 'Ready' queue.
- It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution.
- This is repeated for all the processes in the 'Ready' queue.
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- The sequence is repeated.

Round Robin (RR) Scheduling (continued)

- The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the first process for execution and moves to the next till the end of the queue and then comes back to the beginning of the queue to pick up the first process.



Round Robin (RR) Scheduling (continued)

- The time slice is provided by the timer tick feature of the time management unit of the OS kernel.
- Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds.
- Round Robin scheduling ensures that every process gets a fixed amount of-CPU time for execution.
- When the process gets its fixed time for execution is determined by the First Come First Serve (FCFS) policy.
- If a process terminates before the elapse of the time slice, the process releases the CPU voluntarily and the next process in the queue is scheduled for execution by the scheduler.

Round Robin (RR) Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enter the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms.

Process ID	Remaining Time
P1	6 ms
P2	4 ms
P3	2 ms

'Ready' queue at 0 ms

P1 is scheduled

Process ID	Remaining Time
P2	4 ms
P3	2 ms
P1	4 ms

'Ready' queue at 2 ms

P1 is preempted
P2 is scheduled

Process ID	Remaining Time
P3	2 ms
P1	4 ms
P2	2 ms

'Ready' queue at 4 ms

P2 is preempted
P3 is scheduled

Process ID	Remaining Time
P1	4 ms
P2	2 ms

'Ready' queue at 6 ms

P3 is completed
P1 is scheduled

Process ID	Remaining Time
P2	2 ms
P1	2 ms

'Ready' queue at 8 ms

P1 is preempted
P2 is scheduled

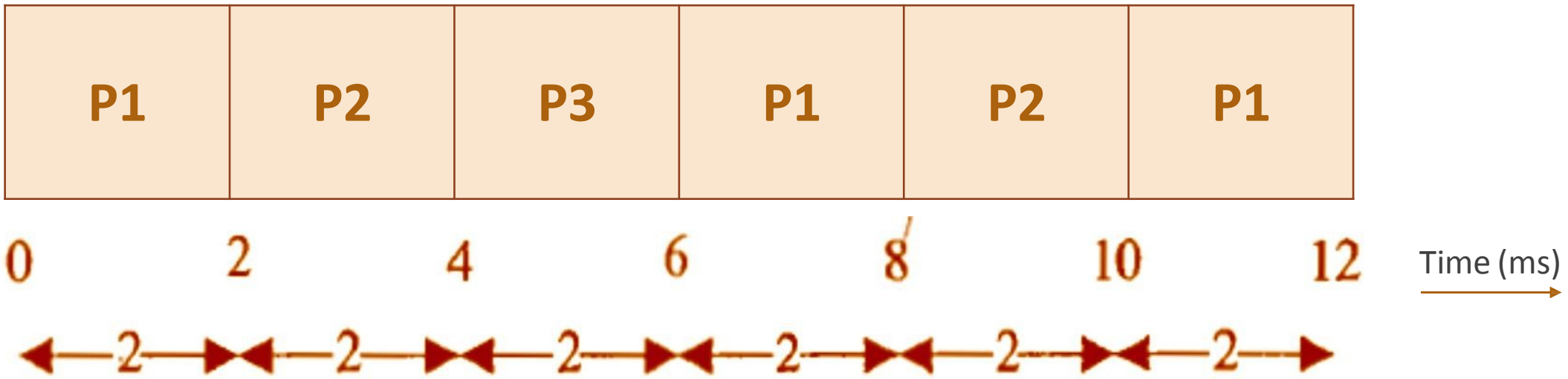
Process ID	Remaining Time
P1	2 ms

'Ready' queue at 10 ms

P2 is completed
P1 is scheduled

Round Robin (RR) Scheduling– Example (continued)

- The execution sequence can be written as below:



Round Robin (RR) Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P1 = 0\ ms + (6\ -\ 2) + (10 - 8)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P2 = (2 - 0) + (8 - 4)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P3 = (4 - 0) = 4\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$
$$= \frac{6+6+4}{3}\ ms = \frac{16}{3}\ ms$$
$$= 5.33\ ms$$

Round Robin (RR) Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 6\ ms + 6\ ms = 12\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 6\ ms + 4\ ms = 10\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 4\ ms + 2\ ms = 6\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{12+10+6}{3}\ ms = \frac{28}{3}\ ms$$

$$= 9.33\ ms$$

Priority Based Scheduling

- The Priority Based Preemptive Scheduling ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the ‘Ready’ queue.
  - Any high priority process entering the 'Ready' queue is immediately scheduled for execution.
- The priority of a task/process can be indicated through various mechanisms.
  - While creating the process/task, the priority can be assigned to it.
  - The priority number associated with a task/process is the direct indication of its priority.
  - The priority number 0 indicates the highest priority.
  - This convention need not be universal and it depends on the kernel level implementation of the priority structure.
- Whenever a new process enters the ‘Ready’ queue, the scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

Priority Based Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0 – highest priority, 3 - lowest priority) respectively enter the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Priority	Process ID	Remaining Time
1	P1	10 ms
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 0 ms

P1 is scheduled

Priority	Process ID	Remaining Time
1	P1	5 ms
3	P2	5 ms
2	P3	7 ms
0	P4	6 ms

'Ready' queue at 5 ms

P1 is preempted
P4 is scheduled

Priority	Process ID	Remaining Time
1	P1	5 ms
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 11 ms

P4 is completed
P1 is scheduled

Priority	Process ID	Remaining Time
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 16 ms

P1 is completed
P3 is scheduled

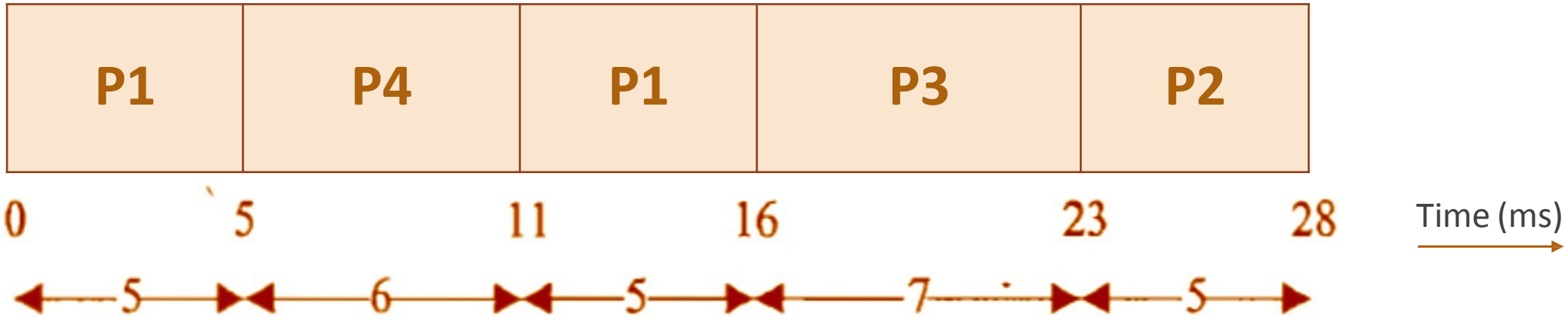
Priority	Process ID	Remaining Time
3	P2	5 ms

'Ready' queue at 23 ms

P3 is completed
P2 is scheduled

Priority Based Scheduling– Example (continued)

- The execution sequence can be written as below:



Priority Based Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P1 = 0\ ms + (11 - 5)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P4 = 0\ ms$
  - $Waiting\ time\ for\ P3 = 16\ ms$
  - $Waiting\ time\ for\ P2 = 23\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$

$$= \frac{6+0+16+23}{4}\ ms = \frac{45}{4}\ ms$$

$$= 11.25\ ms$$

Priority Based Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 6\ ms + 10\ ms = 16\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P4 = 0\ ms + 6\ ms = 6\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 16\ ms + 7\ ms = 23\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 23\ ms + 5\ ms = 28\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{16+6+23+28}{4}\ ms = \frac{73}{4}\ ms$$

$$= 18.25\ ms$$

Task Communication

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 running on an OS are classified as
  - **Co-operating Processes:**
    - 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.

Task Communication (continued)

- Co-operating processes exchanges information and communicate through the following methods:
  - **Co-operation through Sharing:**
    - The co-operating process exchange data through some shared resources.
  - **Co-operation through Communication:**
    - No data is shared between the processes.
    - But they communicate for synchronisation.

Task Communication (continued)

- The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC).
  - Inter Process Communication is essential for process co-ordination.
- The various types of Inter Process Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent.
- Some of the important IPC mechanisms adopted by various kernels are:
  - ***Shared Memory***
    - Pipes and Memory Mapped Objects
  - ***Message Passing***
    - Message Queue, Mailbox and Signalling
  - ***Remote Procedure Call and Sockets***

Shared Memory

- Processes share some area of the memory to communicate among them.
- Information to be communicated by the process is written to the shared memory area.
- Other processes which require this information can read the same from the shared memory area.

Fig.: Concept of Shared Memory

- Different mechanisms are adopted by different kernels for implementing the concept of shared memory:
  - Pipes
  - Memory Mapped Objects

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 a *pipe server* and a process which connects to a pipe is known as *pipe client*.
- A pipe can be considered as a conduit for information flow and has two conceptual ends.
- It can be unidirectional, allowing information flow in one direction or bidirectional allowing bidirectional information flow.

Pipes (continued)

- 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 bidirectional pipe allows both reading and writing at one end.
- The figure shows a unidirectional pipe.

Fig.: Concept of Pipe for IPC

Pipes (continued)

- The implementation of 'Pipes' is OS dependent.
- Microsoft Windows supports two types of 'Pipes' for Inter Process Communication:
  - **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 and 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.

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.

Message Passing

- Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread Communication.
- The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of information/data is passed through message passing.
  - Also, message passing is relatively fast and free from the synchronisation overheads compared to shared memory.
- Based on the message passing operation between the processes, message passing is classified into:
  - Message Queue
  - Mailbox
  - Signalling

Message Queue

- 'Message queue’ is a First-In-First-Out (FIFO) queue which stores the messages temporarily in a system defined memory object to pass it to the desired process.
- Usually the process which wants to talk to another process posts the message to a message queue.
- Messages are sent and received through *send* and *receive* methods.
  - *send (Name of the process to which the message is to be sent, message)*
  - *receive (Name of the process from which the message is to be received, message)*
- The implementation of the message queue, send and receive methods are OS kernel dependent.

Message Queue (continued)

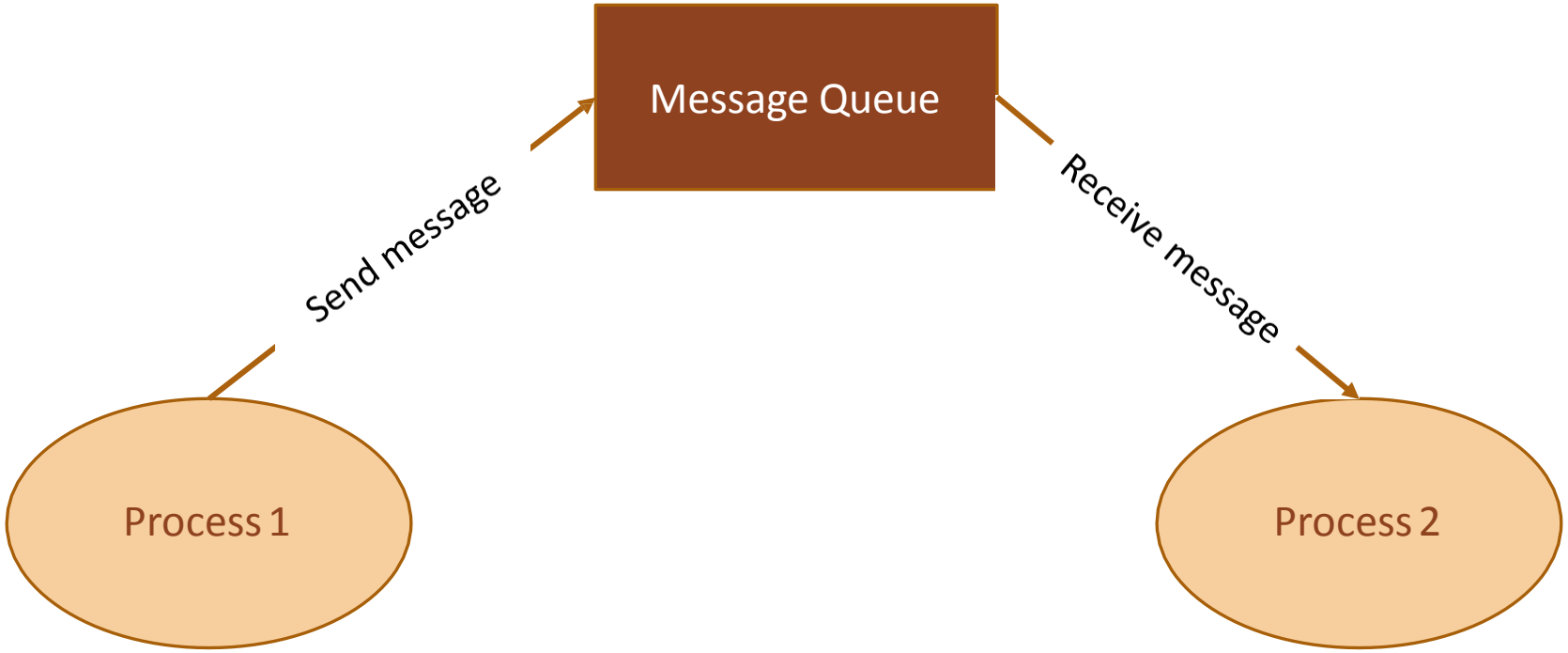


Fig.: Concept of Message Queue based indirect messaging for IPC

Message Queue (continued)

- The Windows XP OS kernel maintains a single system message queue and one process/thread specific message queue.
- A thread which wants to communicate with another thread posts the message to the system message queue.
- The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread.
- The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread.
  - In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted.
  - In synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.
    - The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

Mailbox

- Mailbox is an alternate form of ‘Message queue’ and it is used in RTOS for IPC usually for one way messaging.
- The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages.
- The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox.
- The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients’.
  - The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox.
  - The clients read the message from the mailbox on receiving the notification.

Mailbox (continued)

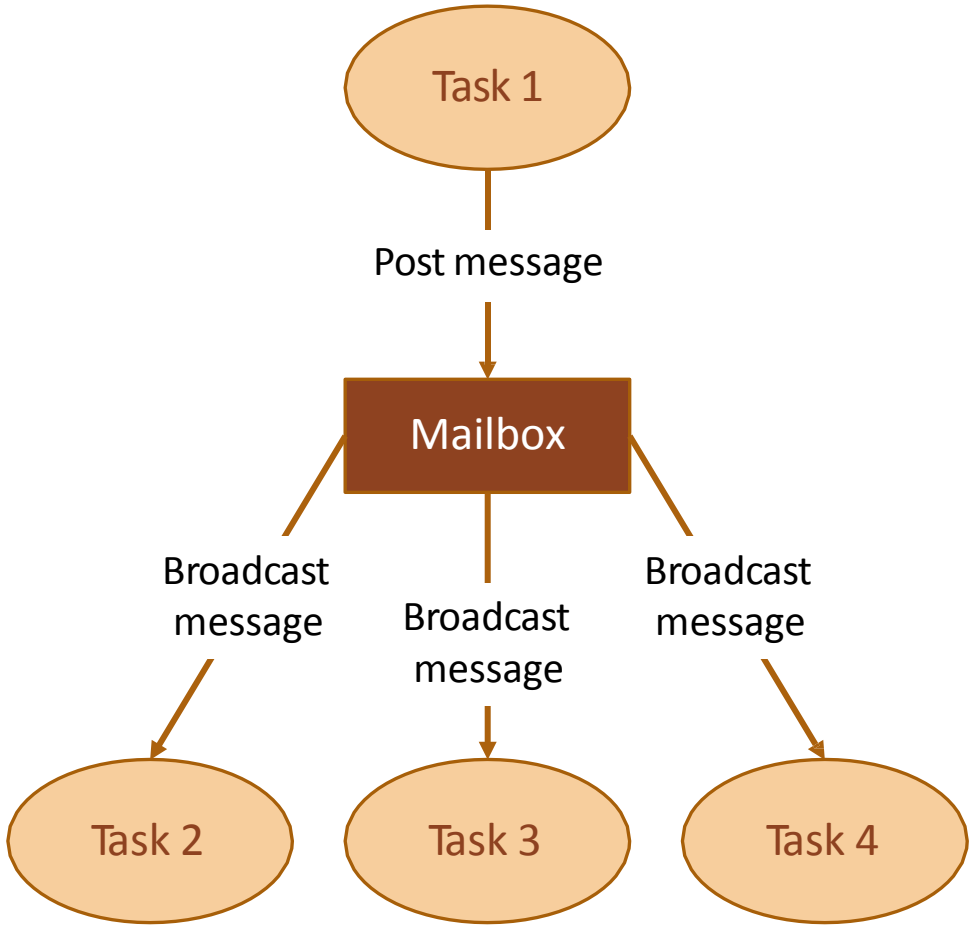


Fig.: Concept of Mailbox based indirect messaging for IPC

Mailbox (continued)

- The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls.
- Mailbox and message queues are same in functionality.
  - The only difference is in the number of messages supported by them.
  - Both of them are used for passing data in the form of message(s) from a task to another task(s).
  - Mailbox is used for exchanging a single message between two tasks or between an Interrupt Service Routine (ISR) and a task.
  - Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox.

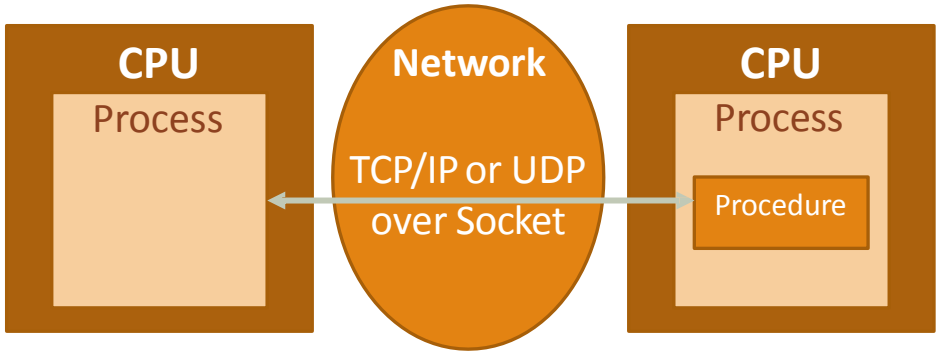
Signalling

- Signalling is a primitive way of communication between processes/threads.
- Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting.
- Signals are not queued and they do not carry any data.
- E.g. Communication mechanisms used in RTX51 Tiny OS, inter process communication in VxWorks OS Kernel are examples for signalling.

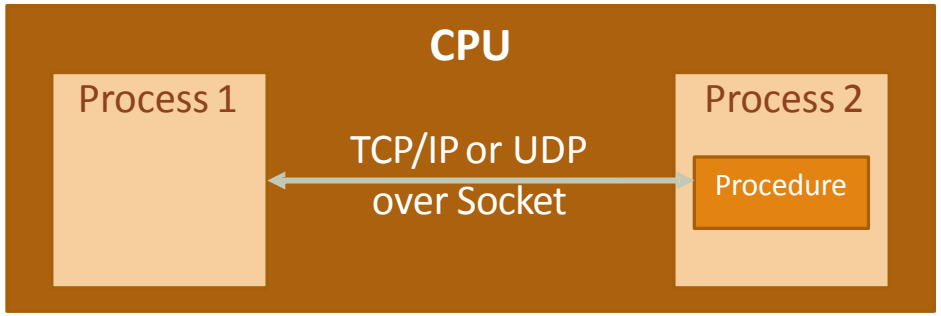
Remote Procedure Call (RPC) and Sockets

- Remote Procedure Call (RPC) 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)*.
- RPC is mainly used for distributed applications like client-server applications.
  - With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different operating systems).
  - 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.

Remote Procedure Call (RPC) and Sockets (continued)



Processes running on different CPUs which are networked



Processes running on the same CPU

Fig.: Concept of Remote Procedure Call (RPC) for IPC

Remote Procedure Call (RPC) and Sockets (continued)

- It is possible to implement RPC communication with different invocation interfaces.
- 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).
  - In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process.
  - In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure.
    - The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

Remote Procedure Call (RPC) and Sockets (continued)

- On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities.
- The client applications (processes) should authenticate themselves with the server for getting access.
- Authentication mechanisms like IDs, public key cryptography (like DES, 3DES), etc. are used by the client for authentication.
- Without authentication, any client can access the remote procedure.
  - This may lead to potential security risks.

Remote Procedure Call (RPC) and Sockets (continued)

- 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.

Remote Procedure Call (RPC) and Sockets (continued)

- INET sockets are classified into:
  - **Stream sockets**
    - These are connection oriented and they use TCP to establish a reliable connection.
  - **Datagram sockets**
    - These rely on UDP for establishing a connection.
    - The UDP connection is unreliable when compared to TCP.

Remote Procedure Call (RPC) and Sockets (continued)

- The client-server communication model uses a socket at the client side and a socket at the server side.
- A port number is assigned to both of these sockets.
  - The client and server should be aware of the port number associated with the socket.
- In order to start the communication, the client needs to send a connection request to the server at the specified port number.
- The client should be aware of the name of the server along with its port number.
- The server always listens to the specified port number on the network.
- Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server.

Remote Procedure Call (RPC) and Sockets (continued)

- The client uses the host name and port number of server for sending requests and server uses the client's name and port number for sending responses.
- If the client and server applications (both processes) are running on the same CPU, both can use the same host name and port number for communication.
- The physical communication link between the client and server uses network interfaces like Ethernet or Wi-Fi for data communication.
- The underlying implementation of socket is OS kernel dependent.
  - Different types of OSs provide different socket interfaces.

Task Synchronisation

Task Synchronisation Issues

- In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources.
- The processes communicate with each other with different IPC mechanisms including shared memory and variables.
- Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this.
  - This would result in unexpected results.
  - This can be solved by making each process aware of the access of a shared resource either directly or indirectly.

Task Synchronisation Issues (continued)

- The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as ‘Task/Process Synchronisation’.
- Various task communication/synchronisation issues may arise in a multitasking environment if processes are not synchronised properly.
  - Racing
  - Deadlock

Racing

- Let us have a look at the following piece of code:

```
#include<windows.h>
#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++;
 }
}
```

Racing (continued)

```
//*********************************************************************
//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;
}
```

Racing (continued)

- From a programmer perspective, the value of counter will be 10 at the end of execution of processes A & B.
  - But in a real world execution, the result depends on the process scheduling policies adopted by the OS kernel.
- 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.

Racing (continued)

- 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
```

- Both the processes Process A and Process B contain the program statement `counter++;`

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 [ebp-4], eax`	`mov dword ptr [ebp-4], eax`

Racing (continued)

- Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the `counter++;` statement.
- 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 figure.

Racing (continued)

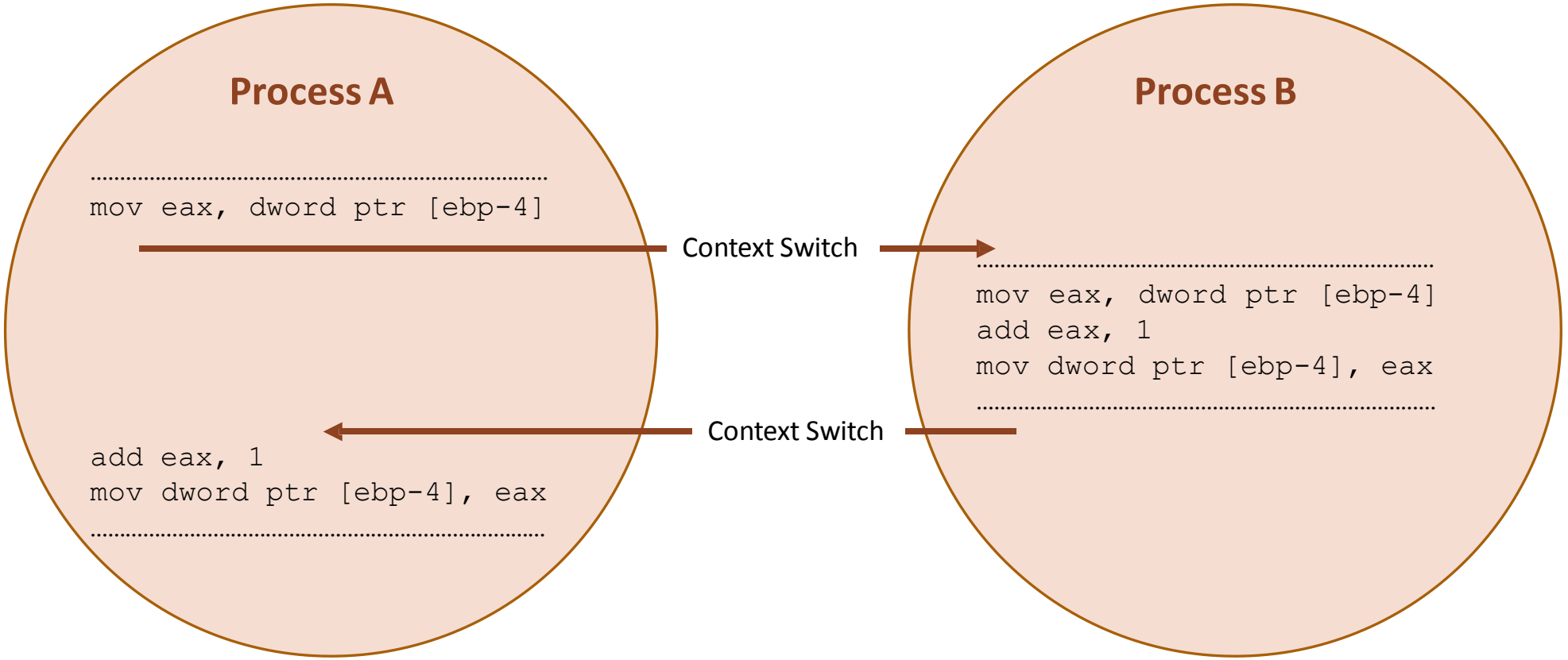


Fig.: Race Condition

Racing (continued)

- 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.
- 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.
- This issue wouldn't have occurred if the underlying actions corresponding to the program statement `counter++;` is finished in a single CPU execution cycle.
- The best way to avoid this situation is to make the access and modification of shared variables mutually exclusive.
  - Meaning when one process accesses a shared variable, prevent the other processes from accessing it.

Racing (continued)

- To summarise, *Racing* or *Race condition* is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently.
- In a Race condition, the final value of the shared data depends on the process which acted on the data finally.

Deadlock

- A race condition produces incorrect results, whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes.
- This is similar to traffic jam issues in a junction as illustrated in the figure.

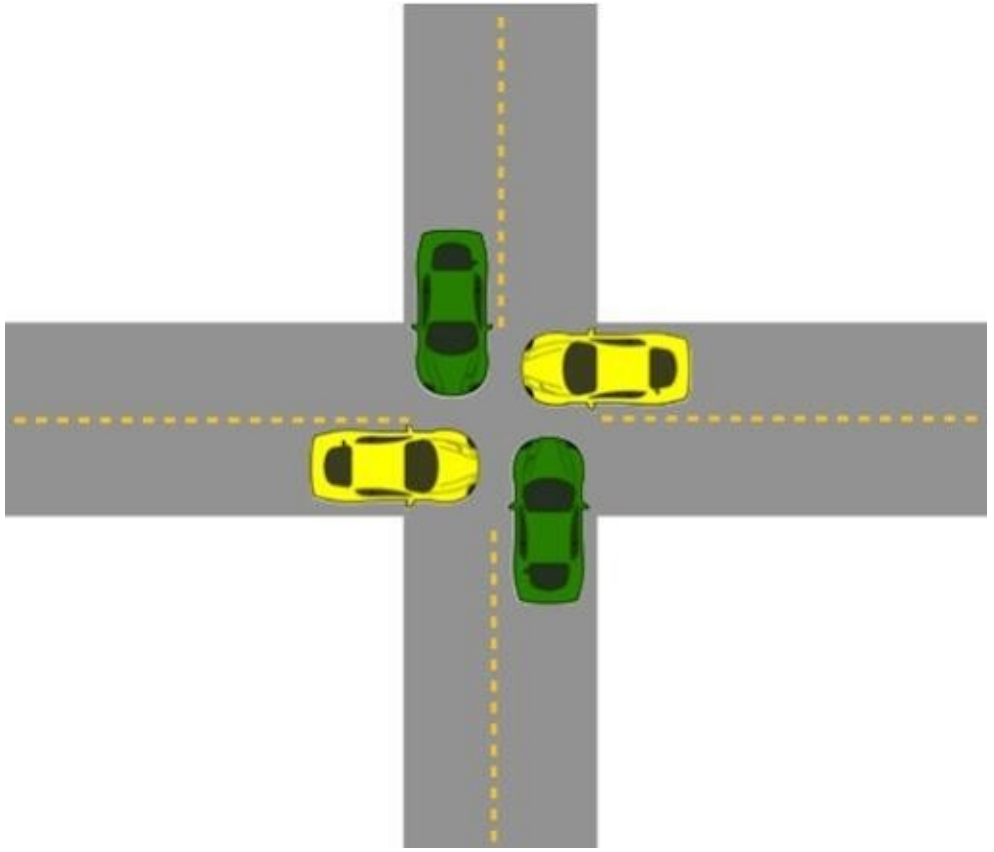


Fig.: Deadlock Visualisation

Deadlock (continued)

- In its simplest form, *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.
- 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.
- The result of the competition is 'deadlock'.
- None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes.

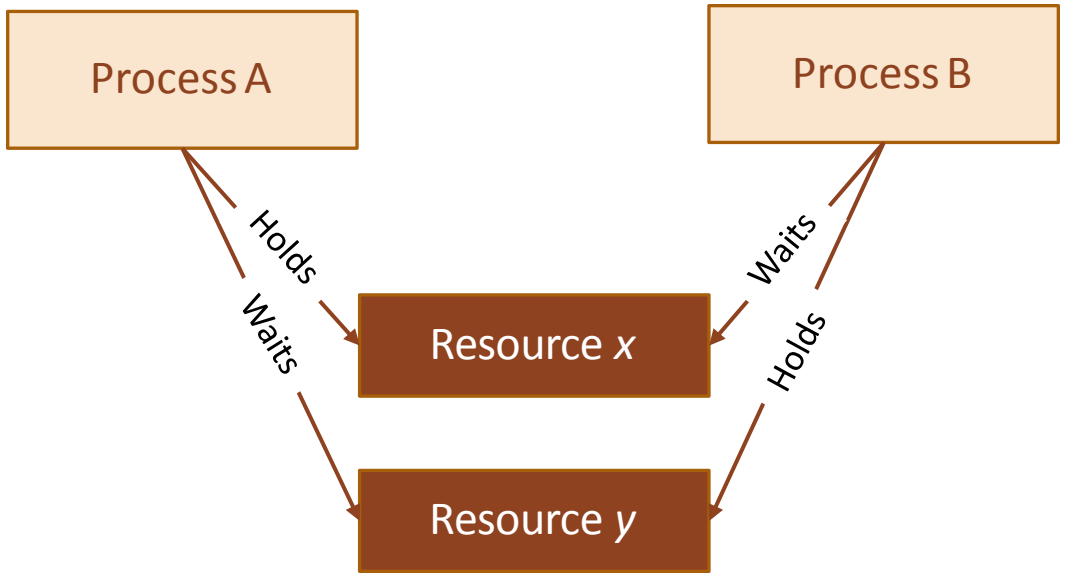


Fig.: Scenario leading to deadlock

Deadlock (continued)

- The different conditions favouring a deadlock situation are:
  - ***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 hardware in an embedded device.
  - ***Hold and 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.

Deadlock (continued)

- ***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.
- 'Deadlock' is a result of the combined occurrence of these four conditions listed above.
  - These conditions were first described by E. G. Coffman in 1971 and it is popularly known as *Coffman conditions*.

Deadlock Handling

- A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation.
- If a deadlock occurs, the reaction to it by OS is nonuniform.
- 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 that 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.

Deadlock Handling (continued)

- ***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) has 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.

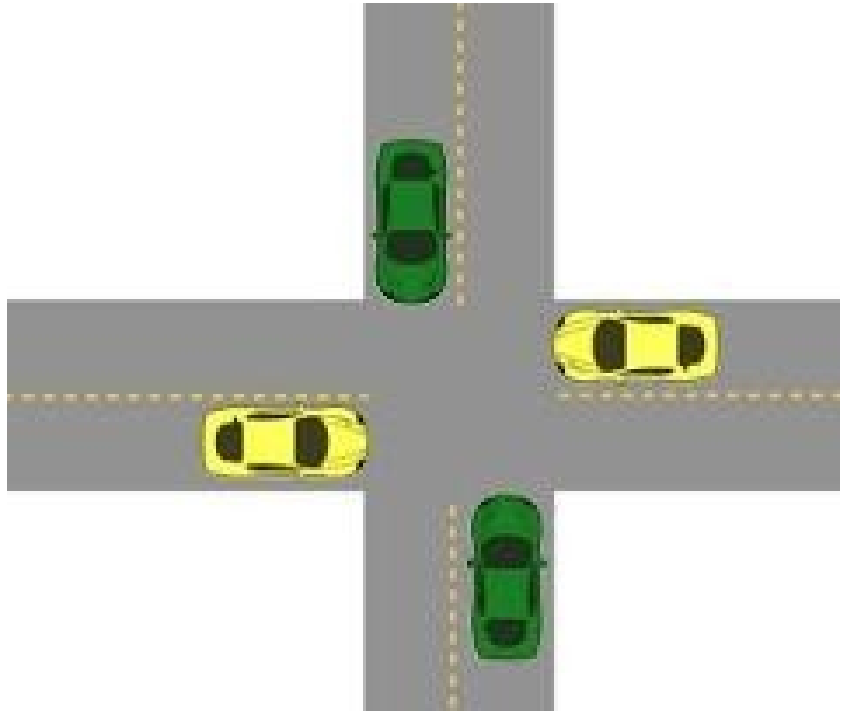


Fig.: ‘Back up cars' technique for deadline recovery

Deadlock Handling (continued)

- 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 analysing the resource graph by graph analyser 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 favouring the deadlock situation.

Deadlock Handling (continued)

- Ensure that a process does not hold any other resources when it requests a resource.
  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.
  1. Release all the resources currently held by a process if a request made by the process for anew resource is not able to fulfil 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.

Deadlock (continued)

- **Livelock**
  - The *Livelock* condition is similar to the deadlock condition except that a process in livelock condition changes its state with time.
  - While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion.
  - The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor.
    - Both the persons move towards each side of the corridor to allow the opposite person to cross.
    - Since the corridor is narrow, none of them are able to cross each other.
    - Here both of the persons perform some action but still they are unable to achieve their target, cross each other.

Deadlock (continued)

- **Starvation**
  - In the multitasking context, *starvation* is the condition in which a process does not get the resources required to continue its execution for a long time.
  - As time progresses, the process starves on resource.
  - Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

Task Synchronisation Techniques

- Process/Task synchronisation is essential for
  1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
  2. Ensuring proper sequence of operation across processes.
  3. Communicating between processes.
- The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as *'critical section'.*
  - In order to synchronise the access to shared resources, the access to the *critical section* should be exclusive.

Task Synchronisation Techniques (continued)

- The exclusive access to critical section of code is provided through *mutual exclusion* mechanism.
- Consider two processes Process A and Process B running on a multitasking system.
- Process A is currently running and it enters its critical section.
- Before Process A completes its operation in the critical section, the scheduler preempts Process A and schedules Process B for execution (Process B is of higher priority compared to Process A).
- Process B also contains the access to the critical section which is already in use by Process A.
- If Process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted.
- A mutual exclusion policy enforces mutually exclusive access of critical sections.

Task Synchronisation Techniques (continued)

- Mutual exclusion blocks a process.
- Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories:
  - Mutual Exclusion through Busy Waiting/Spin Lock
  - Mutual Exclusion through Sleep & Wakeup

Mutual Exclusion through Sleep & Wakeup

- When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes 'Sleep' and enters the 'blocked' state.
- The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section.
- The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section.
- The *'Sleep & Wakeup'* policy for mutual exclusion can be implemented in different ways.
  - Windows XP/CE OS kernels use semaphores for *'Sleep & Wakeup'* policy implementation for mutual exclusion.

Semaphore

- Semaphore is a *'Sleep & Wakeup'* based mutual exclusion implementation for shared resource access.
- Semaphore is a system resource and the 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 acquired 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 for the 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.

Semaphore (continued)

- Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two, namely *'Binary Semaphore'* and *'Counting Semaphore'*.
- The *Binary Semaphore* provides 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 owned by a process.
  - Under certain OS kernel, it is referred as *mutex*.
- The *Counting Semaphore* limits the access of resources by a fixed number of processes/threads.
  - *Counting Semaphore* maintains a count between zero and a value.
  - It limits the usage of the resource to the maximum value of the count supported by it.

Counting Semaphore

- The *Counting Semaphore* limits the access of resources by a fixed number of processes/threads.
- *Counting Semaphore* maintains a count between zero and a value.
- It limits the usage of the resource to the maximum value of the count supported by it.
- The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero.
- 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 *'Semaphore object'* is set to 'non-signalled' 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).

Counting Semaphore (continued)

- A real world example for the counting semaphore concept is the dormitory system for accommodation, as shown in the figure.
- A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory.
- If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability.
  - If beds are available in the dorm, the caretaker will hand over the keys to the user.
  - If beds are not available currently, the user can register his/her name to get notifications when a slot is available.
- Those who are availing the dormitory share the dorm facilities like TV, telephone, toilet, etc.
- When a dorm user vacates, he/she gives the keys back to the caretaker.
  - The caretaker informs the users, who booked in advance, about the dorm availability.

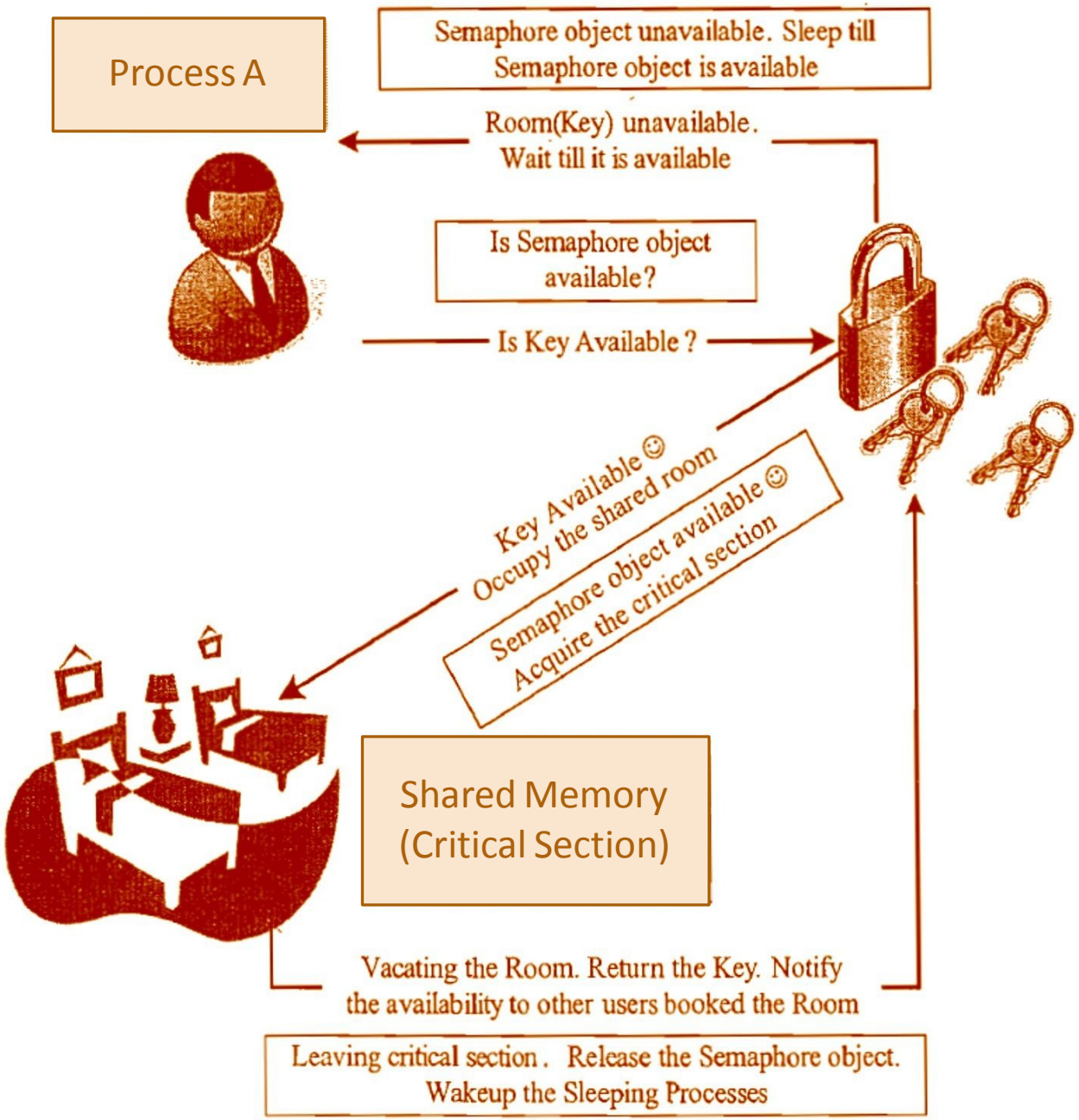


Fig.: The Concept of Counting Semaphore

Counting Semaphore vs. Binary Semaphore

- *Counting Semaphores* are similar to *Binary Semaphores* in operation.
- The only difference between *Counting Semaphore* and *Binary Semaphore* is that
  - Binary Semaphore can only be used for exclusive access, *whereas*
  - Counting Semaphores can be used for both
    - exclusive access (by restricting the maximum count value associated with the semaphore object to one at the time of creation of the semaphore object) *and*
    - limited access (by restricting the maximum count Value associated with the semaphore object to the limited number at the time of creation of the semaphore object)

Binary Semaphore (Mutex)

- *Binary Semaphore* (Mutex) is a synchronisation object provided by OS for process/thread synchronisation.
- Any process/thread can create a *'mutex object'* and other processes/threads of the system can use this *'mutex object'* for synchronising the access to critical sections.
- Only one process/thread can own the *'mutex object'* at a time.
- The state of a mutex object is set to ‘signalled’ when it is not owned by any process/thread, and set to ‘non-signalled’ when it is owned by any process/thread.

Binary Semaphore (Mutex) (continued)

- A real world example for the mutex concept is the hotel accommodation system (lodging system), as shown in the figure.
- The rooms in a hotel are shared for the public.
- Any user who pays and follows the norms of the hotel can avail the rooms for accommodation.
- A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability.
  - If room is available, the receptionist will handover the room key to the user.
  - If room is not available currently, the user can book the room to get notifications when a room is available.
- When a person gets a room, he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc.
- When a user vacates the room, he/she gives the keys back to the receptionist.
  - The receptionist informs the users, who booked in advance, about the room's availability.

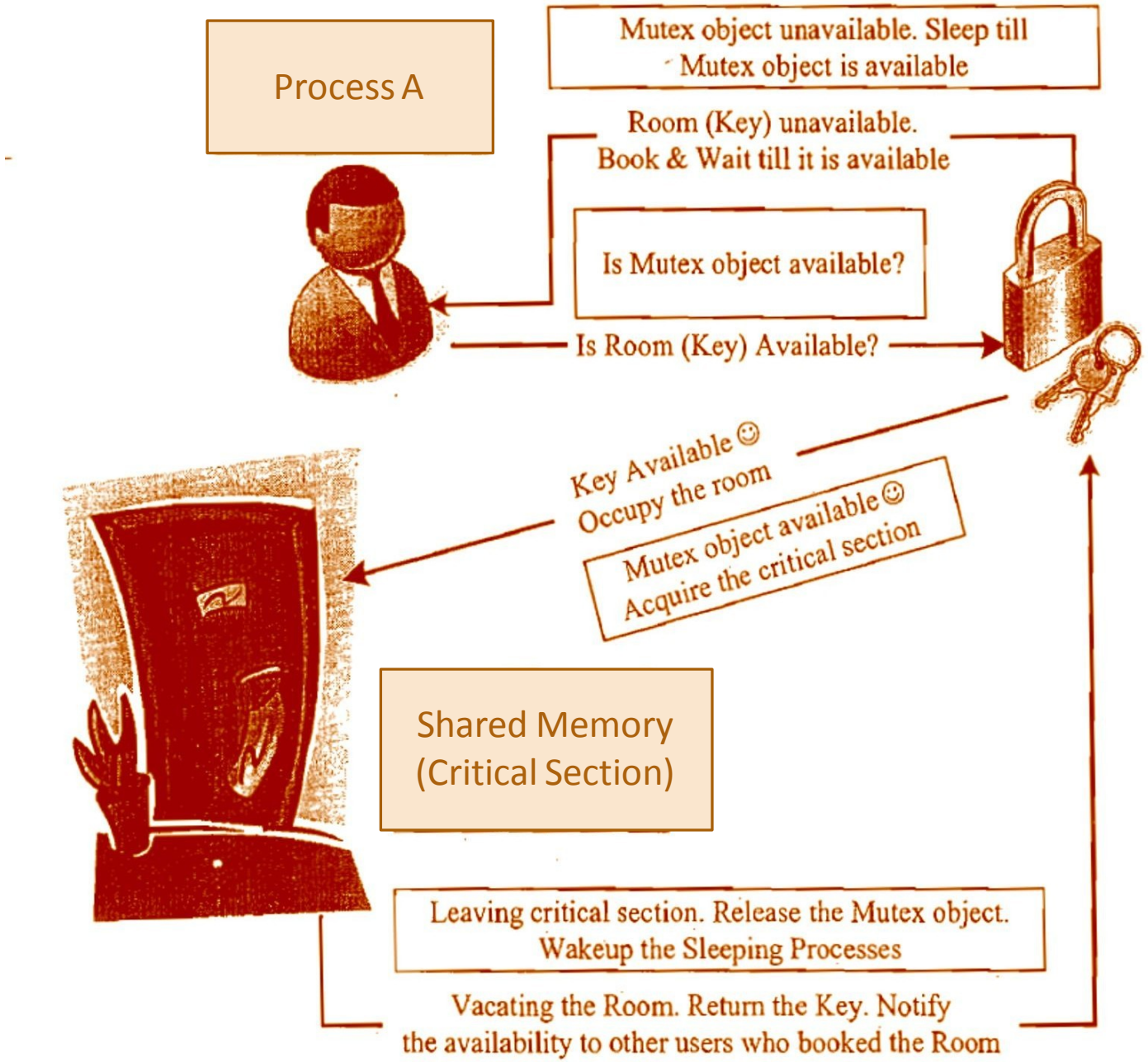


Fig.: The Concept of Binary Semaphore (Mutex)

How to Choose an RTOS

How to Choose an RTOS

- The decision of choosing an RTOS for an embedded design is very crucial.
- A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS.
- The requirements that needs to be analysed in the selection of an RTOS for an embedded design fall under two categories:
  - Functional requirements
  - Non-functional requirements

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.

Functional Requirements (continued)

- **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 behaviour.
  - The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS.
  - Analyse 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.

Functional Requirements (continued)

- **Inter Process Communication and Task Synchronisation**
  - The implementation of Inter Process Communication and Synchronisation is OS kernel dependent.
  - Certain kernels may provide a bunch of options whereas others provide very limited options.
- **Modularisation 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 modularisation where in 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.

Functional Requirements (continued)

- **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) customised 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 for 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 customising an Open source OS.
  - The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

Non-Functional Requirements (continued)

- **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.

Non-Functional Requirements (continued)

- **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 form of e-mail, on-call services, etc. for bug fixes, critical patch updates and support for production issues, etc. should be analysed thoroughly.

Integration and Testing of Embedded Hardware and Firmware

Integration and Testing of Embedded Hardware and Firmware – Introduction

- Integration and testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development.
- Embedded hardware and firmware are developed in various steps.
  - 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.
- 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.

Integration and Testing of Embedded Hardware and Firmware – Introduction (continued)

- 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.
- As far as the embedded firmware is concerned, its targeted functionalities can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE (Integrated Development Environment).

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.
- For non-operating system based embedded products, if the processor/controller contains 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.

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* (also called *programmer*).

Fig.: Firmware Embedding Tool – Device Programmer: LabTool-48UXP

- The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals.

Out-of-Circuit Programming (continued)

- The programmer contains a ZIF socket with locking pin to hold the device to be programmed.
- The programming device will be under the control of a utility program running on a PC.
- Usually the programmer 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.

Fig.: Universal Programmer

Out-of-Circuit Programming (continued)

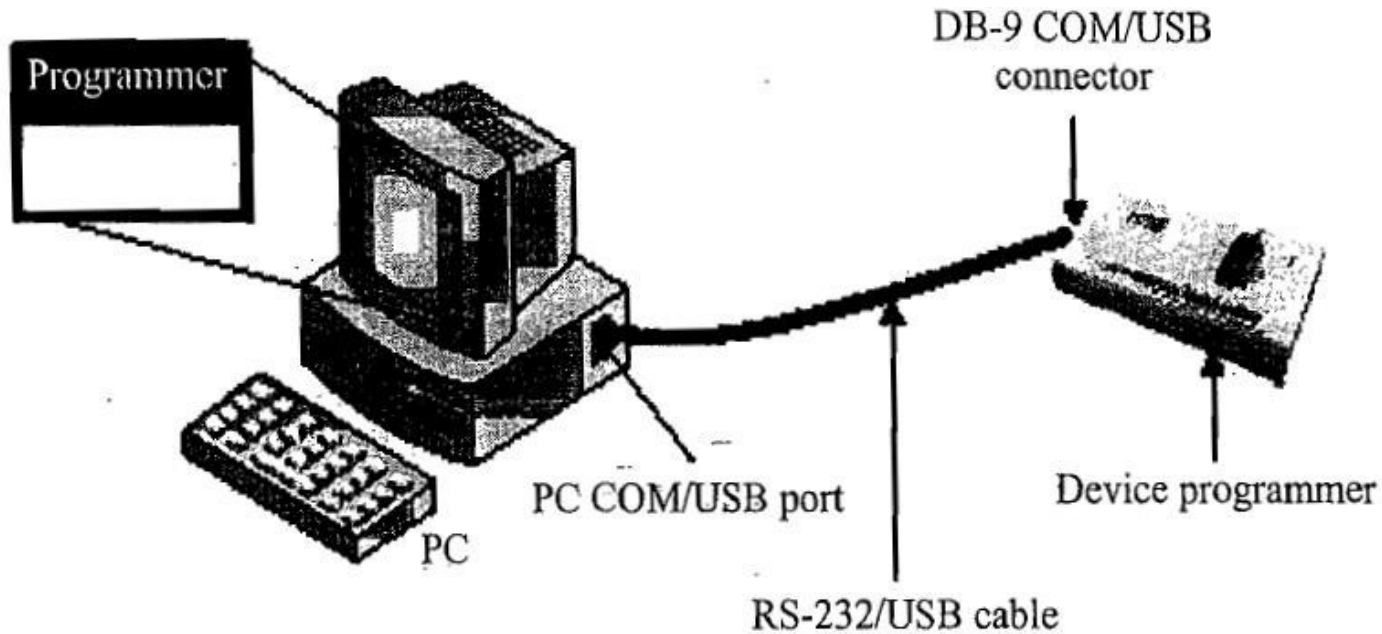


Fig.: Interfacing of Device Programmer with PC

Out-of-Circuit Programming (continued)

- 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 (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.
  6. Lock the ZIF socket.

Out-of-Circuit Programming (continued)

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 successful 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 take the device out of programmer.

Out-of-Circuit Programming (continued)

- Once 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.
- If you 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.
  - Some old embedded systems may be built around UVEPROM chips and such chips should be erased using a separate 'UV Chip Eraser' before programming.

Out-of-Circuit Programming (continued)

- Drawbacks
  - 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 programmer facilitates programming of only one chip at a time and it is not suitable for batch production.
      - Can be resolved using a 'Gang Programmer', which contains multiple ZIF sockets (4 to 8) and capable of programming multiple devices at a time.
      - But it is bit expensive compared to an ordinary programmer.
  - Another big drawback of out-of-circuit programming is that once the product is deployed in the market in a production environment, it is very difficult to upgrade the firmware.

Out-of-Circuit Programming (continued)

Fig.: Gang Programmer

Out-of-Circuit Programming (continued)

- Applications
  - 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)

- Here, the 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 utility will be in a serial format.
  - The serial protocols used for ISP may be 'Joint Test Action Group (JTAG)' or 'Serial Peripheral Interface (SPI)' or any other proprietary protocol.

In System Programming (ISP) (continued)

- In order to perform ISP operations, the target device should be powered up in a special ‘ISP mode’.
- ISP mode allows the device to communicate with an external host, such as a PC or terminal, through a serial interface.
- The device receives commands and data from the host, erases and reprograms code memory according to the received command.
- Once the ISP operations are completed, the device is re-configured so that it will operate normally by applying a reset or a re-power up.

In System Programming (ISP) (continued)

- Devices with *SPI - In System Programming* support contains a built-in SPI interface *(Serial Peripheral Interface)* and the on-chip EEPROM or FLASH memory is programmed through this interface.
- The primary I/O lines involved in SPI - In System Programming are:
  - MOSI - Master Out Slave In
  - MISO - Master In Slave Out
  - SCK - System Clock
  - RST - Reset of Target Device
  - GND - Ground of Target Device

In System Programming (ISP) (continued)

- 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.
- Since the target device works under a supply voltage less than 5V (TTL/CMOS), it is better to connect these lines of the target device with the parallel port of the PC.
  - Since parallel port operations are also at 5V logic, no need for any other intermediate hardware for signal conversion.
- Standard SPI-ISP utilities are feely available on the internet and there is no need for going for writing own program.

In System Programming (ISP) (continued)

- For ISP operations, 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 XTAL1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.
  4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/P1.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 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/P1 .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.

In System Programming (ISP) (continued)

- 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.
  - 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.
- By default the Reset vector starts the code memory execution at location 0000H.
- If the ISP mode is enabled through the special ISP Power up sequence, the execution will start at the *Boot ROM* vector location.

In System Programming (ISP) (continued)

- In System Programming technique is the best advised programming technique for development work since the effort required to re-program the device in case of firmware modification is very little.
- Firmware upgrades for products supporting ISP is quite simple.

In Application Programming

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

In Application Programming (continued)

- The *Boot ROM* resident API instructions which perform various functions such as programming, erasing, and reading the Flash memory during ISP-mode, are made available to the end-user written firmware for IAP.
  - Thus, it is possible for an end-user application to perform operations on the Flash memory.
- A common entry point to these API routines is provided for interfacing them to the end-user's application.
- Functions are performed by setting up specific registers as required by a specific operation and performing a call to the common entry point.
  - Like any other subroutine call, after completion of the function, control will return to the end-user's code.

In Application Programming (continued)

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

Use of Factory Programmed Chip

- It is possible to embed the firmware into the target processor/controller memory at the time of chip fabrication itself.
  - Such chips are known as 'Factory programmed chips’.
- Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory.
- Factory programmed chips are convenient for mass production applications and it greatly reduces the product development time.
- It is not recommended to use factory programmed chips for development purpose where the firmware undergoes frequent changes.
- Factory programmed ICs are bit expensive.

Firmware Loading for Operating System 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 loader* 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 initialisation implementation for initialising the supported interfaces like UART, TCP/IP etc.

Firmware Loading for Operating System Based Devices (continued)

- Boot loader implements menu options for selecting the source for OS image to load.
  - E.g. Load from FLASH ROM, Load from Network, Load through UART etc.
- In case of the network based loading, the boot loader broadcasts the target's presence over the network and the host machine on which the OS image resides can identify the target device by capturing this message.
  - Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

Embedded System Development Environment

Embedded System Development Environment – Block Diagram

- The embedded system development environment consists of:
  - A Development Computer (PC) or Host, which acts as the heart of the development environment,
  - Integrated Development Environment (IDE) Tool for embedded firmware development and debugging,
  - Electronic Design Automation (EDA) Tool for Embedded Hardware design,
  - An emulator hardware for debugging the target board,
  - Signal sources (like Function generator) for simulating the inputs to the target board,
  - Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.) *and*
  - The target hardware.

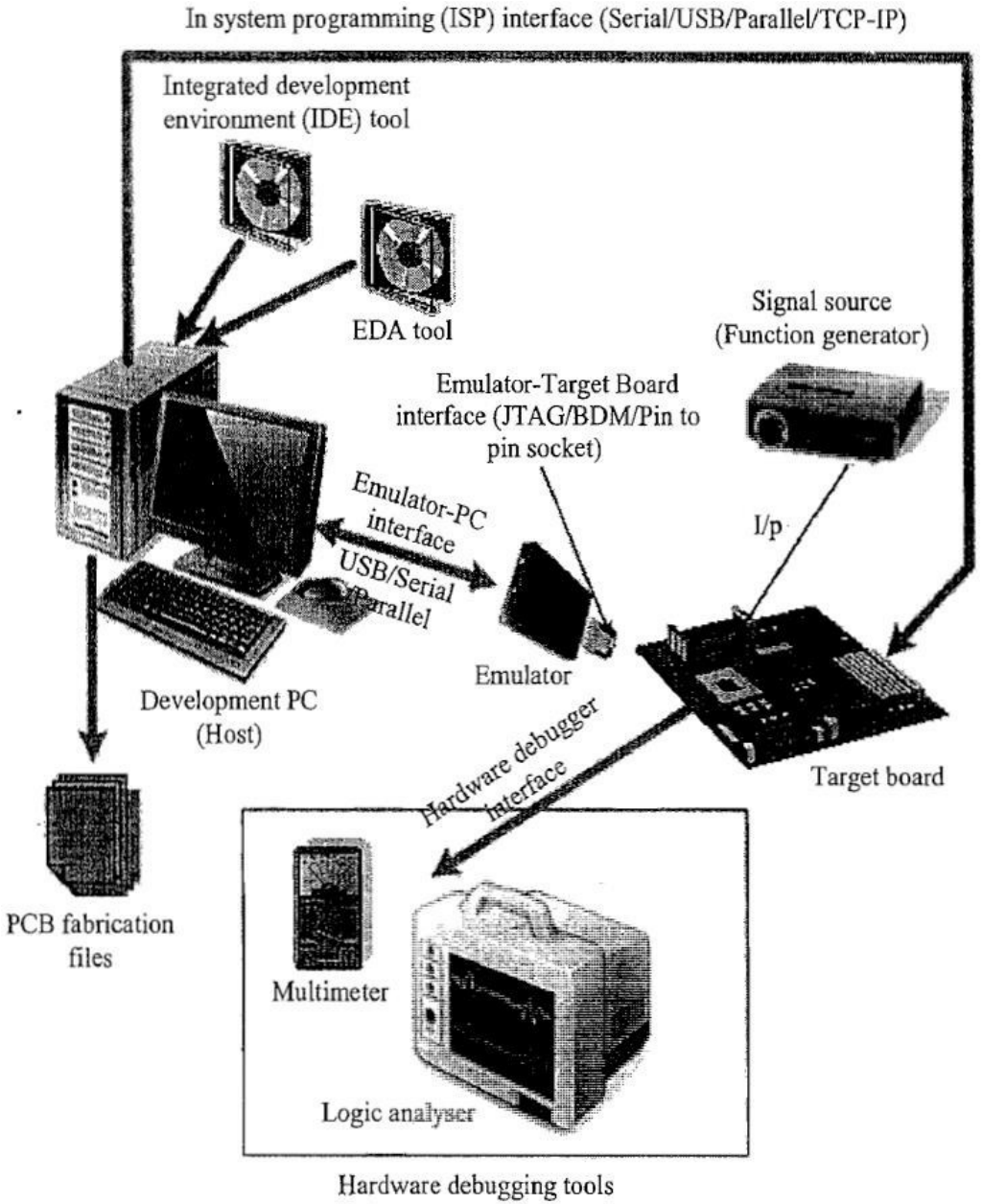


Fig.: The Embedded System Development Environment

Embedded System Development Environment – Block Diagram (continued)

- The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement and they are supplied as Installable files in CDs by vendors.
- These tools need to be installed on the host PC used for development activities.
- These tools can be either freeware or licensed copy or evaluation versions.
  - Licensed versions of the tools are fully featured and fully functional whereas trial versions fall into two categories, tools with limited features, and full featured copies with limited period of usage.

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-compiler (for cross platform development and compiler for same platform development),
  - Linker *and*
  - Debugger.

Integrated Development Environment (IDE) (continued)

- IDEs used in embedded firmware development are slightly different from the generic IDs used for high level language based development for desktop applications.
- In embedded applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open Source.
  - Keil µVision from Keil software is an example for a third party IDE, which is used for developing embedded firmware for 8051 family microcontrollers and also ARM microcontrollers.
  - MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers.
  - CodeWarrior by Metrowerks is an example of IDE for ARM family of processors.

Disassembler/Decompiler

- *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, disassembling 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.

Disassembler/Decompiler (continued)

- 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 analysing the presence of malicious codes (virus information) in an executable image.

Simulators

- *Simulator* is a software tool used 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.
- Simulators simulate the target hardware and the firmware execution can be inspected using simulators.
- The features of simulator based debugging are:
  - Purely software based
  - Doesn't require a real target system
  - Very primitive (Lack of featured I/O support. Everything is a simulated one)
  - Lack of Real-time behaviour

Simulators (continued)

- **Advantages of Simulator Based Debugging**
  - Simulator based debugging techniques are simple and straightforward.
- The major advantages of simulator based firmware debugging techniques are:
  - **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 finalised.
    - This saves development time.

Simulators (continued)

- **Simulate I/O Peripherals**
  - Simulator provides the option to simulate various I/O peripherals.
  - Using simulator's I/O support, the values for I/O registers can be edited 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 behaviour of the firmware under abnormal input conditions.

Simulators (continued)

- **Limitations of Simulator Based Debugging**
  - **Deviation from Real Behaviour**
    - 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 behaviour.
      - The debugging is developer driven and it is no way capable of creating a real-time behaviour.
    - Moreover in a real application the I/O condition may be varying or unpredictable.
      - Simulation goes for simulating those conditions for known values.

Emulators

- *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.
- A circuit for emulating target device remains independent of a particular target system and processor.
- The emulator emulates the target system with extended memory and with code downloading ability during the edit-test-debug cycles.
- Emulators maintain the original look, feel, and behaviour of the embedded system.
- Even though the cost of developing an emulator is high, it proves to be the more cost efficient solution over time.
- Emulators allow software exclusive to one system to be used on another.
- It is more difficult to design emulators and it also requires better hardware than the original system.

Simulator vs. Emulator

- Simulator is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU.
- The simulator is a host-based program that imitates the functionality and instruction set of the target processor.
- In summary, the simulator *'simulates'*
- the target board CPU.

- Emulator is a self-
- device which emu
- The emulator hardware co emulation logic and it is h debugging application ru development PC on one e the target board through the other end.
- In summary, the emulator board CPU.

Debuggers

- *Debugger* is a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.
- *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

- Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behaviour.
- There are several techniques for firmware debugging:
  - Incremental EEPROM Burning Technique
  - Inline Breakpoint Based Firmware Debugging
  - Monitor Program Based Firmware Debugging
  - In Circuit Emulator (ICE) Based Firmware Debugging
  - On Chip Firmware Debugging (OCD)

Incremental EEPROM Burning Technique

- This is the most primitive type of firmware debugging technique.
- In this technique, 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.
  - This means the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.
- In this technique, we are not doing any debugging, but we are observing the status of firmware execution as a debug method.
- Incremental firmware burning technique is widely adopted in small, simple system developments and in product development where time is not a big constraint (e.g. R&D projects).
  - It is also very useful in product development environments where no other debug tools are available.

Inline Breakpoint Based Firmware Debugging

- This 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, an inline debug code is inserted immediately after the point.
- The debug code is a *printf()* function which prints a string given as per the firmware.
- The debug codes (*printf()* commands) can be inserted at each point where you want to ensure the firmware execution is covering that point.
- The source code is cross-compiled along with the debug codes embedded within it.
- The corresponding hex file is burned into the EEPROM.
- The *printf()* generated data can be viewed on the *HyperTerminal.*

Monitor Program Based Firmware Debugging

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

Monitor Program Based Firmware Debugging (continued)

- Once the commands for each operation is fixed, the code is written for performing the actions corresponding to these commands.
- The commands may be received through any of the external interface of the target processor (e.g. RS-232C serial interface/parallel interface/USB, etc.).
  - The monitor program should query this interface to get commands or should handle the command reception if the data reception is implemented through interrupts.
- On receiving a command, it is examined and the action corresponding to it is performed.
- The entire code stuff handling the command reception and corresponding action implementation is known as the "*Monitor Program*".
- After the successful completion of the *'Monitor Program'* development, it is compiled and burned into the FLASH memory or ROM of the target board.
- The code memory containing the monitor program is known as the *'Monitor ROM'*.

Monitor Program Based Firmware Debugging (continued)

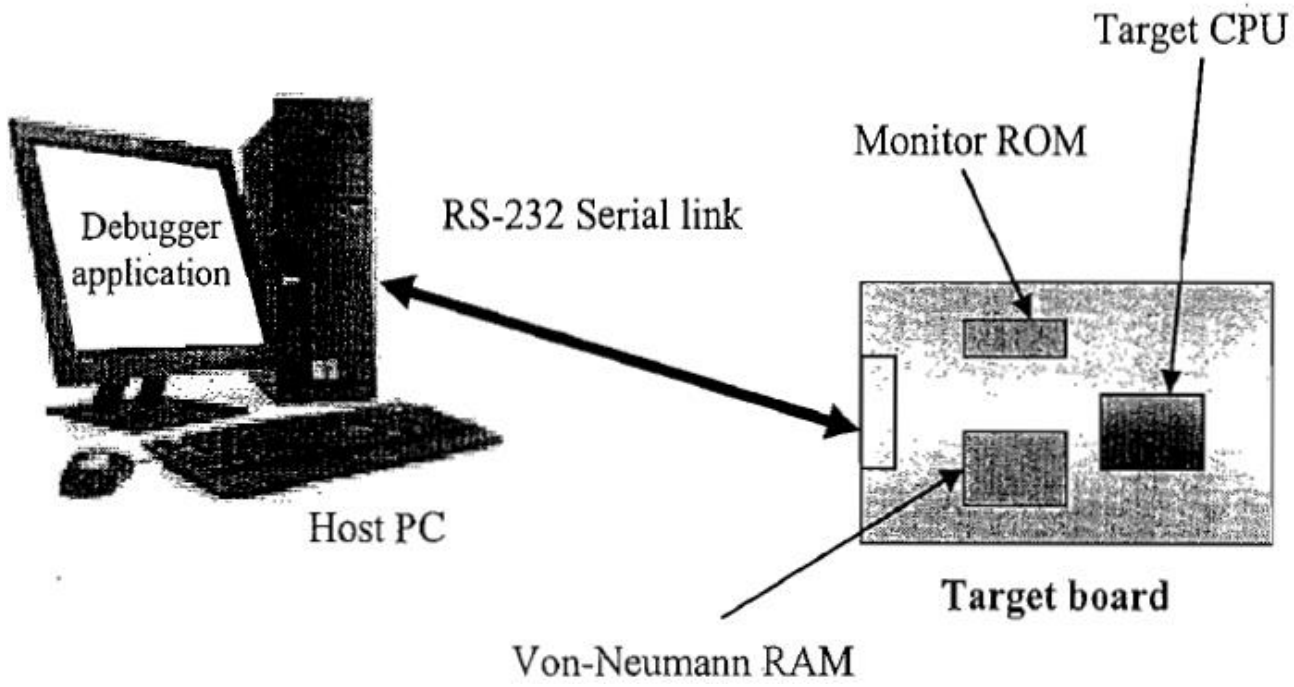


Fig.: Monitor Program Based Target Firmware Debug Setup

In Circuit Emulator (ICE) Based Firmware Debugging

- Emulator is a special hardware device used for emulating the functionality of a processor/controller and performing various debug operations like halt firmware execution, set breakpoints, get or set internal RAM/CPU register, etc.
- Nowadays pure software applications which perform the functioning of a hardware emulator is also called as 'Emulators' (though they are 'Simulators' in operation).
- The emulator application for emulating the operation of a PDA phone for application development is an example of a 'Software Emulator’.
- A hardware emulator is controlled by a debugger application running on the development PC.
  - Most of the IDEs incorporate debugger support for some of the emulators commonly available in the market.

In Circuit Emulator (ICE) Based Firmware Debugging (continued)

- Figure illustrates the different subsystems and interfaces of an 'Emulator' device.

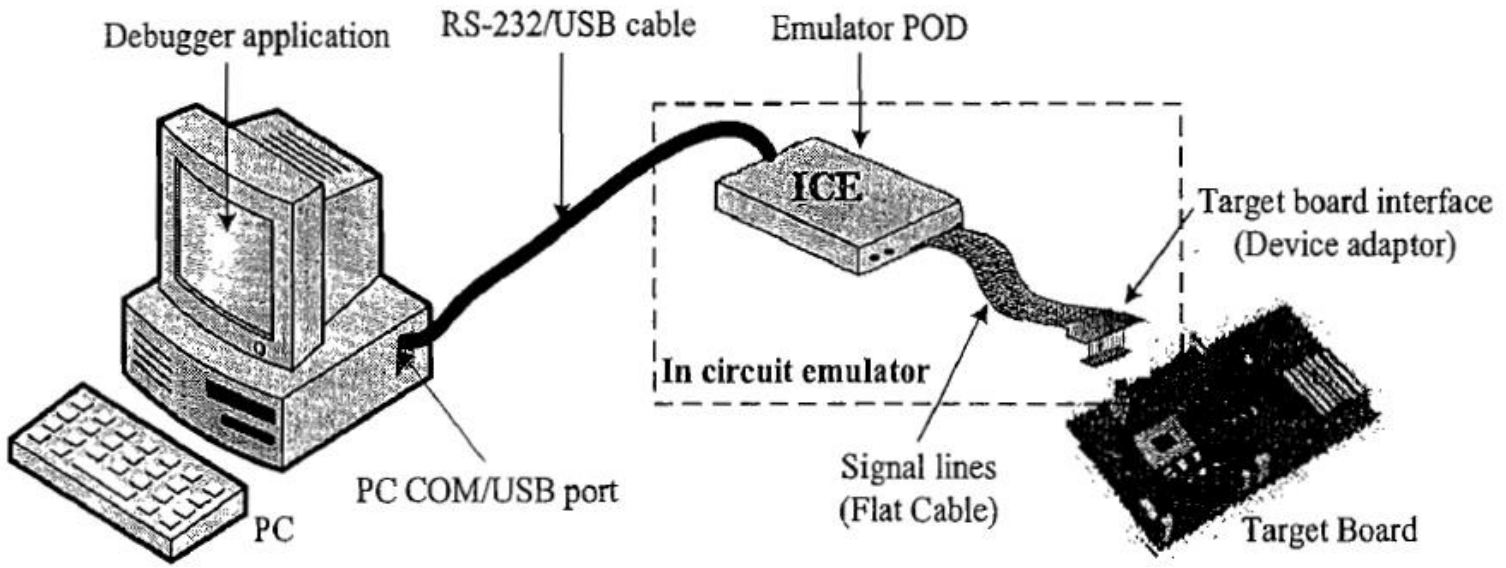


Fig.: In Circuit Emulator (ICE) Based Target Debugging

On Chip Firmware Debugging (OCD)

- Modern processors/controllers incorporate built in debug modules called On Chip Debug (OCD) support.
- Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging.
- 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.
- Some vendors add 'on chip software debug support' through JTAG (Joint Test Action Group) port.
- Usually the on-chip debugger provides the means to set simple breakpoints, query the internal state of the chip and single step through code.
- Background Debug Mode (BDM) and JTAG (Joint Test Action Group) are two commonly used interfaces for OCD.
- OCD module implements dedicated registers for controlling debugging.

Target Hardware Debugging

- Hardware debugging involves the monitoring of various signals of the target board (address/data lines, port pins, etc.), checking the interconnection among various components, circuit continuity checking, etc.
- The various hardware debugging tools used in embedded product development are:
  - Magnifying Glass (Lens)
  - Multimeter
  - Digital CRO
  - Logic Analyser
  - Function Generator

Magnifying Glass (Lens)

- Magnifying glass is the primary hardware debugging tool used for embedded hardware debugging.
- 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.
- The magnifying station incorporates magnifying glasses attached to a stand with CFL tubes for providing proper illumination for inspection.
- The station usually incorporates multiple magnifying lenses.
- The main lens acts as a visual inspection tool for the entire hardware board whereas the other small lens within the station is used for magnifying a relatively small area of the board which requires thorough inspection.

Multimeter

- A multimeter is used for measuring various electrical quantities like voltage (Both AC and DC), current (DC and 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 toolkit of an embedded hardware developer.
- It is the primary debugging tool for physical contact based hardware debugging.

Multimeter

- In embedded hardware debugging, it is mainly used for checking the circuit continuity between different points on the board, measuring the supply voltage, checking the signal value, polarity, etc.
- Both analog and digital versions of a multimeter are available.
  - The digital version is preferred over analog the one for various reasons like
  - readability, accuracy, etc.

Digital CRO

- Cathode Ray Oscilloscope (CRO) is used for waveform capturing and analysis, measurement of signal strength, etc.
- CRO is a very good tool in analysing 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, featurewise they are best suited for target board debugging applications.
- Digital CROS are available for high frequency support and they also incorporate modern techniques for recording waveform over a period of time, capturing waves on the basis of a configurable event (trigger) from the target board.
- Most of the modern digital CROs contain more than one channel and it is easy to capture and analyse various signals from the target board using multiple channels simultaneously.
- Various measurements like phase, amplitude, etc. is also possible with CROs.

Logic Analyser

- Logic analyser 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.
- A logic analyser contains special connectors and clips which can be attached to the target board for capturing digital data.
- In target board debugging applications, a logic analyser captures the states of various port pins, address bus and data bus of the target processor/controller, etc.
- Logic analysers give an exact reflection of what happens when particular line of firmware is running.
- This is achieved by capturing the address line logic and data line logic of target hardware.
- Most modern logic analysers contain provisions for storing captured data, selecting a desired region of the captured waveform, zooming selected region of the captured waveform, etc.

Function Generator

- Function generator is not a debugging tool.
- It is an 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

- *Boundary scan* is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board.
- Boundary scan is also widely used as a debugging method to watch integrated circuit pin states, measure voltage, or analyse sub-blocks inside an integrated circuit.
- The boundary scan test architecture provides a means to test interconnects between integrated circuits on a board without using physical test probes.
- It adds a boundary scan cell that includes a multiplexer and latches, to each pin on the device.

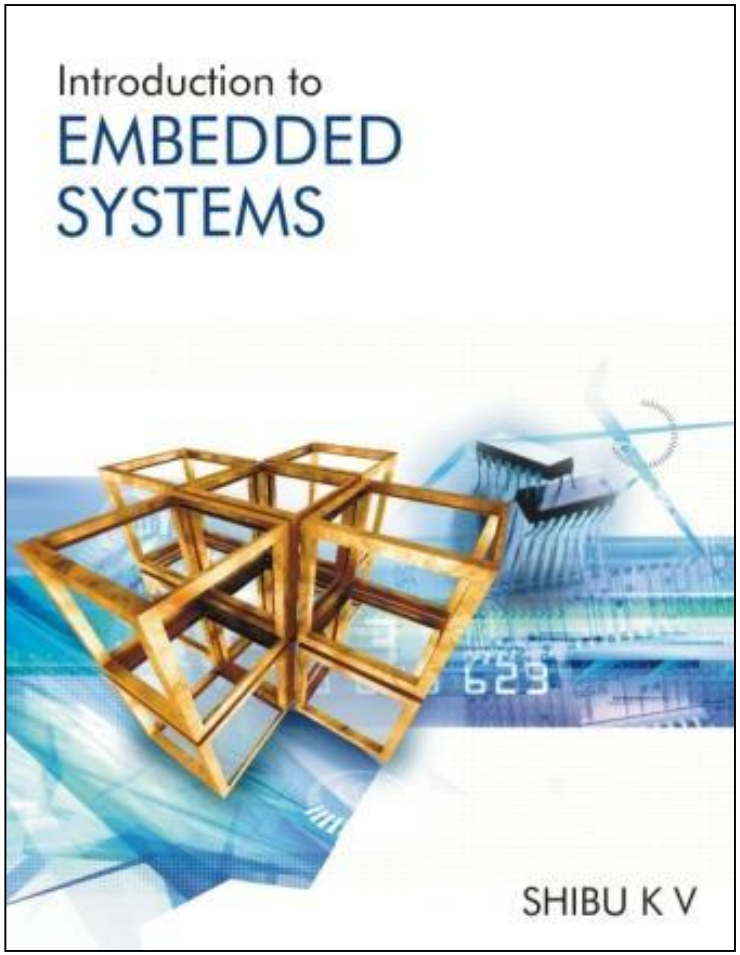
Boundary Scan (continued)

- *Boundary Scan Description Language (BSDL)* is used for implementing boundary scan tests using JTAG.
  - BSDL is a subset of VHDL and it describes the JTAG implementation in a device.
- The benefits provided by boundary scan are:
  - Lower test generation costs
  - Reduced test time
  - Reduced time to market
  - Simpler and less costly testers
  - Compatibility with tester interfaces
  - High-density packaging devices accommodation

References

1. Shibu K V, ***"Introduction to Embedded Systems"***, Tata McGraw Hill, 2009.
2. Raj Kamal, ***"Embedded Systems: Architecture and Programming",*** Tata McGraw Hill, 2008.

Mc Graw Hill Education



Introduction to Embedded System

What is Embedded System?

An Electronic/Electro mechanical system which is designed to perform a specific function and is a combination of both hardware and firmware (Software)

E.g. Electronic Toys, Mobile Handsets, Washing Machines, Air Conditioners, Automotive Control Units, Set Top Box, DVD Player etc…

Embedded Systems are:

- ❑ Unique in character and behavior
- ❑ With specialized hardware and software

Embedded Systems Vs General Computing Systems

General Purpose System	Embedded System
A system which is a combination of generic hardware and General Purpose Operating System for executing a variety of applications	A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
Contain a General Purpose Operating System (GPOS)	May or may not contain an operating system for functioning
Applications are alterable (programmable) by user (It is possible for the end user to re-install the Operating System, and add or remove user applications)	The firmware of the embedded system is pre-programmed and it is non-alterable by end-user (There may be exceptions for systems supporting OS kernel image flashing through special hardware settings)
Performance is the key deciding factor on the selection of the system. Always 'Faster is Better'	Application specific requirements (like performance, power requirements, memory usage etc) are the key deciding factors
Less/not at all tailored towards reduced operating power requirements, options for different levels of power management.	Highly tailored to take advantage of the power saving modes supported by hardware and Operating System
Response requirements are not time critical	For certain category of embedded systems like mission critical systems, the response time requirement is highly critical
Need not be deterministic in execution behavior	Execution behavior is deterministic for certain type of embedded systems like 'Hard Real Time' systems

Introduction to Embedded System

Classification of Embedded Systems:

- ❑ Based on Generation
- ❑ Based on Complexity & Performance Requirements
- ❑ Based on deterministic behavior
- ❑ Based on Triggering

Introduction to Embedded System

Embedded Systems - Classification based on Generation

First Generation: The early embedded systems built around 8bit microprocessors like 8085 and Z80 and 4bit microcontrollers

Second Generation: Embedded Systems built around 16bit microprocessors and 8 or 16bit microcontrollers, following the first generation embedded systems

Third Generation: Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs)

Fourth Generation: Embedded Systems built around System on Chips (SoCs), Re-configurable processors and multicore processors

Introduction to Embedded System

Embedded Systems - Classification based on Complexity & Performance

Small Scale: The early embedded systems built around 8bit microprocessors like 8085 and Z80 and 4bit microcontrollers

Medium Scale: Embedded Systems built around 16bit microprocessors and 8 or 16bit microcontrollers, following the first generation embedded systems

Large Scale/Complex: Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs)

Introduction to Embedded System

Embedded Systems - Classification based on Deterministic Behaviour.

- The classification based on these are applicable for "**Real Time**" systems.
- The application/task execution behavior for an embedded systems can be either deterministic or non-deterministic.
- Based on execution behavior , real time embedded systems are classified into Hard and Soft.

Introduction to Embedded System

Embedded Systems - Classification based on trigger.

- Embedded systems are Reactive in nature{like process control system in industrial control applications) can be classified based on the trigger.
- Reactive systems can be either event triggered or time triggered.

Introduction to Embedded System

Major Application Areas of Embedded Systems

- Consumer Electronics: Camcorders, Cameras etc.
- Household Appliances: Television, DVD players, Washing machine, Fridge, Microwave Oven etc.
- Home Automation and Security Systems: Air conditioners, sprinklers, Intruder detection alarms, Closed Circuit Television Cameras, Fire alarms etc.
- Automotive Industry: Anti-lock breaking systems (ABS), Engine Control, Ignition Systems, Automatic Navigation Systems etc.
- Telecom: Cellular Telephones, Telephone switches, Handset Multimedia Applications etc.
- Computer Peripherals: Printers, Scanners, Fax machines etc.
- Computer Networking Systems: Network Routers, Switches, Hubs, Firewalls etc.
- Health Care: Different Kinds of Scanners, EEG, ECG Machines etc.
- Measurement & Instrumentation: Digital multi meters, Digital CROs, Logic Analyzers PLC systems etc.
- Banking & Retail: Automatic Teller Machines (ATM) and Currency counters, Point of Sales (POS)
- Card Readers: Barcode, Smart Card Readers, Hand held Devices etc.

Introduction to Embedded System

Purpose of Embedded Systems

Each Embedded Systems is designed to serve the purpose of any one or a combination of the following tasks.

- Data Collection/Storage/Representation
- Data Communication
- Data (Signal) Processing
- Monitoring
- Control
- Application Specific User Interface

Introduction to Embedded System

Purpose of Embedded Systems – Data Collection/Storage/Representation

- ✓ Performs acquisition of data from the external world.
- ✓ The collected data can be either analog or digital
- ✓ Data collection is usually done for storage, analysis, manipulation and transmission
- ✓ The collected data may be stored directly in the system or may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly after giving a meaningful representation

Digital Camera for Image capturing/storage/display
Photo Courtesy of Casio -Model EXILIM ex-Z850
(www.casio.com)

Introduction to Embedded System

Purpose of Embedded Systems – Data Communication

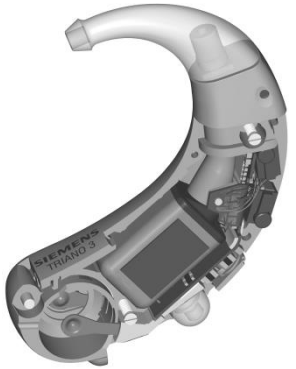
- ✓ Embedded Data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems
- ✓ Embedded Data communication systems are dedicated for data communication
- ✓ The data communication can happen through a wired interface (like Ethernet, RS-232C/USB/IEEE1394 etc) or wireless interface (like Wi-Fi, GSM,/GPRS, Bluetooth, ZigBee etc)
- ✓ Network hubs, Routers, switches, Modems etc are typical examples for dedicated data transmission embedded systems

Wireless Network Router for Data Communication
Photo Courtesy of Linksys (www.linksys.com).
A division of CISCO system

Introduction to Embedded System

Purpose of Embedded Systems – Data (Signal) Processing

- ✓ Embedded systems with Signal processing functionalities are employed in applications demanding signal processing like Speech coding, synthesis, audio video codec, transmission applications etc
- ✓ Computational intensive systems
- ✓ Employs Digital Signal Processors (DSPs)

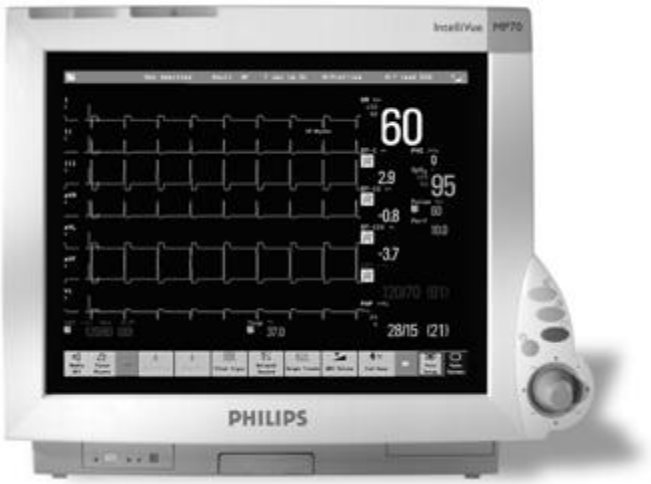


Digital hearing Aid employing Signal Processing Technique
Siemens TRIANO 3 Digital hearing aid;
Siemens Audiology Copyright © 2005

Introduction to Embedded System

Purpose of Embedded Systems – Monitoring

- ✓ Embedded systems coming under this category are specifically designed for monitoring purpose
- ✓ They are used for determining the state of some variables using input sensors
- ✓ They cannot impose control over variables.
- ✓ Electro Cardiogram (ECG) machine for monitoring the heart beat of a patient is a typical example for this
- ✓ The sensors used in ECG are the different Electrodes connected to the patient's body
- ✓ Measuring instruments like Digital CRO, Digital Multi meter, Logic Analyzer etc used in Control & Instrumentation applications are also examples of embedded systems for monitoring purpose



Patient Monitoring system
Photo courtesy of Philips Medical Systems
(www.medical.philips.com/)

Introduction to Embedded System

Purpose of Embedded Systems – Control

- ✓ Embedded systems with control functionalities are used for imposing control over some variables according to the changes in input variables
- ✓ Embedded system with control functionality contains both sensors and actuators
- ✓ Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable
- ✓ The actuators connected to the output port are controlled according to the changes in input variable to put an impact on the controlling variable to bring the controlled variable to the specified range
- ✓ Air conditioner for controlling room temperature is a typical example for embedded system with 'Control' functionality
- ✓ Air conditioner contains a room temperature sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature
- ✓ The air compressor unit acts as the actuator. The compressor is controlled according to the current room temperature and the desired temperature set by the end user.

Air Conditioner for controlling room temperature
Photo Courtesy of Electrolux Corporation
(www.electrolux.com/au)

Introduction to Embedded System

Purpose of Embedded Systems – Application Specific User Interface

- ✓ Embedded systems which are designed for a specific application
- ✓ Contains Application Specific User interface (rather than general standard UI) like key board, Display units etc
- ✓ Aimed at a specific target group of users
- ✓ Mobile handsets, Control units in industrial applications etc are examples for this

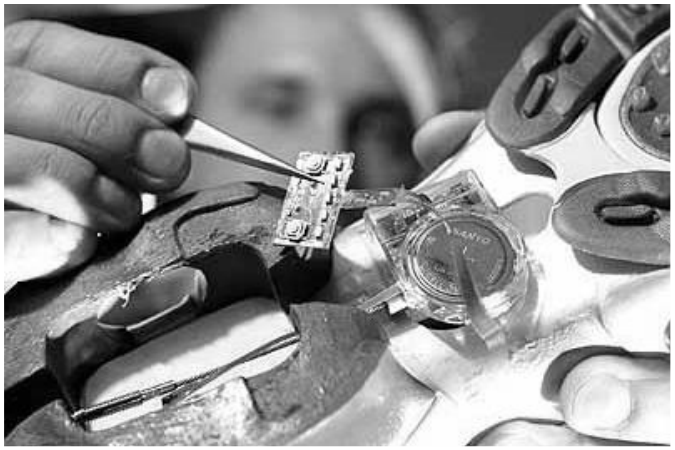
Patient Monitoring system
Photo courtesy of Philips Medical Systems
(www.medical.philips.com/)

Introduction to Embedded System

'Smart' running shoes from Adidas – The Innovative bonding of Life Style with Embedded Technology

- ✓ Shoe developed by Adidas, which constantly adapts its shock-absorbing characteristics to customize its value to the individual runner, depending on running style, pace, body weight, and running surface
- ✓ It contains sensors, actuators and a microprocessor unit which runs the algorithm for adapting the shock-absorbing characteristics of the shoe
- ✓ A 'Hall effect sensor' placed at the top of the "cushioning element" senses the compression and passes it to the Microprocessor
- ✓ A micro motor actuator controls the cushioning as per the commands from the MPU, based on the compression sensed by the 'Hall effect sensor'

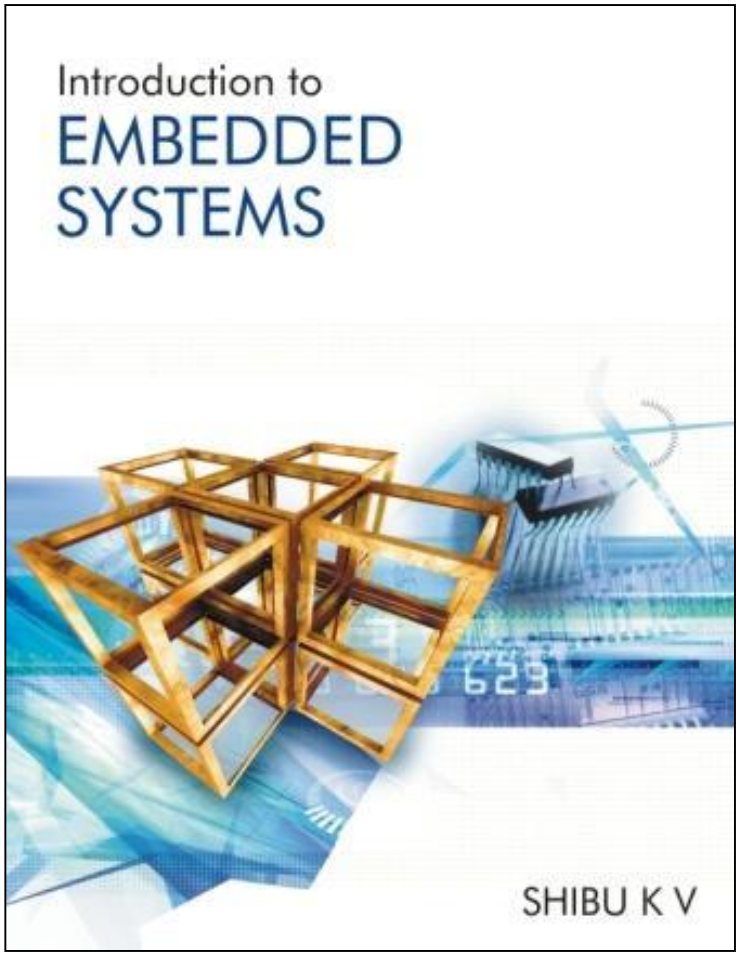
What an innovative bonding of Embedded Technology with Real life needs !!!☺



Electronics-enabled "Smart" running shoes from Adidas

Photo Courtesy of Adidas – Salomon AG (www.adidas.com)

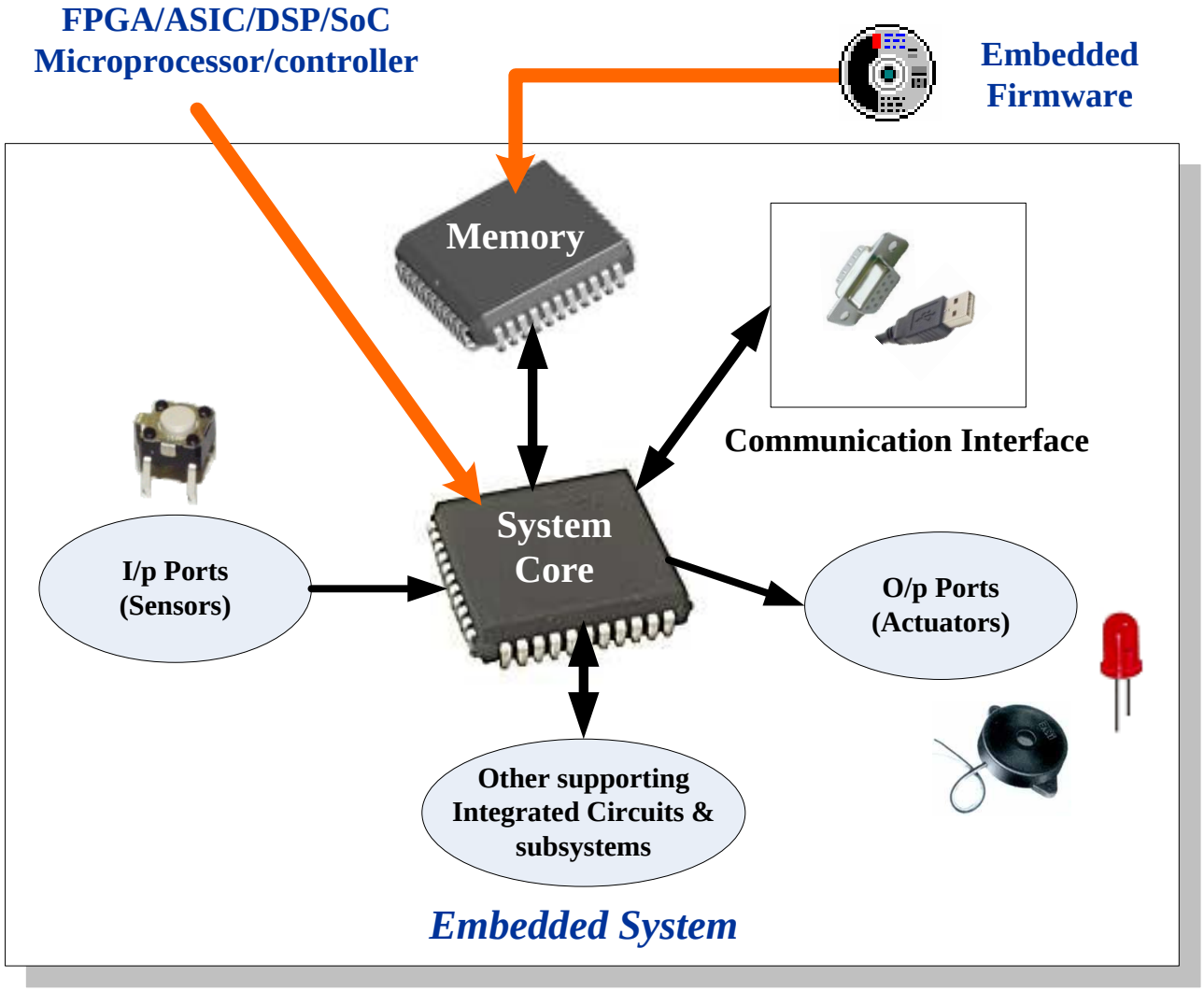
Mc Graw Hill Education



Module 4

The Typical Embedded System

The Typical Embedded System



The Typical Embedded System

The Core of the Embedded Systems

The core of the embedded system falls into any one of the following categories.

- **General Purpose and Domain Specific Processors**
  - Microprocessors
  - Microcontrollers
  - Digital Signal Processors
- **Programmable Logic Devices (PLDs)**
- **Application Specific Integrated Circuits (ASICs)**
- **Commercial off the shelf Components (COTS)**

The Typical Embedded System

Microprocessor

- ✓ A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions, which is specific to the manufacturer
- ✓ In general the CPU contains the Arithmetic and Logic Unit (ALU), Control Unit and Working registers
- ✓ Microprocessor is a dependant unit and it requires the combination of other hardware like Memory, Timer Unit, and Interrupt Controller etc for proper functioning.
- ✓ ntel claims the credit for developing the first Microprocessor unit Intel 4004, a 4 bit processor which was released in Nov 1971

The Typical Embedded System

General Purpose Processor (GPP) Vs Application Specific Instruction Set Processor (ASIP)

- ✓ General Purpose Processor or GPP is a processor designed for general computational tasks
- ✓ GPPs are produced in large volumes and targeting the general market. Due to the high volume production, 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)
- ✓ 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 fill the architectural spectrum between General Purpose Processors and Application Specific Integrated Circuits (ASICs)
- ✓ The need for an ASIP arises when the traditional general purpose processor are unable to meet the increasing application needs
- ✓ Some Microcontrollers (like Automotive AVR, USB AVR from Atmel), System on Chips, Digital Signal Processors etc are examples of Application Specific Instruction Set Processors (ASIPs)
- ✓ ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory

The Typical Embedded System

Microcontroller

- ✓ A highly integrated silicon chip containing 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
- ✓ Microcontroller can be general purpose (like Intel 8051, designed for generic applications and domains) or application specific (Like Automotive AVR from Atmel Corporation. Designed specifically for automotive applications)
- ✓ Since a microcontroller contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in place of microprocessors
- ✓ Microcontrollers are cheap, cost effective and are readily available in the market
- ✓ Texas Instruments TMS 1000 is considered as the world's first microcontroller

The Typical Embedded System

Microprocessor Vs Microcontroller

Microprocessor	Microcontroller
A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions	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
It is a dependent unit. It requires the combination of other chips like Timers, Program and data memory chips, Interrupt controllers etc for functioning	It is a self contained unit and it doesn't require external Interrupt Controller, Timer, UART etc for its functioning
Most of the time general purpose in design and operation	Mostly application oriented or domain specific
Doesn't contain a built in I/O port. The I/O Port functionality needs to be implemented with the help of external Programmable Peripheral Interface Chips like 8255	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit Port or as individual port pins
Targeted for high end market where performance is important	Targeted for embedded market where performance is not so critical (At present this demarcation is invalid)
Limited power saving options compared to microcontrollers	Includes lot of power saving features

The Typical Embedded System

Digital Signal Processors (DSPs)

- ✓ 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
- ✓ 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
- ✓ 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 key units
  - ✓ Program Memory
  - ✓ Data Memory
  - ✓ Computational Engine
  - ✓ I/O Unit
- ✓ Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed

The Typical Embedded System

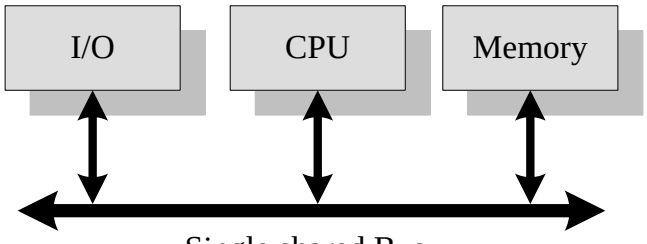
RISC V/s CISC Processors/Controllers

RISC	CISC
Lesser no. of instructions	Greater no. of Instructions
Instruction Pipelining and increased execution speed	Generally no instruction pipelining feature
Orthogonal Instruction Set (Allows each instruction to operate on any register and use any addressing mode)	Non Orthogonal Instruction Set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction specific)
Operations are performed on registers only, the only memory operations are load and store	Operations are performed on registers or memory depending on the instruction
Large number of registers are available	Limited no. of general purpose registers
Programmer needs to write more code to execute a task since the instructions are simpler ones	Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Single, Fixed length Instructions	Variable length Instructions
Less Silicon usage and pin count	More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
With Harvard Architecture	Can be Harvard or Von-Neumann Architecture

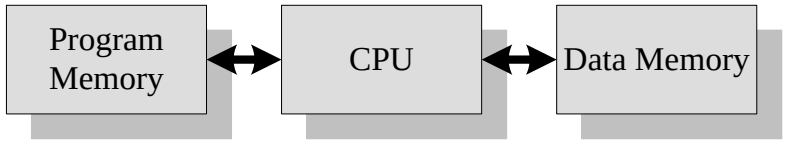
The Typical Embedded System

Harvard V/s 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
- ✓ 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 the 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")



Von-Neumann Architecture



Harvard Architecture

The Typical Embedded System

Harvard V/s Von-Neumann Processor/Controller Architecture

Harvard Architecture	Von-Neumann Architecture
Separate buses for Instruction and Data fetching	Single shared bus for Instruction and Data fetching
Easier to Pipeline, so high performance can be achieved	Low performance Compared to Harvard Architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in same chip, chances for accidental corruption of program memory

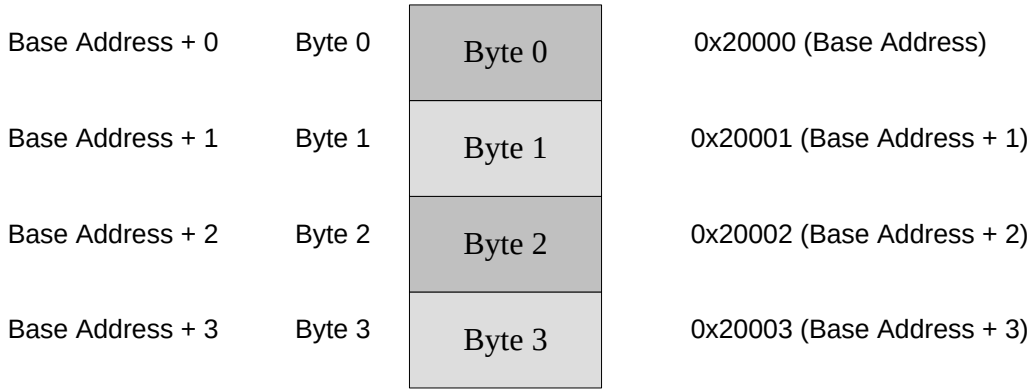
The Typical Embedded System

Big-endian V/s Little-endian processors

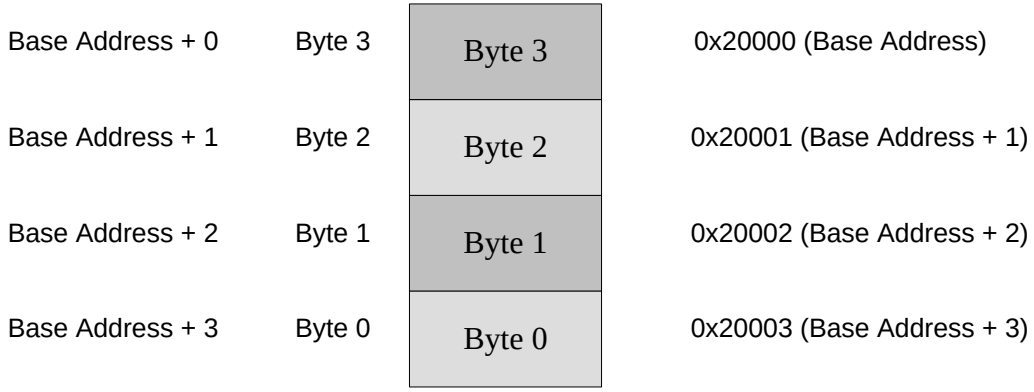
- ✓ Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system (Processors whose word size is greater than one byte). Suppose the word length is two byte then data can be stored in memory in two different ways
  - ✓ Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory
  - ✓ Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory
- ✓ ***Little-endian*** means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first)
- ✓ ***Big-endian*** means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)

The Typical Embedded System

Big-endian V/s Little-endian processors



Little-endian Operation

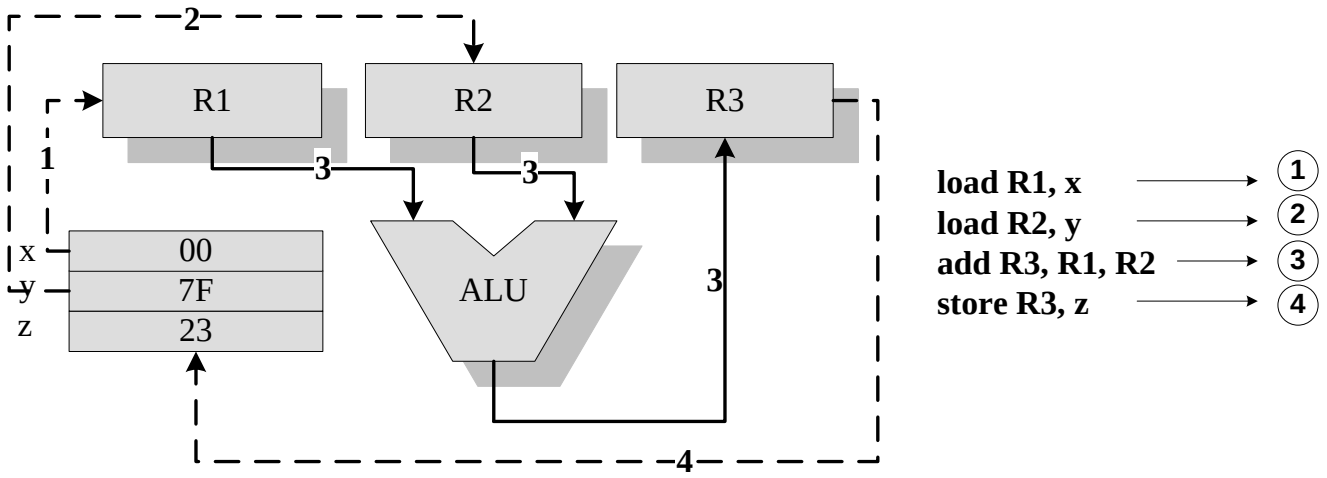


Big-endian Operation

The Typical Embedded System

Load Store Operation & Instruction Pipelining

The RISC processor instruction set is orthogonal and 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

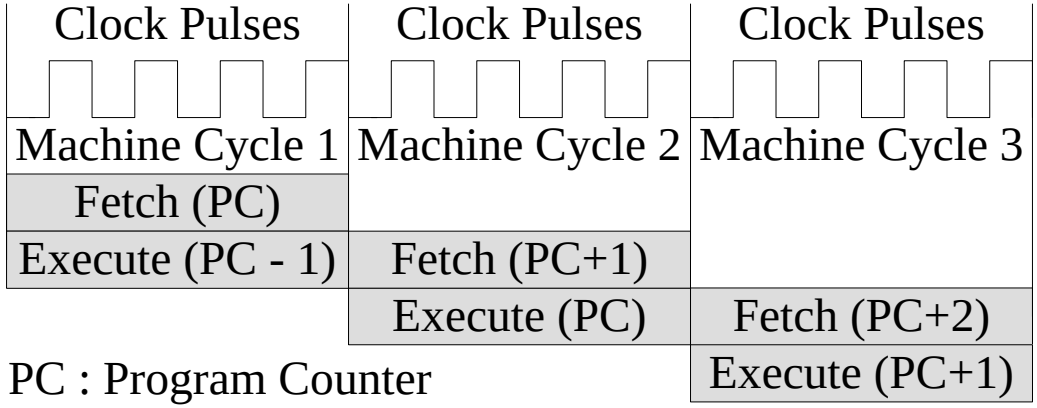


Load Store Operation

The Typical Embedded System

Instruction Pipelining

- ✓ The conventional instruction execution by the processor follows the fetch-decode-execute sequence
- ✓ The 'fetch' part fetches the instruction from program memory or code memory and the decode part decodes the instruction to generate the necessary control signals
- ✓ 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
- ✓ During the decode operation the memory address bus is available and if it possible to effectively utilize it for an instruction fetch, the processing speed can be increased
- ✓ In its simplest form instruction pipelining refers to the overlapped execution of instructions



The Single stage pipelining concept

The Typical Embedded System

Application Specific Integrated Circuit (ASIC)

- ✓ A microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips.
- ✓ ASIC integrates several functions into a single chip and thereby reduces the system development cost
- ✓ Most of the ASICs are proprietary products. As a single chip, ASIC consumes very small area in the total system and thereby helps in the design of smaller systems with high capabilities/functionalities.
- ✓ 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
- ✓ Fabrication of ASICs requires a non-refundable initial investment (Non Recurring Engineering (NRE) charges) for the process technology and configuration expenses
- ✓ If the Non-Recurring Engineering Charges (NRE) is born 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)

The Typical Embedded System

Programmable Logic Devices (PLDs)

- ✓ 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. The circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed
- ✓ 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
- ✓ Designers can use inexpensive software tools to quickly develop, simulate, and test their logic designs in PLD based design. The design can be quickly programmed into a device, and immediately tested in a live circuit
- ✓ PLDs are based on re-writable memory technology and the device is reprogrammed to change the design

The Typical Embedded System

Programmable Logic Devices (PLDs) – CPLDs and FPGA

- ✓ Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs) are the two major types of programmable logic devices
- ✓ FPGAs offer the highest amount of logic density, the most features, and the highest performance.
- ✓ These advanced FPGA devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies
- ✓ FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing
- ✓ CPLDs, by contrast, offer much smaller amounts of logic - up to about 10,000 gates
- ✓ CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications
- ✓ CPLDs such as the Xilinx **CoolRunner** series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants

The Typical Embedded System

Commercial off the Shelf Component (COTS)

- ✓ 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
- ✓ Typical examples for the COTS hardware unit are Remote Controlled Toy Car control unit including the RF Circuitry part, High performance, high frequency microwave electronics (2 to 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
- ✓ A COTS component in turn contains a General Purpose Processor (GPP) or Application Specific Instruction Set Processor (ASIP) or Application Specific Integrated Chip (ASIC)/Application Specific Standard Product (ASSP) or Programmable Logic Device (PLD)
- ✓ The major advantage of using COTS is that they are readily available in the market, cheap and a developer can cut down his/her development time to a great extend

The Typical Embedded System

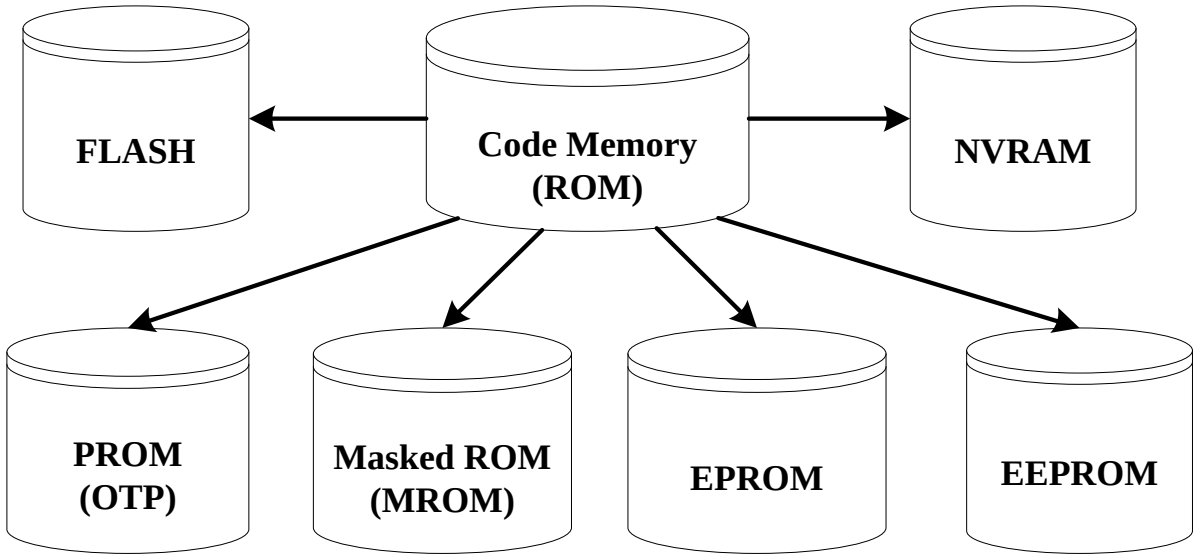
Memory

- ✓ Memory is an important part of an embedded system. The memory used in embedded system can be either Program Storage Memory (ROM) or Data memory (RAM)
- ✓ Certain Embedded processors/controllers contain built in program memory and data memory and this memory is known as on-chip memory

The Typical Embedded System

Memory – Program Storage Memory

- ✓ Stores the program instructions
- ✓ 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 Typical Embedded System

Memory – Program Storage Memory – Masked ROM (MROM)

- ✓ One-time programmable memory. Uses hardwired technology for storing data. The device is factory programmed by masking and metallization process according to the data provided by the end user
- ✓ The primary advantage of MROM 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
  - ✓ Creation of an enhancement or depletion mode transistor through channel implant
  - ✓ By creating the memory cell either using a standard transistor or a high threshold transistor. In the high threshold mode, the supply voltage required to turn ON the transistor is above the normal ROM IC operating voltage. This ensures that the transistor is always off and the memory cell stores always logic 0.
- ✓ 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

The Typical Embedded System

Memory – Program Storage Memory – Programmable Read Only Memory (PROM) / (OTP)

- ✓ Unlike MROM it is not pre-programmed by the manufacturer
- ✓ PROM/OTP 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 a logic “1” where as 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

The Typical Embedded System

Memory – Program Storage Memory – Erasable Programmable Read Only Memory (EPROM)

- ✓ Erasable Programmable Read Only (EPROM) memory gives the flexibility to re-program 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 Ultra violet rays for a fixed duration, the entire memory will be erased
- ✓ Even though the EPROM chip is flexible in terms of re-programmability, it needs to be taken out of the circuit board and needs to be put in a UV eraser device for 20 to 30 minutes

The Typical Embedded System

Memory – Program Storage Memory – Electrically Erasable Programmable Read Only Memory (EEPROM)

- ✓ Erasable Programmable Read Only (EPROM) memory gives the flexibility to re-program the same chip using electrical signals
- ✓ 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 within the 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 when compared with the standard ROM (A few kilobytes).

The Typical Embedded System

Memory – Program Storage Memory – FLASH

- ✓ FLASH memory is a variation of EEPROM technology
- ✓ It combines the re-programmability of EEPROM and the high capacity 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 done at sector level or page level without affecting the other sectors or pages
- ✓ Each sector/page should be erased before re-programming

The Typical Embedded System

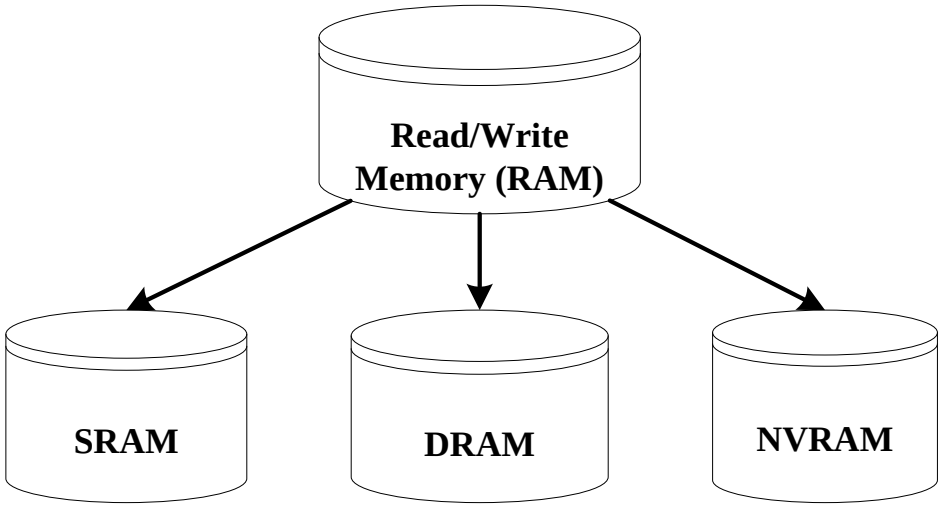
Memory – RAM – Non Volatile RAM (NVRAM)

- ✓ 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
- ✓ NVRAM is used for the non volatile storage of results of operations or for setting up of flags etc
- ✓ The life span of NVRAM is expected to be around 10 years
- ✓ DS1744 from Maxim/Dallas is an example for 32KB NVRAM

The Typical Embedded System

Memory – Read-Write Memory/Random Access Memory (RAM)

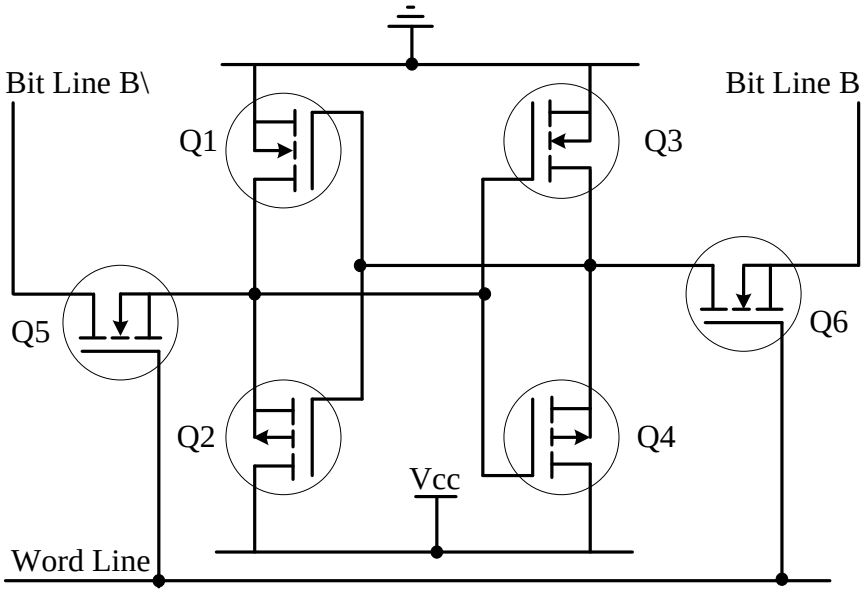
- ✓ RAM is the data memory or working memory of the controller/processor
- ✓ RAM is volatile, meaning when the power is turned off, all the contents are destroyed
- ✓ RAM is a direct access memory, meaning we can access the desired memory location directly without the need for traversing through the entire memory locations to reach the desired memory position (i.e. Random Access of memory location)



The Typical Embedded System

Memory – RAM – Static RAM (SRAM)

- ✓ Static RAM stores data in the form of Voltage. They are made up of flip-flops
- ✓ In typical implementation, an SRAM cell (bit) is realized using 6 transistors (or 6 MOSFETs). Four of the transistors are used for building the latch (flip-flop) part of the memory cell and 2 for controlling the access.
- ✓ Static RAM is the fastest form of RAM available. SRAM is fast in operation due to its resistive networking and switching capabilities

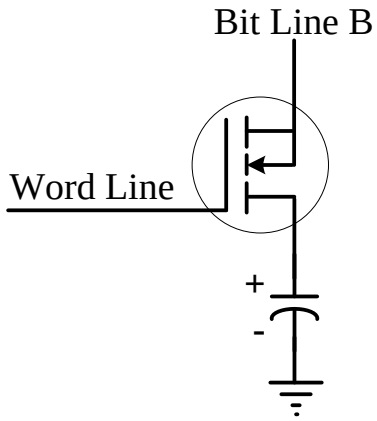


SRAM cell implementation

The Typical Embedded System

Memory – RAM – 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



DRAM cell implementation

The Typical Embedded System

Memory – RAM – SRAM Vs DRAM)

SRAM Cell	DRAM Cell
Made up of 6 CMOS transistors (MOSFET)	Made up of a MOSFET and a capacitor
Doesn't Require refreshing	Requires refreshing
Low capacity (Less dense)	High Capacity (Highly dense)
More expensive	Less Expensive
Fast in operation. Typical access time is 10ns	Slow in operation due to refresh requirements. Typical access time is 60ns. Write operation is faster than read operation.

The Typical Embedded System

Sensors & Actuators

Sensor:

A transducer device which converts energy from one form to another for any measurement or control purpose. Sensors acts as input device

Eg. Hall Effect Sensor which measures the distance between the cushion and magnet in the Smart Running shoes from adidas

Actuator:

A form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device

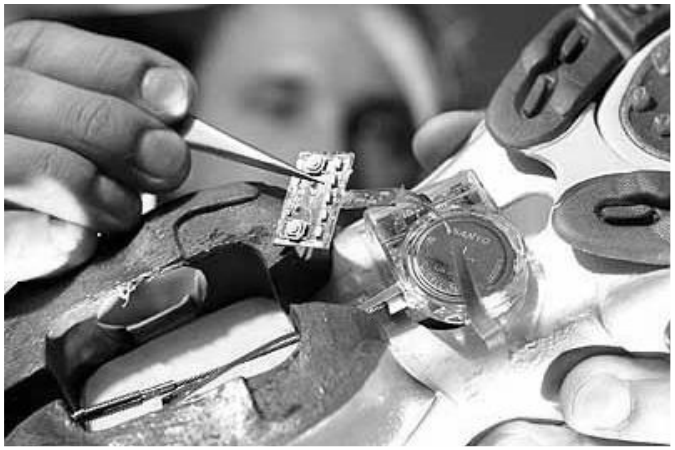
Eg. Micro motor actuator which adjusts the position of the cushioning element in the Smart Running shoes from adidas

Introduction to Embedded System

'Smart' running shoes from Adidas – The Innovative bonding of Life Style with Embedded Technology

- ✓ Shoe developed by Adidas, which constantly adapts its shock-absorbing characteristics to customize its value to the individual runner, depending on running style, pace, body weight, and running surface
- ✓ It contains sensors, actuators and a microprocessor unit which runs the algorithm for adapting the shock-absorbing characteristics of the shoe
- ✓ A 'Hall effect sensor' placed at the top of the "cushioning element" senses the compression and passes it to the Microprocessor
- ✓ A micro motor actuator controls the cushioning as per the commands from the MPU, based on the compression sensed by the 'Hall effect sensor'

What an innovative bonding of Embedded Technology with Real life needs !!!☺



Electronics-enabled "Smart" running shoes from Adidas
Photo Courtesy of Adidas – Salomon AG (www.adidas.com)

The Typical Embedded System

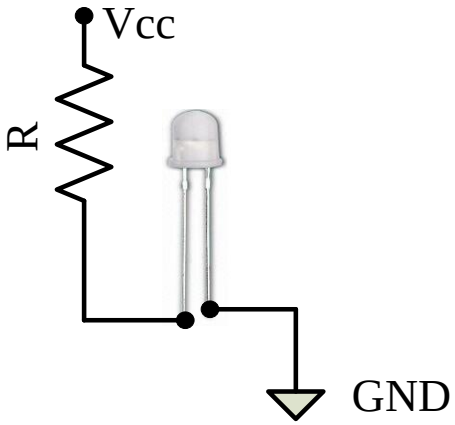
The I/O Subsystem

- ✓ The I/O subsystem of the embedded system facilitates the interaction of the embedded system with external world
- ✓ The interaction happens through the sensors and actuators connected to the Input and output ports respectively of the embedded system
- ✓ The sensors may not be directly interfaced to the Input ports, instead they may be interfaced through signal conditioning and translating systems like ADC, Optocouplers etc

The Typical Embedded System

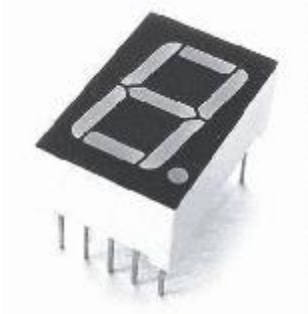
The I/O Subsystem – I/O Devices - Light Emitting Diode (LED)

- ✓ Light Emitting Diode (LED) is an 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
- ✓ LED 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
- ✓ The current flowing through the LED must limited to a value below the maximum current that it can conduct. A resister is used in series between the power supply and the resistor to limit the current through the LED

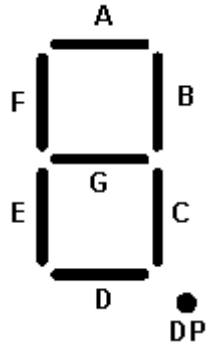


The Typical Embedded System

The I/O Subsystem – I/O Devices – 7-Segment LED Display



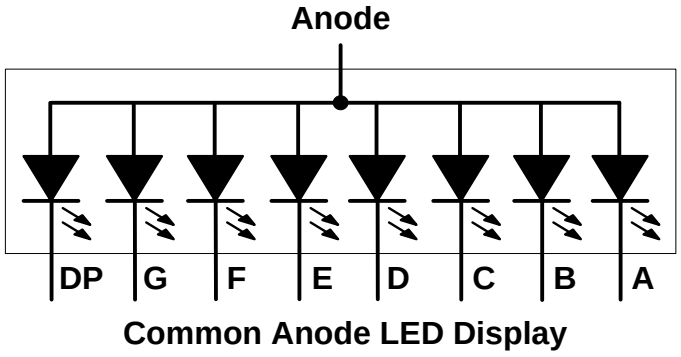
- ✓ The 7 – segment LED display is an output device for displaying alpha numeric characters
- ✓ It contains 8 light-emitting diode (LED) segments arranged in a special form. Out of the 8 LED segments, 7 are used for displaying alpha numeric characters
- ✓ The LED segments are named A to G and the decimal point LED segment is named as DP
- ✓ The LED Segments A to G and DP should be lit accordingly to display numbers and characters
- ✓ The 7 – segment LED displays are available in two different configurations, namely; Common anode and Common cathode
- ✓ In the Common anode configuration, the anodes of the 8 segments are connected commonly whereas in the Common cathode configuration, the 8 LED segments share a common cathode line

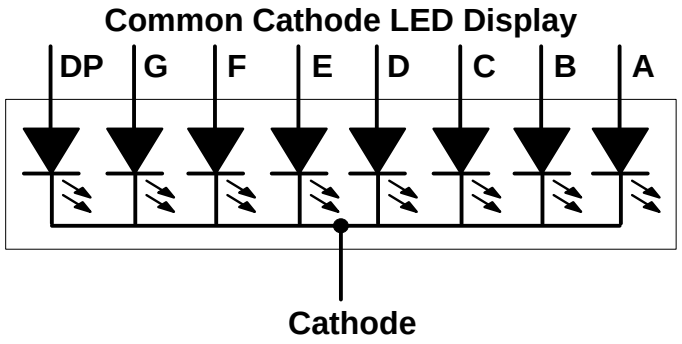


The Typical Embedded System

The I/O Subsystem – I/O Devices – 7-Segment LED Display

- ✓ Based on the configuration of the 7 – segment LED unit, the LED segment anode or cathode is connected to the Port of the processor/controller in the order 'A' segment to the Least significant port Pin and DP segment to the most significant Port Pin.
- ✓ The current flow through each of the LED segments should be limited to the maximum value supported by the LED display unit
- ✓ The typical value for the current falls within the range of 20mA
- ✓ The current through each segment can be limited by connecting a current limiting resistor to the anode or cathode of each segment

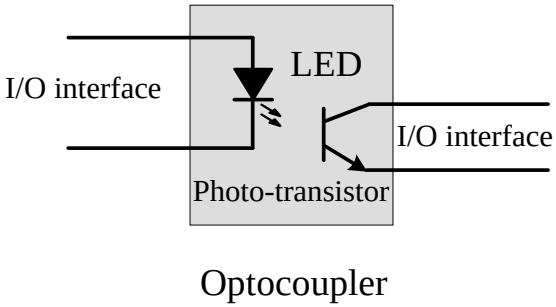




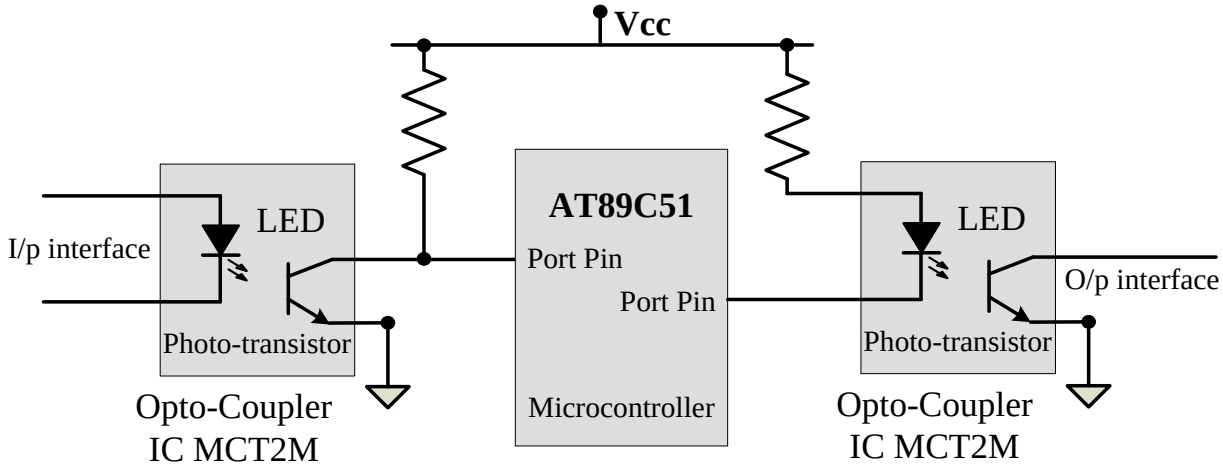
The Typical Embedded System

The I/O Subsystem – I/O Devices – 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)
- ✓ In electronic circuits, optocoupler is used for suppressing interference in data communication, circuit isolation, High voltage separation, simultaneous separation and intensification signal etc
- ✓ Optocouplers can be used in either input circuits or in output circuits



Optocoupler

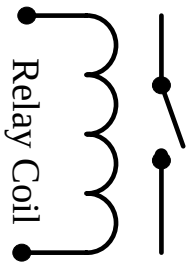


Optocoupler in input and output circuit

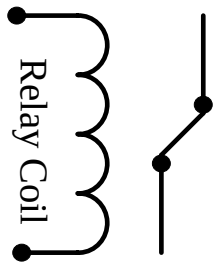
The Typical Embedded System

The I/O Subsystem – I/O Devices – Relay

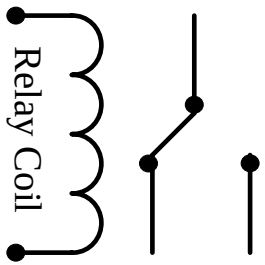
- ✓ An electro mechanical device which acts as dynamic path selectors for signals and power
- ✓ The 'Relay' unit contains a relay coil made up of insulated wire on a metal core and a metal armature with one or more contacts.
- ✓ 'Relay' works on electromagnetic principle. When a voltage is applied to 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



Single Pole Single Throw Normally Open



Single Pole Single Throw Normally Closed

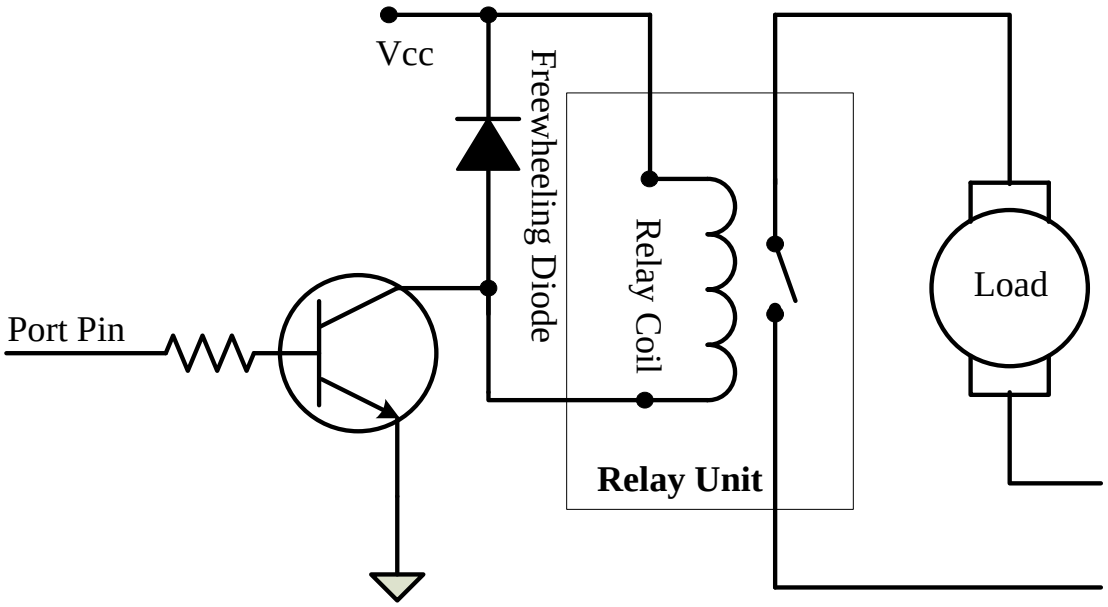


Single Pole Double Throw

The Typical Embedded System

The I/O Subsystem – I/O Devices – Relay Driver Circuit

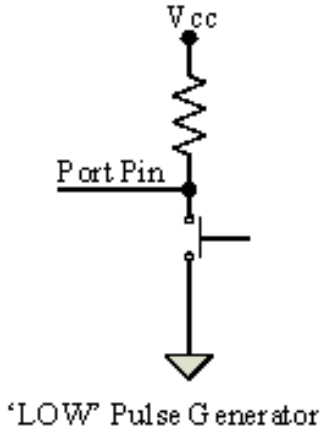
- ✓ The Relay is normally controlled using a relay driver circuit connected to the port pin of the processor/controller
- ✓ A transistor can be used as the relay driver. The transistor can be selected depending on the relay driving current requirements

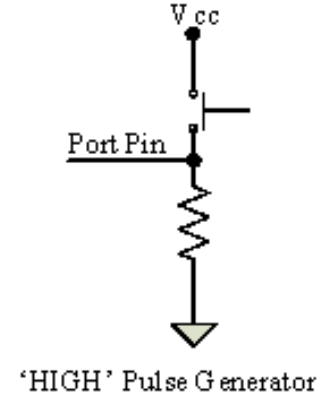


The Typical Embedded System

The I/O Subsystem – I/O Devices – 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'
- ✓ The switch is normally in the open state and it makes a circuit contact when it is pushed or pressed in the 'Push to Make' configuration
- ✓ 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





The Typical Embedded System

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; namely; Device/board level communication interface (Onboard Communication Interface) and Product level communication interface (External Communication Interface)
- ✓ Embedded product is a combination of different types of components (chips/devices) arranged on a Printed Circuit Board (PCB). The communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface)
- ✓ Serial interfaces like I2C, SPI, UART, 1-Wire etc and Parallel bus interface are examples of 'Onboard Communication Interface'
- ✓ 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 wired media or wireless media and it can be a serial or parallel interface. Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS etc are examples for wireless communication interface
- ✓ RS-232C/RS-422/RS 485, USB, Ethernet (TCP-IP), IEEE 1394 port, Parallel port, CF-II Slot, SDIO, PCMCIA etc are examples for wired interfaces

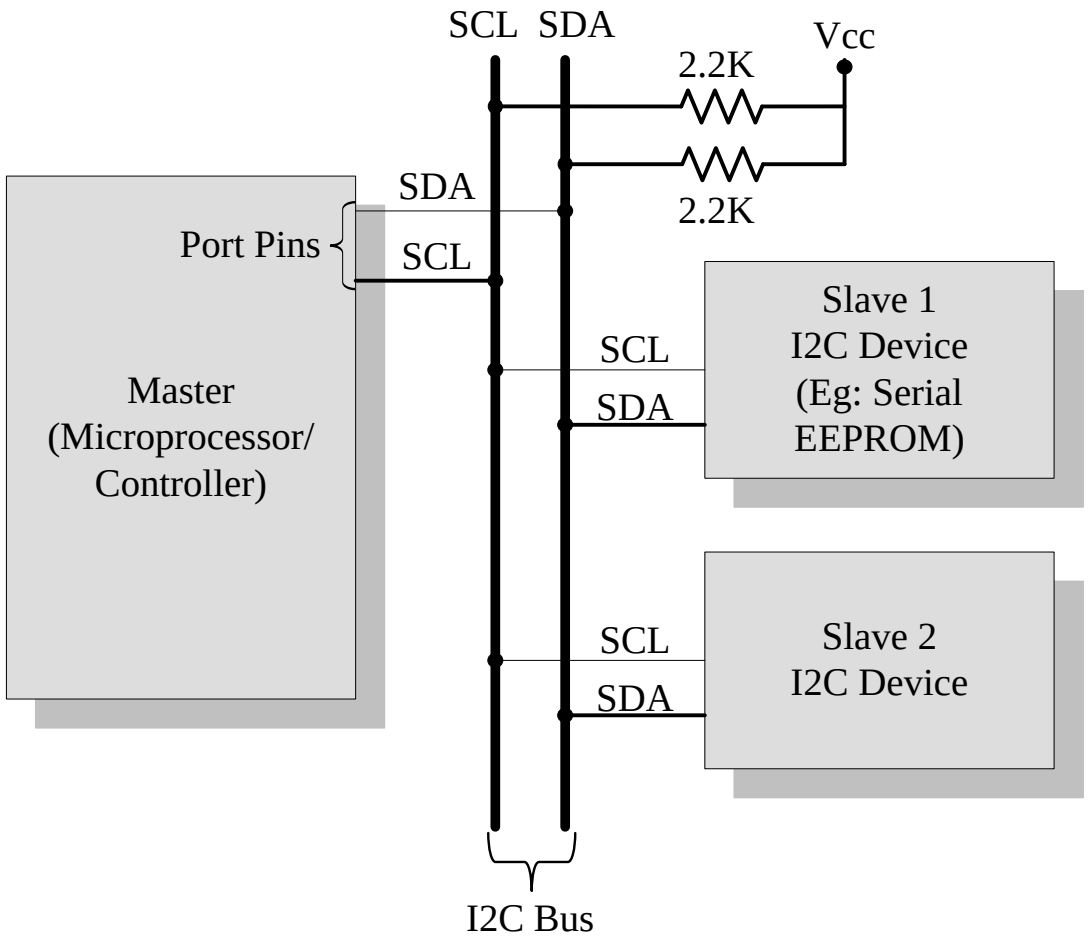
The Typical Embedded System

On-board Communication Interface - I2C

- ✓ Inter Integrated Circuit Bus (I2C - Pronounced 'I square C') is a synchronous bi-directional 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 1980's. 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 is comprised of two bus lines, namely; Serial Clock – SCL and Serial Data – SDA. SCL line is responsible for generating synchronization clock pulses and SDA is responsible for transmitting the serial data across devices.
- ✓ I2C bus is a shared bus system to which many number of I2C devices can be connected. 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 Typical Embedded System

On-board Communication Interface - I2C



The Typical Embedded System

On-board Communication Interface - I2C

The sequence of operation for communicating with an I2C slave device is:

1. Master device pulls the clock line (SCL) of the bus to 'HIGH'
2. 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. Master 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. Master sends the Read or Write bit (Bit value = 1 Read Operation; Bit value = 0 Write Operation) according to the requirement
5. Master 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, master sends the 8bit 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. Master 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. Master terminates the transfer by pulling the SDA line 'HIGH' when the clock line SCL is at logic 'HIGH' (Indicating the 'STOP' condition)

The Typical Embedded System

On-board Communication Interface – Serial Peripheral Interface (SPI) Bus

The Serial Peripheral Interface Bus (SPI) is a synchronous bi-directional full duplex four wire serial interface bus. The concept of SPI is 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)

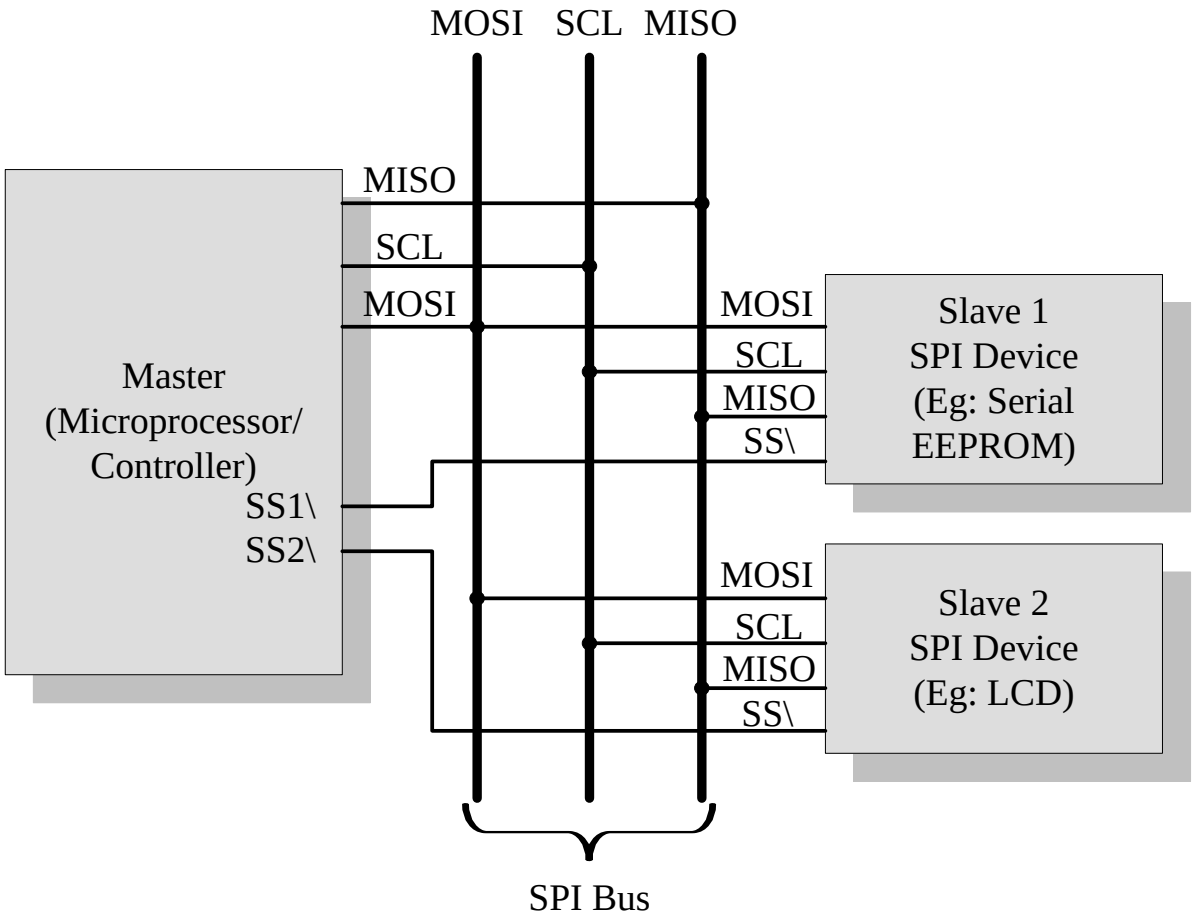
Master In Slave Out (MISO): Signal line carrying the data from slave to master device. It is also known as Slave Output (SO/SDO)

Serial Clock (SCLK): Signal line carrying the clock signals

Slave Select (SS): Signal line for slave device select. It is an active low signal

The Typical Embedded System

On-board Communication Interface – Serial Peripheral Interface (SPI) Bus



The Typical Embedded System

On-board Communication Interface – Serial Peripheral Interface (SPI) Bus

- ✓ The master device is responsible for generating the clock signal. Master device 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
- ✓ The serial data transmission through SPI Bus is fully configurable. SPI devices contain certain set of registers for holding these configurations. The Serial Peripheral Control Register holds the various configuration parameters like master/slave selection for the device, baudrate selection for communication, clock signal control etc. The status register holds the status of various conditions for transmission and reception.
- ✓ 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

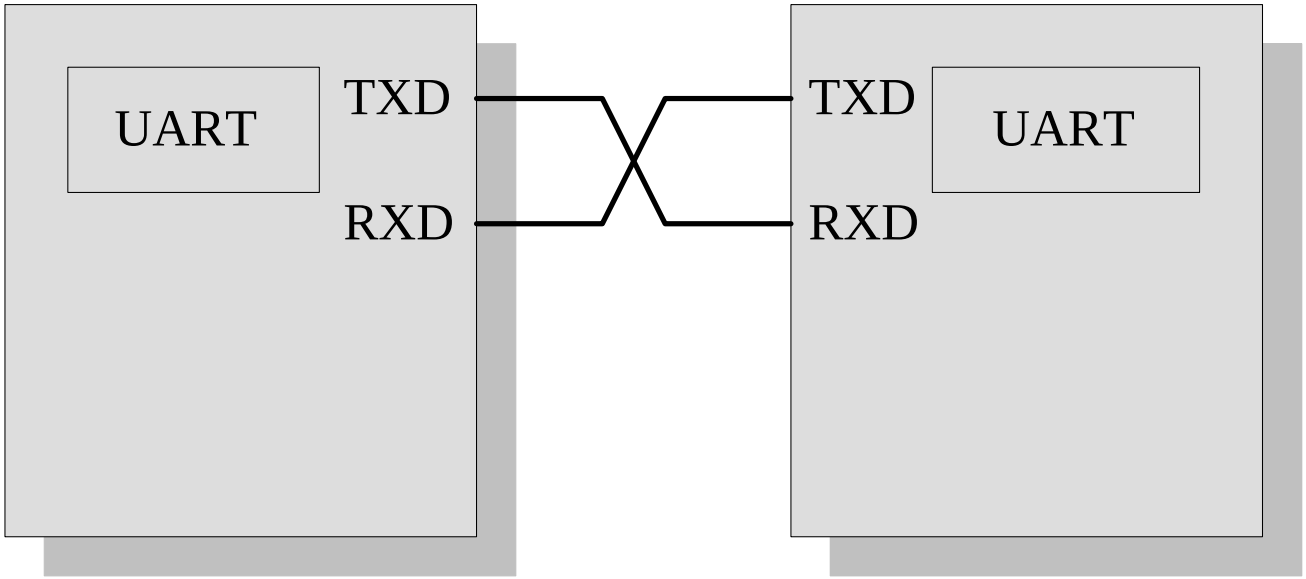
The Typical Embedded System

On-board Communication Interface – Universal Asynchronous Receiver Transmitter (UART)

- ✓ Universal Asynchronous Receiver Transmitter (UART) based data transmission is an asynchronous form of serial data transmission
- ✓ The serial communication settings (Baudrate, No. of bits per byte, parity, No. of start bits and stop bit and flow control) for both transmitter and receiver should be set as identical
- ✓ The start and stop of communication is indicated through inserting special bits in the data stream
- ✓ While sending a byte of data, a start bit is added first and a stop bit is added at the end of the bit stream. The 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 baudrate settings
- ✓ If parity is enabled for communication, the UART of the transmitting device adds a parity bit
- ✓ 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

The Typical Embedded System

On-board Communication Interface – Universal Asynchronous Receiver Transmitter (UART)



TXD: Transmitter Line
RXD: Receiver Line

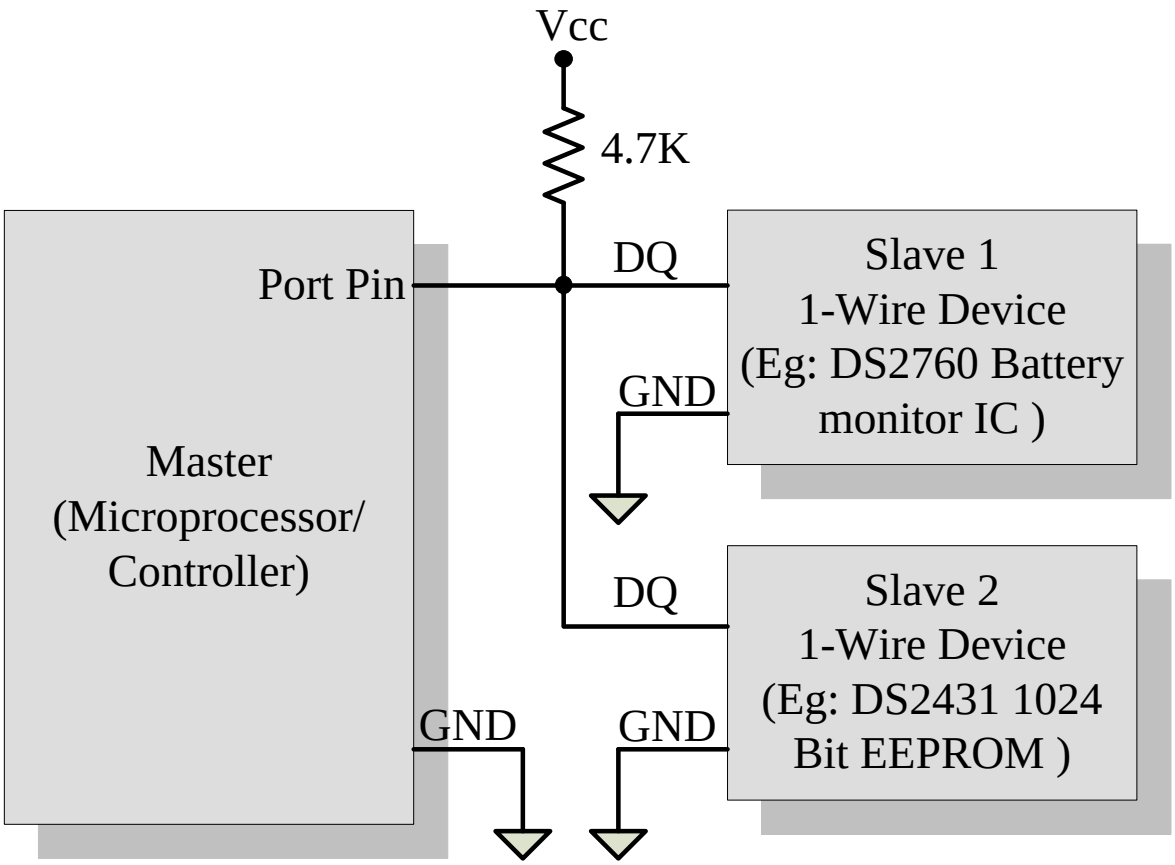
The Typical Embedded System

On-board Communication Interface – 1-Wire Interface

- ✓ An asynchronous half-duplex communication protocol developed by Maxim Dallas Semiconductor (http://www.maxim-ic.com)
- ✓ 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
- ✓ The 1-Wire interface supports a single master and one or more slave devices on the bus
- ✓ The 1-Wire is capable of carrying power to the slave device apart from carrying the signals. Slave devices incorporate internal capacitor to generate power to operate the device from the 1-Wire
- ✓ 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 in 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 Typical Embedded System

On-board Communication Interface – 1-Wire Interface



The Typical Embedded System

On-board Communication Interface – 1-Wire Interface

The sequence of operation for communicating with a 1-Wire slave device is:

1. Master device sends a ‘Reset’ pulse on the 1-Wire bus.
2. Slave device(s) present on the bus respond with a ‘Presence’ pulse.
3. 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. Master device sends a read/write function command to read/write the internal memory or register of the slave device.
5. Master initiates a Read data /Write data from the device or to the device

The Typical Embedded System

On-board Communication Interface – 1-Wire Interface

- ✓ All communication over the 1-Wire bus is master initiated
- ✓ The communication over the 1-Wire bus is divided into timeslots of 60 microseconds
- ✓ The 'Reset' pulse occupies 8 time slots. For starting a communication, the master asserts the reset pulse by pulling the 1-Wire bus 'LOW' for at least 8 time slots (480µs)
- ✓ If a 'Slave' device is present on the bus and is ready for communication it should respond to the master with a 'Presence' pulse, within 60µs of the release of the 'Reset' pulse by the master
- ✓ The slave device(s) responds with a 'Presence' pulse by pulling the 1-Wire bus 'LOW' for a minimum of 1 time slot (60µs)
- ✓ For writing a bit value of 1 on the 1-Wire bus, the bus master pulls the bus for 1 to 15µs and then releases the bus for the rest of the time slot
- ✓ A bit value of '0' is written on the bus by master pulling the bus for a minimum of 1 time slot (60µs) and a maximum of 2 time slots (120µs)
- ✓ To Read a bit from the slave device, the master pulls the bus 'LOW' for 1 to 15µs
- ✓ If the slave wants to send a bit value '1' in response to the read request from the slave, it simply releases the bus for the rest of the time slot
- ✓ If the slave wants to send a bit value '0', it pulls the bus 'LOW' for the rest of the time slot.

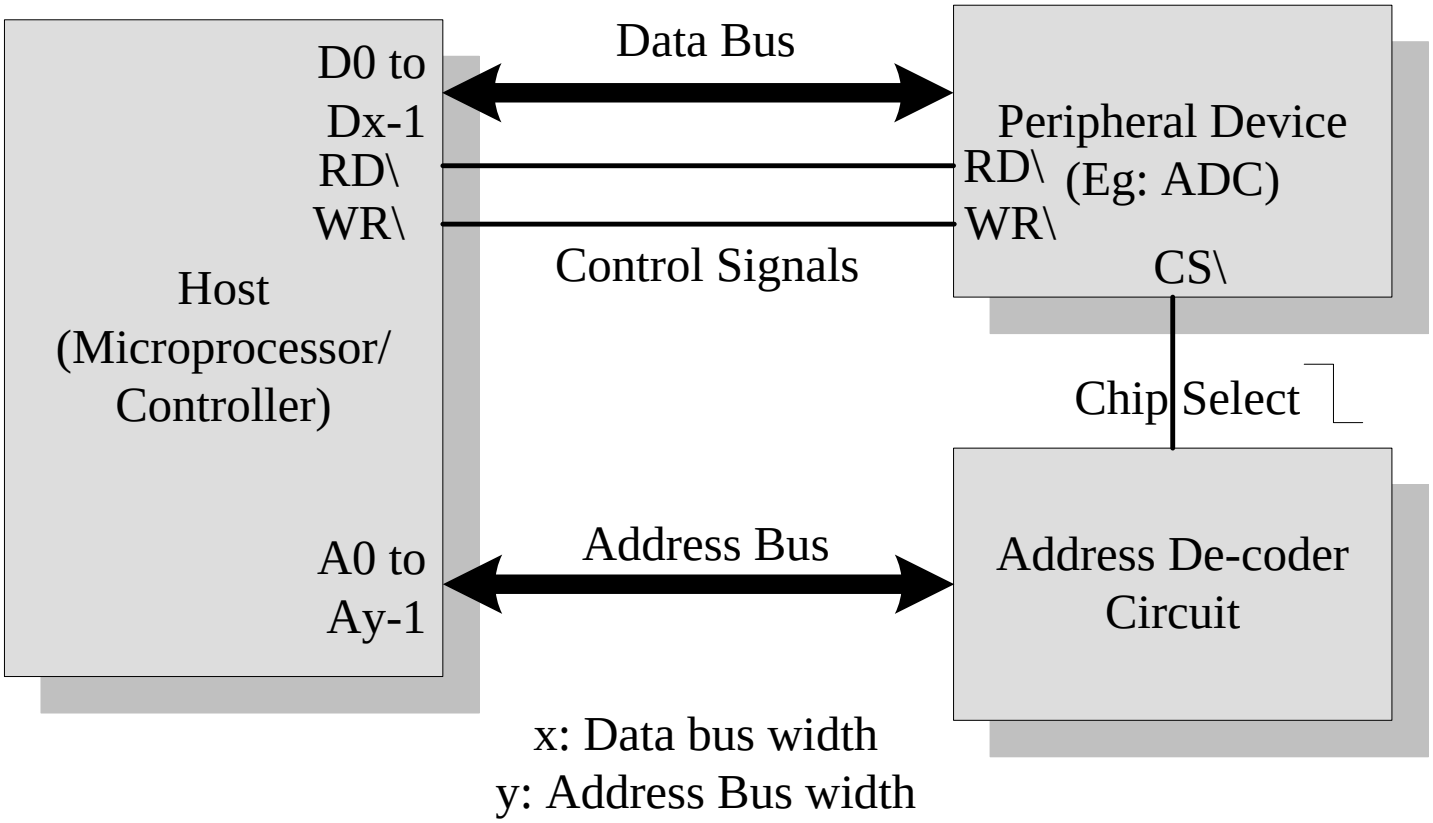
The Typical Embedded System

On-board Communication Interface – Parallel Interface

- ✓ 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 Typical Embedded System

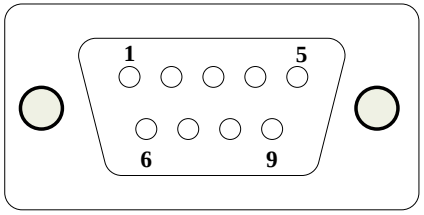
On-board Communication Interface – Parallel Interface



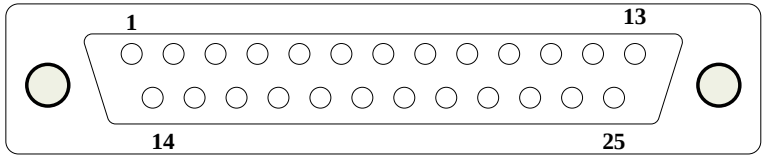
The Typical Embedded System

External Communication Interface – 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
- ✓ 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 RS232 use the EIA standard for bit transmission. As per 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 RS232 interface define 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, namely; DB-9: 9-Pin connector and DB-25: 25-Pin connector.



DB-9



DB-25

The Typical Embedded System

External Communication Interface – RS-232 C & RS-485

Pin Name	Pin No: (For DB-9 Connector)	Description
TXD	3	Transmit Pin. Used for Transmitting Serial Data
RXD	2	Receive Pin. Used for Receiving serial Data
RTS	7	Request to send.
CTS	8	Clear To Send
DSR	6	Data Set ready
GND	5	Signal Ground
DCD	1	Data Carrier Detect
DTR	4	Data Terminal Ready
RI	9	Ring Indicator

The Typical Embedded System

External Communication Interface – RS-232 C & RS-485

- ✓ RS-232 is a point-to-point communication interface and the devices involved in RS-232 communication are called 'Data Terminal Equipment (DTE)' and 'Data Communication Equipment (DCE)'
- ✓ If no data flow control is required, only TXD and RXD signal lines and ground line (GND) are required for data transmission and reception. The RXD pin of DCE should be connected to the TXD pin of DTE and vice versa for proper data transmission.
- ✓ If hardware data flow control is required for serial transmission, various control signal lines of the RS-232 connection are used appropriately. The control signals are implemented mainly for modem communication and some of them may be irrelevant for other type of devices
- ✓ The Request To Send (RTS) and Clear To Send (CTS) signals co-ordinate the communication between DTE and DCE. Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line
- ✓ The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data. The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link. DTR should be in the activated state before the activation of DSR
- ✓ The Data Carrier Detect (DCD) is used by the DCE to indicate the DTE that a good signal is being received
- ✓ Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line

The Typical Embedded System

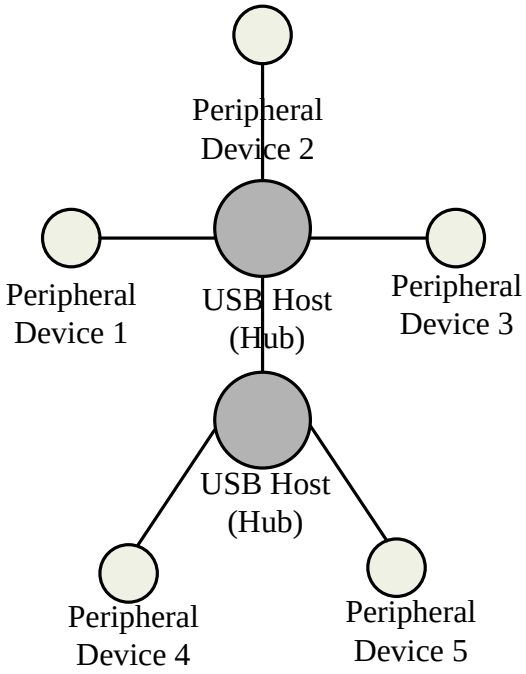
External Communication Interface – RS-232 C & RS-485

- ✓ As per the EIA standard RS-232 C supports baudrates up to 20Kbps (Upper limit 19.2Kbps) The commonly used baudrates by devices are 300bps, 1200bps, 2400bps, 9600bps, 11.52Kbps and 19.2Kbps
- ✓ The maximum operating distance supported in RS-232 communication is 50 feet at the highest supported baudrate.
- ✓ Embedded devices contain a UART for serial communication and they generate signal levels conforming to TTL/CMOS logic. A level translator IC like MAX 232 from Maxim Dallas semiconductor is used for converting the signal lines from the UART to RS-232 signal lines for communication. On the receiving side the received data is converted back to digital logic level by a converter IC. Converter chips contain converters for both transmitter and receiver
- ✓ RS-232 uses single ended data transfer and supports only point-to-point communication and not suitable for multi-drop communication
- ✓ RS-422 is another serial interface standard from EIA for differential data communication. It supports data rates up to 100Kbps 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 makes use of the ‘addressing’ mechanism to identify slave devices

The Typical Embedded System

External Communication Interface – Universal Serial Bus (USB)

- ✓ Universal Serial Bus (USB) is a wired high speed serial bus for data communication
- ✓ The USB communication system follows a star topology with a USB host at the center and one or more USB peripheral devices/USB hosts connected to it
- ✓ A USB host can support connections up to 127, including slave peripheral devices and other USB hosts
- ✓ 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 packet. There are different standards for implementing the USB Host Control interface; namely Open Host Control Interface (OHCI) and Universal Host Control Interface (UHCI)



The Typical Embedded System

External Communication Interface – Universal Serial Bus (USB)

- ✓ The Physical connection between a USB peripheral device and master device is established with a USB cable
- ✓ The USB cable supports communication distance of up to 5 meters
- ✓ The USB standard uses two different types of connectors namely ‘Type A’ and ‘Type B’ 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)

Pin No:	Pin Name	Description
1	V_{BUS}	Carries power (5V)
2	D-	Differential data carrier line
3	D+	Differential data carrier line
4	GND	Ground signal line

The Typical Embedded System

External Communication Interface – Universal Serial Bus (USB)

- ✓ Each USB device contains a Product ID (PID) and a Vendor ID (VID)
- ✓ The PID and VID are embedded into the USB chip by the USB device manufacturer
- ✓ The VID for a device is supplied by the USB standards forum
- ✓ PID and VID are essential for loading the drivers corresponding to a USB device for communication
- ✓ 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 re-transmission of data in case of any transmission loss
- ✓ 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.

The Typical Embedded System

External Communication Interface – IEEE 1394 (Firewire)

- ✓ 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 Apple Inc's (www.apple.com) implementation of 1394 protocol is popularly known as ***Firewire***.
- ✓ ***i.LINK*** is the 1394 implementation from Sony Corporation (www.sony.net) and ***Lynx*** is the implementation from Texas Instruments (www.ti.com)
- ✓ 1394 supports peer-to-peer connection and point-to-multipoint communication allowing 63 devices to be connected on the bus in a tree topology
- ✓ The 1394 standard supports a data rate of 400 to 3200Mbits/Second
- ✓ IEEE 1394 uses differential data transfer and the interface cable supports 3 types of connectors, namely; 4-pin connector, 6-pin connector (alpha connector) and 9 pin connector (beta connector)
- ✓ The 6 and 9 pin connectors carry power also to support external devices. It can supply unregulated power in the range of 24 to 30V (The Apple implementation is for battery operated devices and it can supply a voltage in the range 9 to 12V)

The Typical Embedded System

External Communication Interface – IEEE 1394 (Firewire)

Pin Name	Pin No: (4 Pin Connector)	Pin No: (6 Pin Connector)	Pin No: (9 Pin Connector)	Description
Power		1	8	Unregulated DC supply. 24 to 30V
Signal Ground		2	6	Ground connection
TPB-	1	3	1	Differential Signal line for Signal Line B
TPB+	2	4	2	Differential Signal line for Signal Line B
TPA-	3	5	3	Differential Signal line for Signal Line A
TPA+	4	6	4	Differential Signal line for Signal Line A
TPA(S)			5	Shield for the differential signal line A. Normally grounded
TPB(S)			9	Shield for the differential signal line B. Normally grounded
NC			7	No connection

The Typical Embedded System

External Communication Interface – IEEE 1394 (Firewire)

- ✓ The IEEE 1394 connector contains two differential data transfer lines namely A and B
- ✓ The differential lines of A are connected to B (TPA+ to TPB+ and TPA- to TPB-) and vice versa
- ✓ Unlike USB interface (Except USB OTG), IEEE 1394 doesn't require a host for communicating between devices. Example, a scanner can be directly connected to a printer for printing
- ✓ The data rate supported by 1394 is far higher than the one supported by USB2.0 interface
- ✓ 1394 is a popular communication interface for connecting embedded devices like 'Digital Camera', 'Camcorder', 'Scanners' with desktop Computers for data transfer and storage

The Typical Embedded System

External Communication Interface – Infrared (IrDA)

- ✓ A serial, half duplex, line of sight based wireless technology for data communication between devices
- ✓ Infrared communication technique makes use of 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 10cm to 1 m
- ✓ 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 16Mbps. The UFIR specs are under development and it is targeting a data rate up to 100Mbps
- ✓ 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 used as the IR source for transmitter and at the receiving end a photodiode is used as the receiver

The Typical Embedded System

External Communication Interface – Bluetooth

- ✓ Low cost, low power, short range wireless technology for data and voice communication
- ✓ Bluetooth operates at 2.4GHz of the Radio Frequency spectrum and uses the Frequency Hopping Spread Spectrum (FHSS) technique for communication.
- ✓ Bluetooth supports a theoretical maximum data rate of up to 1Mbps and a range of approximately 30 feet for data communication
- ✓ Bluetooth communication 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 and protocol part is responsible for defining the rules of 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 rules governing the Bluetooth communication is implemented in the 'Bluetooth protocol stack'. The Bluetooth communication IC holds the 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
- ✓ A network formed with one Bluetooth device as master and more than one device as slaves is known as Piconet

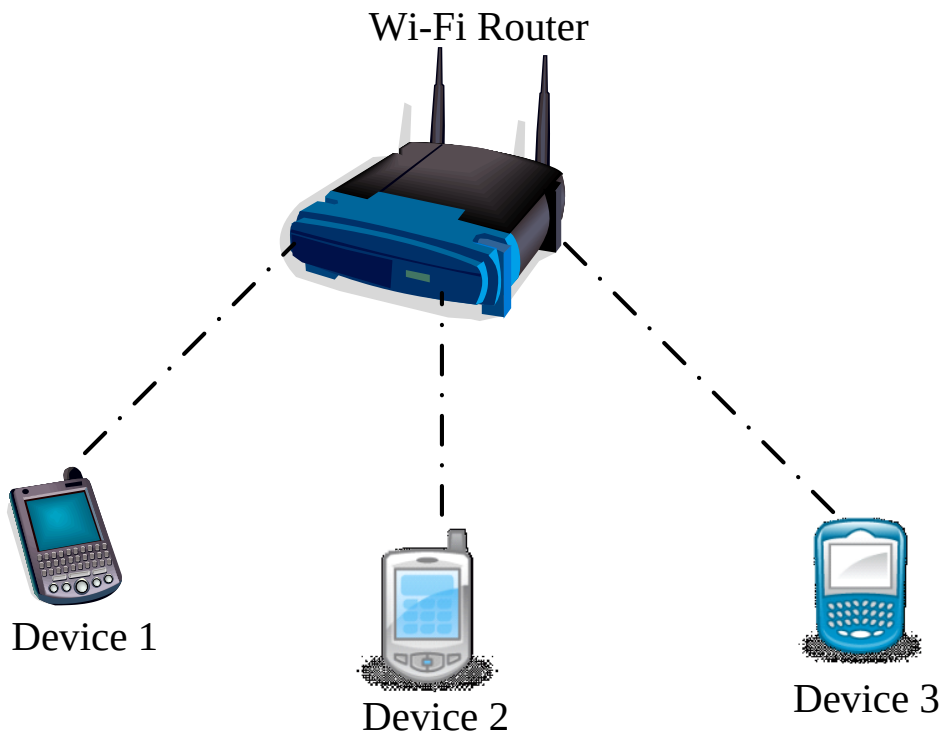
The Typical Embedded System

External Communication Interface – Wi-Fi

- ✓ 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 it supports Internet Protocol (IP) based communication
- ✓ 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
- ✓ Wi-Fi operates at 2.4GHZ or 5GHZ of radio spectrum and they co-exist with other ISM band devices like Bluetooth
- ✓ A Wi-Fi network is identified with a Service Set Identifier (SSID). A Wi-Fi device can connect to a network by selecting the SSID of the network and by providing the credentials if the network is security enabled
- ✓ Wi-Fi networks implements different security mechanisms for authentication and data transfer
- ✓ Wireless Equivalency Protocol (WEP), Wireless Protected Access (WPA) etc are some of the security mechanisms supported by Wi-Fi networks in data communication

The Typical Embedded System

External Communication Interface – Wi-Fi



The Typical Embedded System

External Communication Interface – ZigBee

- ✓ 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
- ✓ ZigBee is primarily targeting application areas like Home & Industrial Automation, Energy Management, Home control/security, Medical/Patient tracking, Logistics & Asset tracking and sensor networks & active RFID
- ✓ Automatic Meter Reading (AMR), smoke and detectors, wireless telemetry, HVAC control, heating control, Lighting controls, Environmental controls, etc are examples for applications which can make use of the ZigBee technology

The Typical Embedded System

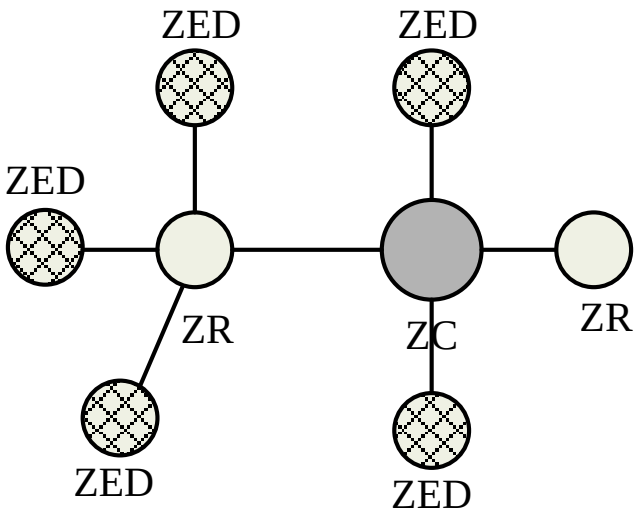
External Communication Interface – ZigBee

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 Typical Embedded System

External Communication Interface – General Packet Radio Service (GPRS)

- ✓ A communication technique for transferring data over a mobile communication network like GSM
- ✓ Data is sent as packets. 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
- ✓ 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

The Typical Embedded System

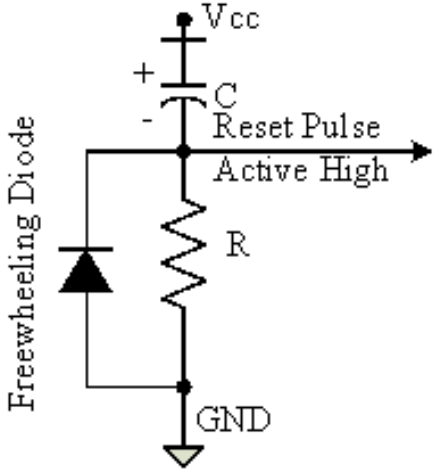
Embedded Firmware

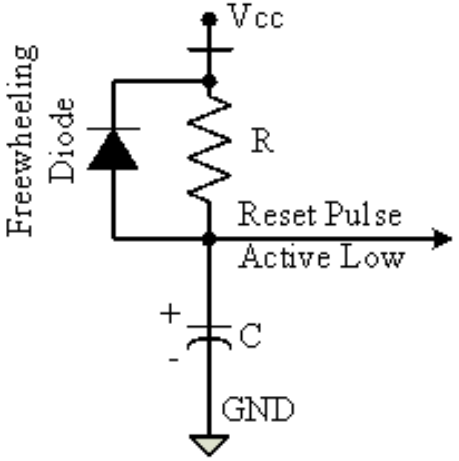
- ✓ The control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system
- ✓ The embedded firmware can be developed in various methods like
  - ✓ Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (The IDE will contain an editor, compiler, linker, debugger, simulator etc. IDEs are different for different family of processors/controllers.
  - ✓ Write the program in Assembly Language using the Instructions Supported by your application's target processor/controller

The Typical Embedded System

Other System Components – 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 vector can be relocated to an address for processors/controllers supporting bootloader
- ✓ 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).

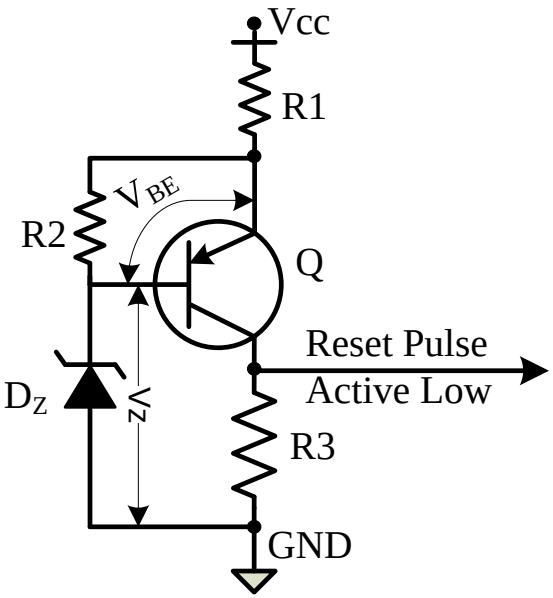




The Typical Embedded System

Other System Components – 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
- ✓ 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 the 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 doesn't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs

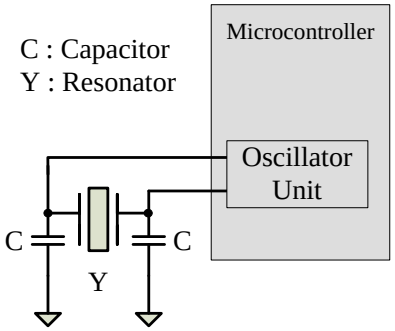


The Typical Embedded System

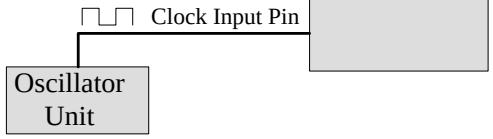
Other System Components – Oscillator Unit

- ✓ A microprocessor/microcontroller is a digital device made up of digital combinational and sequential circuits
- ✓ The instruction execution of a microprocessor/controller occurs in sync 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
- ✓ Certain processor/controller chips may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally
- ✓ Quartz crystal Oscillators are example for clock pulse generating devices

Quartz Crystal Resonator



Crystal Oscillator



The Typical Embedded System

Other System Components – Real Time Clock (RTC)

- ✓ The system component responsible for keeping track of time. RTC holds information like current time (In hour, 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 updation, managing software timers etc when an RTC timer tick interrupt occurs

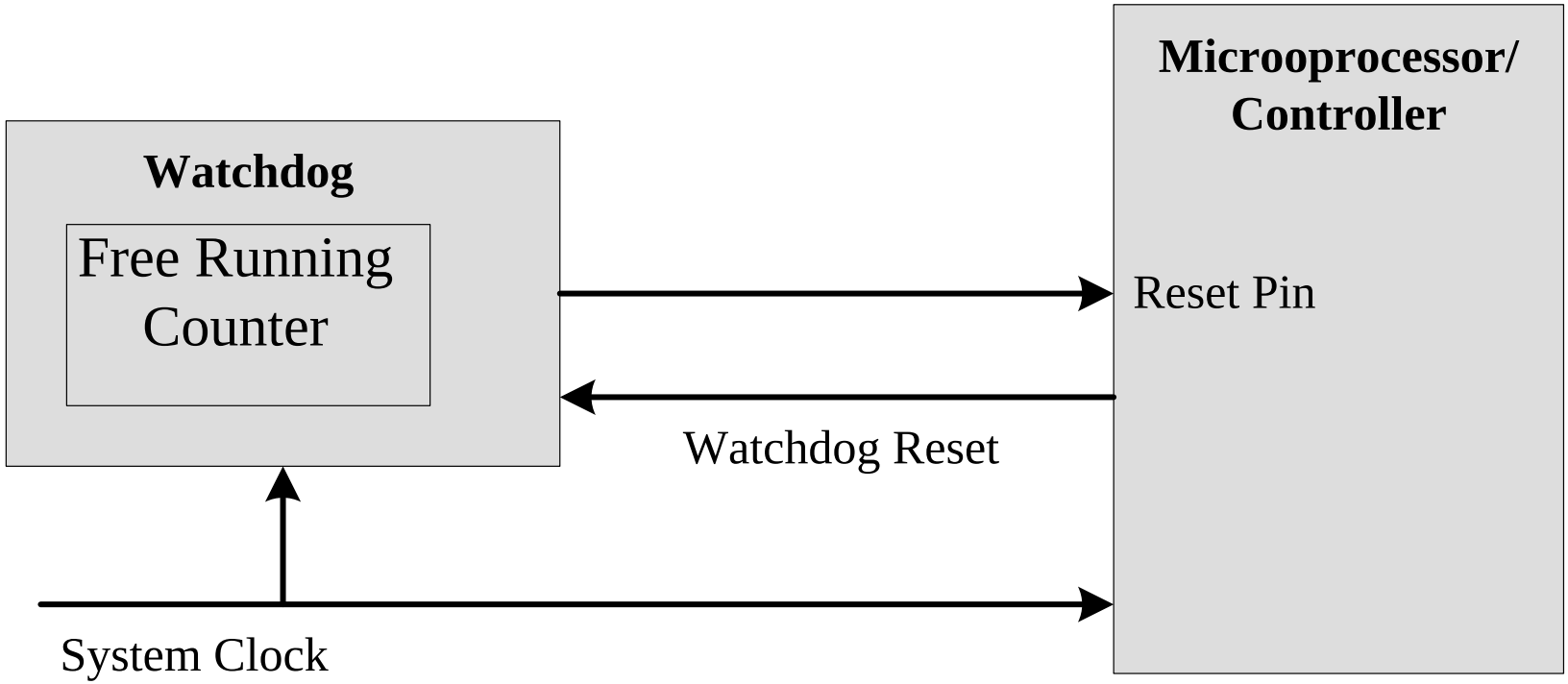
The Typical Embedded System

Other System Components – Watch Dog Timer (WDT)

- ✓ A timer unit 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 (subroutine or portion of code which is susceptible to execution hang up) 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 timer the WDT can be stopped from action
- ✓ Most of the processors implement watchdog as a built-in component and provides status register to control the watchdog timer (like enabling and disabling watchdog functioning) and watchdog timer register for writing the count value. 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 Typical Embedded System

Other System Components – Watch Dog Timer (WDT)



External Watch Dog Timer Unit Interfacing with Processor