



Introduction to ARM

Module 1,ESD(18EC62)

Sheetal Patted

Asst Prof,Electronics and Communication Dept

APS College of Engg

Introduction

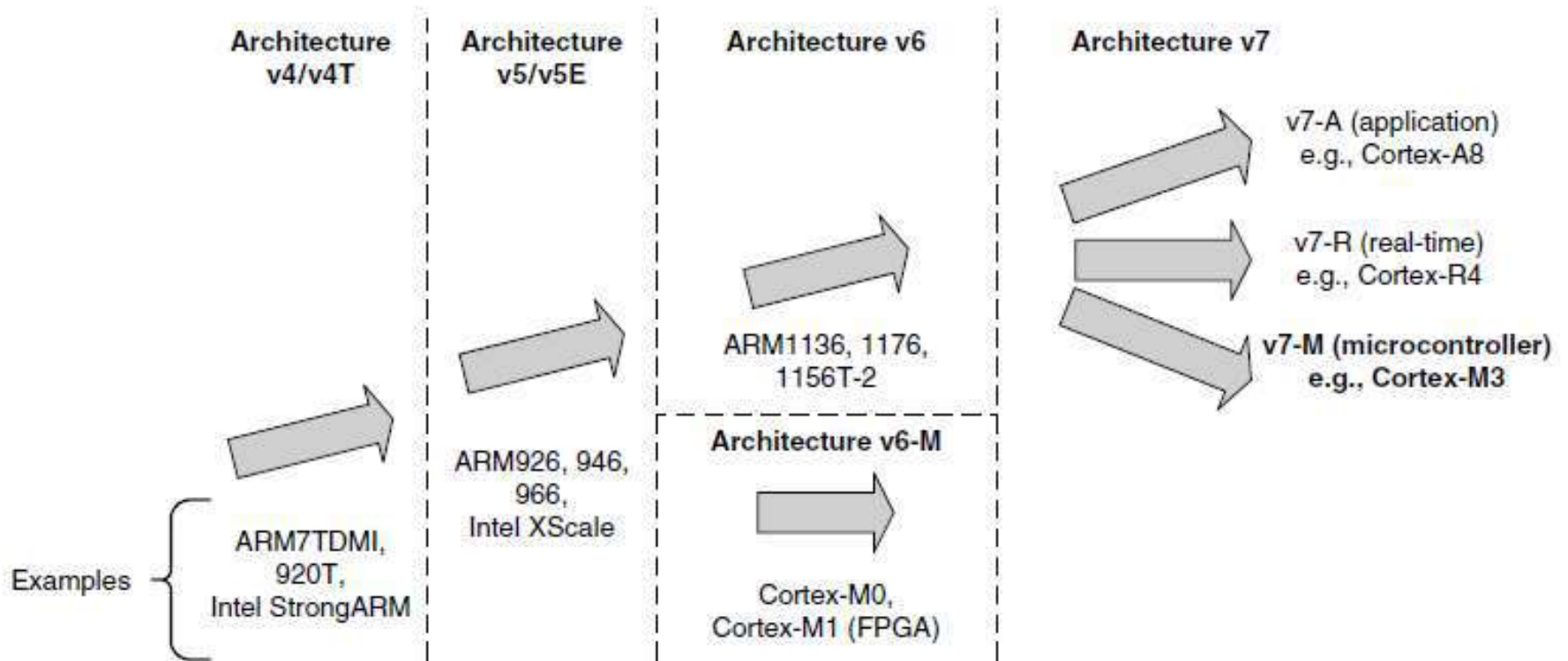
- Microcontrollers are required to handle more work without increasing products frequency or power.
- ARM cortex M3 was designed to target 32 bit controller.
- Addresses the requirements of 32 bit controller in following ways
 - **Greater performance efficiency**
 - More work without increasing frequency or power
 - **Low power consumption**
 - Longer battery life
 - Critical in portable products
 - **Enhanced determinism**
 - Critical tasks and interrupts are serviced as quickly as possible in known number of cycles

- **Improved code density**
 - Code fits in smallest memory foot print
- **Lower cost solutions**
- **Wide choice of development tools**
 - From low cost or free compilers to full featured development suites
- ARM 7 is the most widely used 32 bit embedded processor in history
- ARM does not manufacture processor or sell the chips directly
 - ARM licenses the processor design to business partners
 - Partners create their:
 - Processors
 - Controllers
 - System on chip solutions
 - This business model is commonly called IP

- ARM architecture design is divided into 3 portfolios
 - **A profile:** high performance, open application platforms
 - **R profile:** High end embedded systems in which real time performance is needed
 - **M profile:** Deeply embedded microcontroller type systems
- **A profile**
 - Designed to handle complex applications such as high end OS
 - Requires
 - Highest processing power
 - Virtual memory system support with Memory Management Unit
 - E.g high end mobile phones, electronic wallets
- **R profile**
 - Real time high performance processors
 - High processing power
 - High reliability
 - Low latency

- **M profile**
 - Target low cost applications
 - Processing efficiency and cost important
 - Power consumption low
 - Low interrupt latency
 - Ease of use
 - E.g Industrial control applications, real time control applications

Evolution of ARM processor



Processor Naming

- Traditionally, ARM used a numbering scheme to name processors.
- In the early days (the 1990s), suffixes were also used to indicate features on the processors. For example, with the ARM7TDMI processor:
 - *T indicates Thumb instruction support*
 - *D indicates JTAG debugging*
 - *M indicates fast multiplier*
 - *I indicates an embedded ICE module.*
- Subsequently, it was decided that these features should become standard features of future ARM processors

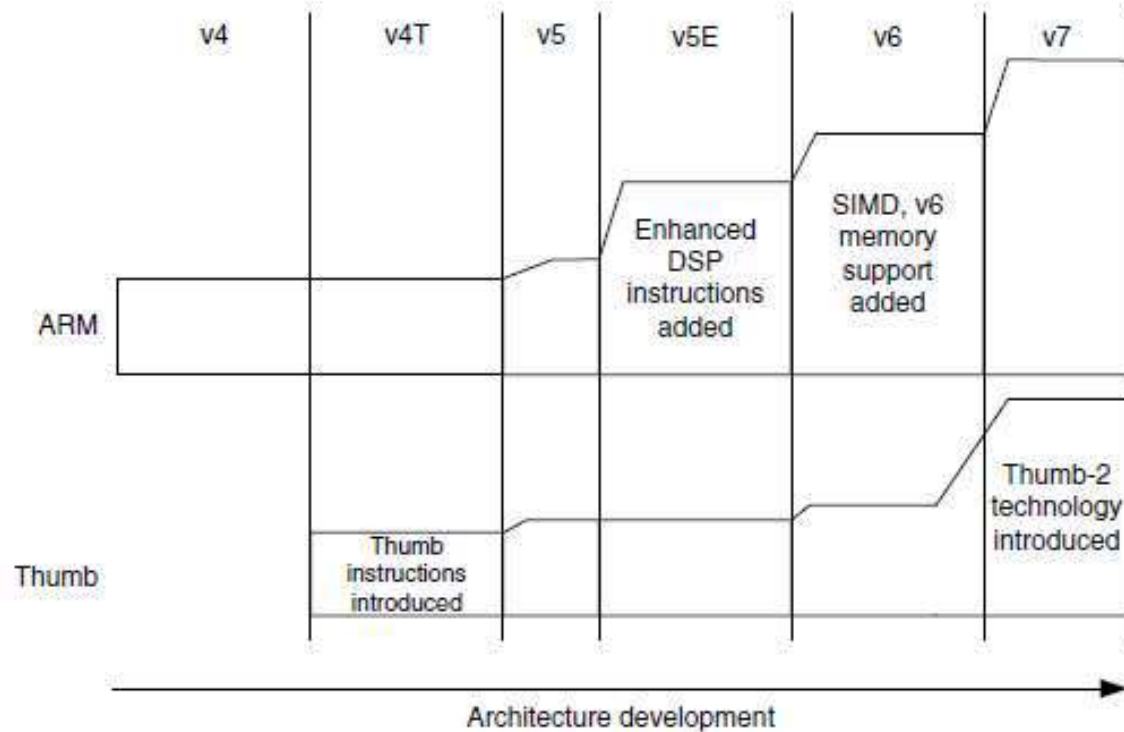
- Instead, variations on memory interface, cache, and tightly coupled memory (TCM) have created a new scheme for processor naming.
- For example, ARM processors with cache and MMUs are now given the suffix “26” or “36,”
- whereas processors with MPUs are given the suffix “46” (e.g., ARM946E-S).
- In addition, other suffixes are added to indicate synthesizable2 (*S*) and Jazelle (*J*) technology. With version 7 of the architecture
- ARM has migrated away from these complex numbering schemes that needed to be decoded, moving to a consistent naming for families of processors, with Cortex its
- initial brand. a v7 processor but was based on the v4T architecture.

Table 1.1 ARM Processor Names

Processor Name	Architecture Version	Memory Management Features	Other Features
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E	DSP	
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU (optional)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU or MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + multiprocessor cache support	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M0	ARMv6-M		NVIC
Cortex-M1	ARMv6-M	FPGA TCM interface	NVIC
Cortex-M3	ARMv7-M	MPU (optional)	NVIC

Processor Name	Architecture Version	Memory Management Features	Other Features
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle, NEON + floating point
Cortex-A9	ARMv7-A	MMU + TrustZone + multiprocessor	DSP, Jazelle, NEON + floating point

Instruction set development



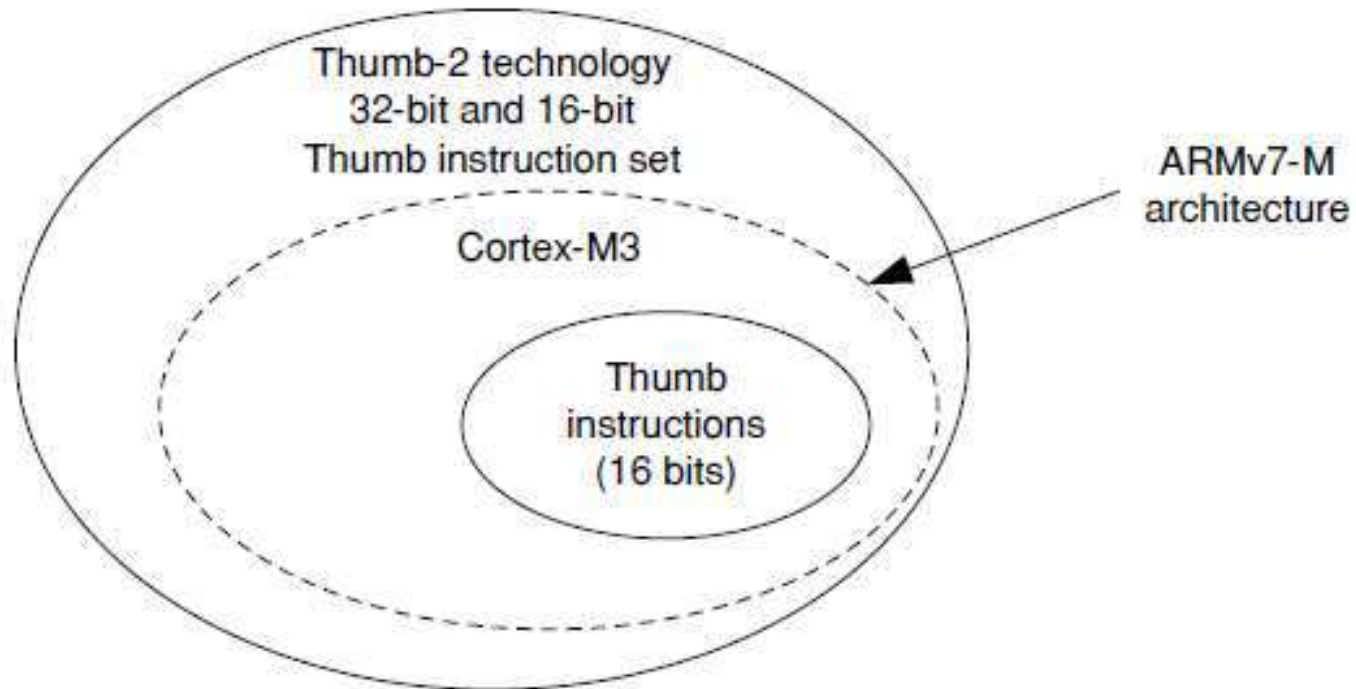
- Two different instruction sets are supported on the ARM processor
 - 32 bits ARM instructions
 - 16 bits Thumb Instructions

- During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one
- Thumb instructions provides only a subset of the ARM instructions
 - It can provide higher code density
 - Useful for products with tight memory requirements
- With update in architecture version, extra instructions have been added to both ARM and thumb.
- Thumb-2 set is a new superset of Thumb instructions
 - Contains both 16 bit and 32 bit instructions

Thumb-2 Technology and ISA

- Thumb-2 technology extended the Thumb ISA into a highly efficient and powerful instruction set.
- Delivers significant benefits in terms of
 - Ease of use
 - Code size
 - Performance
- Allows more complex operations to be carried out in the thumb state
 - Allows higher efficiency by reducing the number of states switching between ARM state and Thumb state.
- Cortex M3 processor is not backward compatible with traditional ARM processors

- Cortex-M3 is not backward compatible with traditional ARM processors



Processor Applications

- **Low-cost microcontrollers**
 - Ideally suited for low cost microcontrollers commonly used in consumer products
 - Toys and electrical appliances
 - Its lower power, high performance and ease of use advantages enable embedded develop32-bit developers to migrate to 32 bit systems and develop ARM products
- **Automotive**
 - Has very high performance efficiency and low interrupt latency allowing it to be used in real time systems.
 - Supports up to 240 external vectored interrupts

- **Data Communications**

- Processors 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**

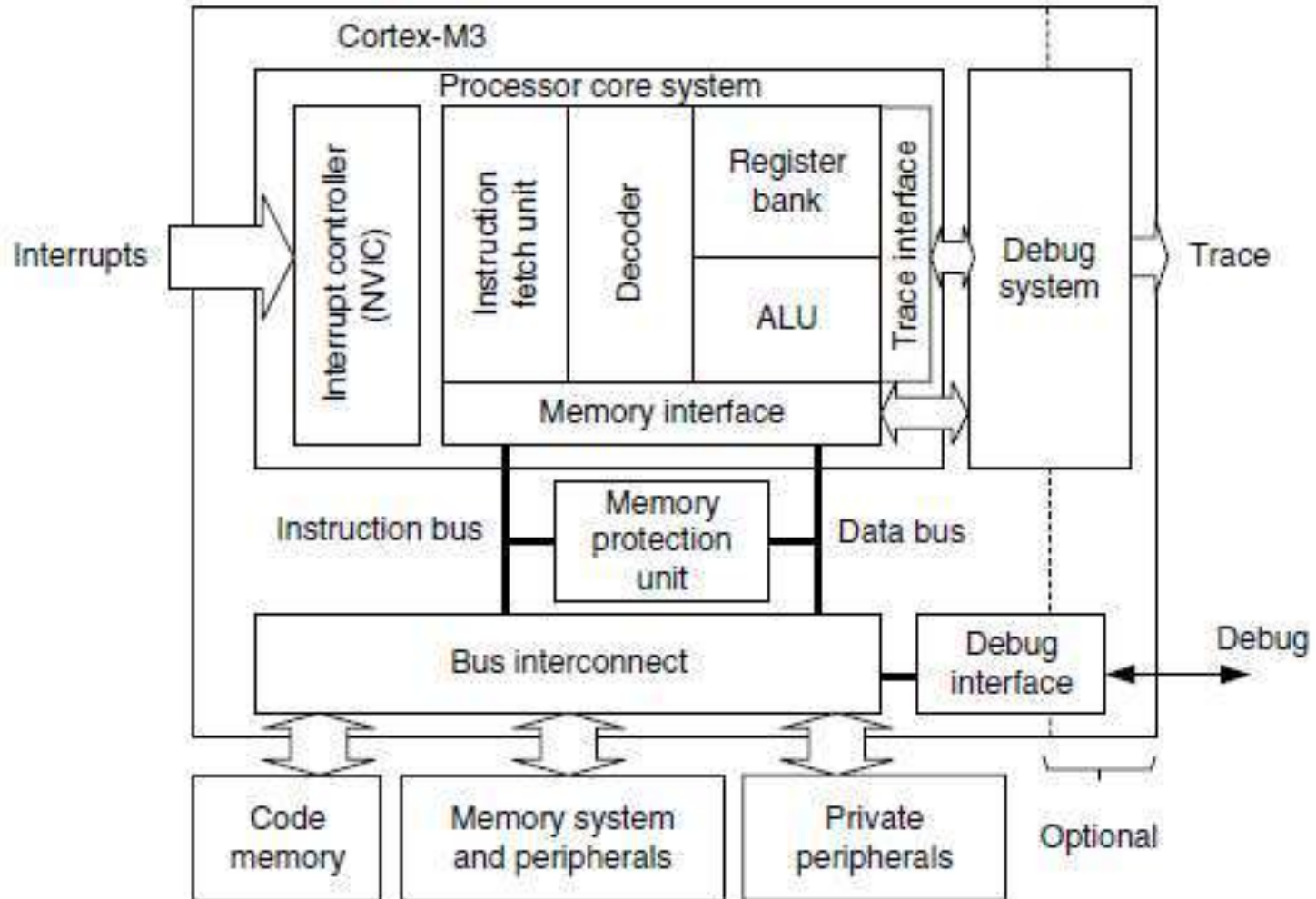
- In industrial control applications simplicity, fast response and reliability are key factors
- Cortex M3 processors interrupt feature, low interrupt latency and enhanced fault handling features make it strong candidate in this area

- Consumer Products
 - Many consumer products have a high-performance microprocessor
 - Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute
 - Provides Robust memory protection

Fundamentals

- Cortex M3 is a 32 bit microprocessor
- It has a 32-bit data path
- Has a 32-bit register bank
- Has a 32-bit memory interfaces
- Processor has a harvard architecture
 - Separate instruction and data bus
- For complex applications that require more memory system features, the cortex-M3 processor has an optional memory protection unit (MPU)

Simplified view of Cortex-M3

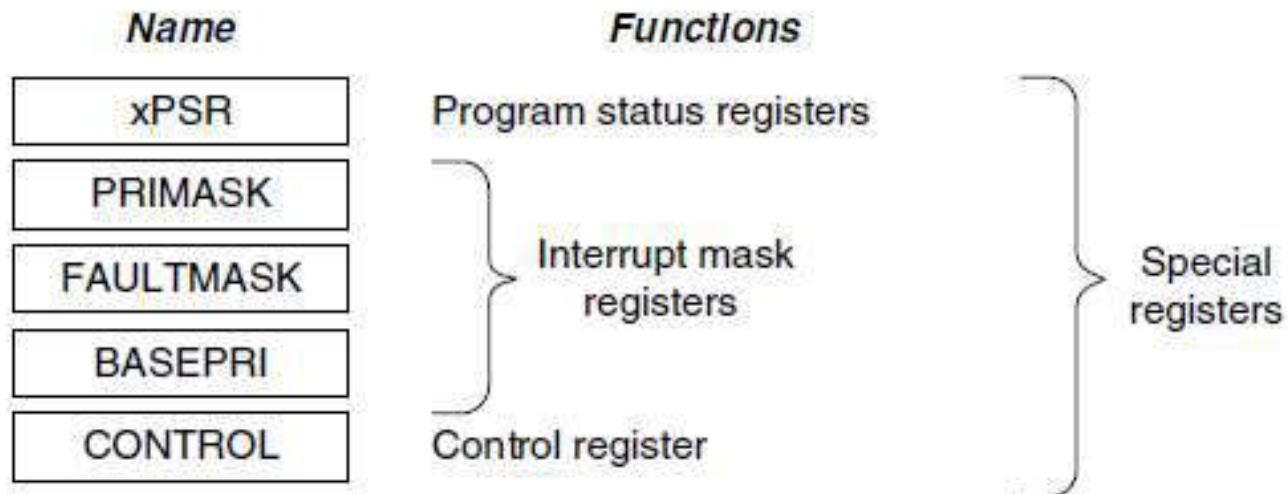


Registers

Name		Functions (and banked registers)	
R0		General-purpose register	} Low registers
R1		General-purpose register	
R2		General-purpose register	
R3		General-purpose register	
R4		General-purpose register	
R5		General-purpose register	
R6		General-purpose register	
R7		General-purpose register	
R8		General-purpose register	} High registers
R9		General-purpose register	
R10		General-purpose register	
R11		General-purpose register	
R12		General-purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

Special Registers

- Cortex-M3 has a number of special registers.
 - Program Status Registers(PSR'S)
 - Interrupt Mask Registers(PRIMASK,FAULTMASK,BASEPRI)
 - Control Registers (CONTROL)



Special Register 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

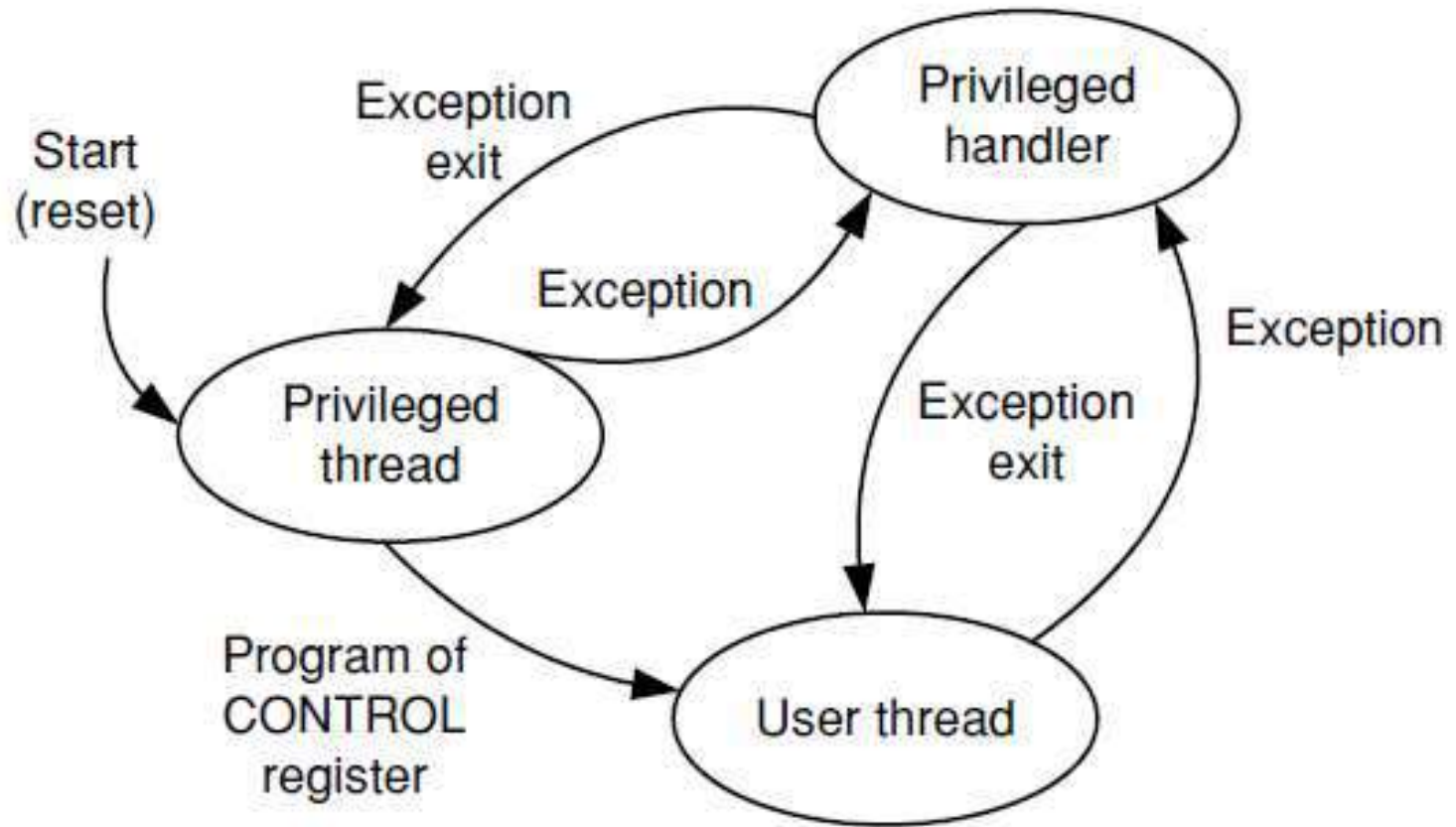
Operation Modes

- Cortex-M3 processor has 2 modes and 2 privilege levels

	<i>Privileged</i>	<i>User</i>
<i>When running an exception handler</i>	Handler mode	
<i>When not running an exception handler (e.g., main program)</i>	Thread mode	Thread mode

- The operation modes determine whether the processor is running a normal program or running an exception
- The privilege levels provide a mechanism for safeguarding memory access to critical regions

Allowed Operation mode transitions



Built in NVIC

- Cortex-M3 processor includes a interrupt controller called the NVIC
- It is closely coupled to the processor and provides a number of features
 - Nested interrupt support
 - Vectored interrupt support
 - Dynamic priority changes support
 - Reduction interrupt latency
 - Interrupt masking

- **Nested Interrupt Support**

- Provides nested interrupt support
- All the external interrupts and most of the system exception can be programmed to different priority levels
- When an interrupt occurs the NVIC compares the priority of the running and newly arrived interrupt
 - If the priority of the newly arrived is higher than the currently executing the interrupt handler will override the current running task
 - If priority of the newly arrived is less than the currently executing, the newly arrived is rejected

- **Vectored Interrupt Support**

- When an interrupt is accepted, the starting address of the ISR is located from a vector table in memory
- No need to use software to determine and branch to the starting address of the ISR.
- Takes less time to process the interrupt request

- **Dynamic Priority Change support**

- Priority levels of interrupts can be changed by the software during run time
- Interrupts that are being serviced are blocked from further activation until the ISR is completed
- Priority can be changed without risk of accidental reentry

- **Reduction of Interrupt Latency**

- Cortex-M3 processor includes a number of advanced features to lower the interrupt latency
- Features Includes
 - Automatic saving and restoring some register contents
 - Reducing delay in switching from one ISR to another
 - Handling of late arrival interrupts

- **Interrupt Masking**

- Interrupts and system exceptions can be masked based on
 - their priority level
 - Using Interrupt masking registers- BASEPRI, PRIMASK, FAULTMASK
- Ensure that time critical tasks can be finished on time without being interrupted

Memory Map

- Cortex-M3 has a predefined memory map
- Allows built in peripherals, such as interrupt controller and the debug components to be accessed by simple memory access instructions
- Most system features are accessible in C program code
- Predefined memory allows the processor to be highly optimized for speed and ease of integration in SOC

0xFFFFFFFF	System level	Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xE0000000		
0xDFFFFFFF	External device	Mainly used as external peripherals
0xA0000000		
0x9FFFFFFF	External RAM	Mainly used as external memory
0x60000000		
0x5FFFFFFF	Peripherals	Mainly used as peripherals
0x40000000		
0x3FFFFFFF	SRAM	Mainly used as static RAM
0x20000000		
0x1FFFFFFF	CODE	Mainly used for program code. Also provides exception vector table after power up
0x00000000		

Bus interfaces

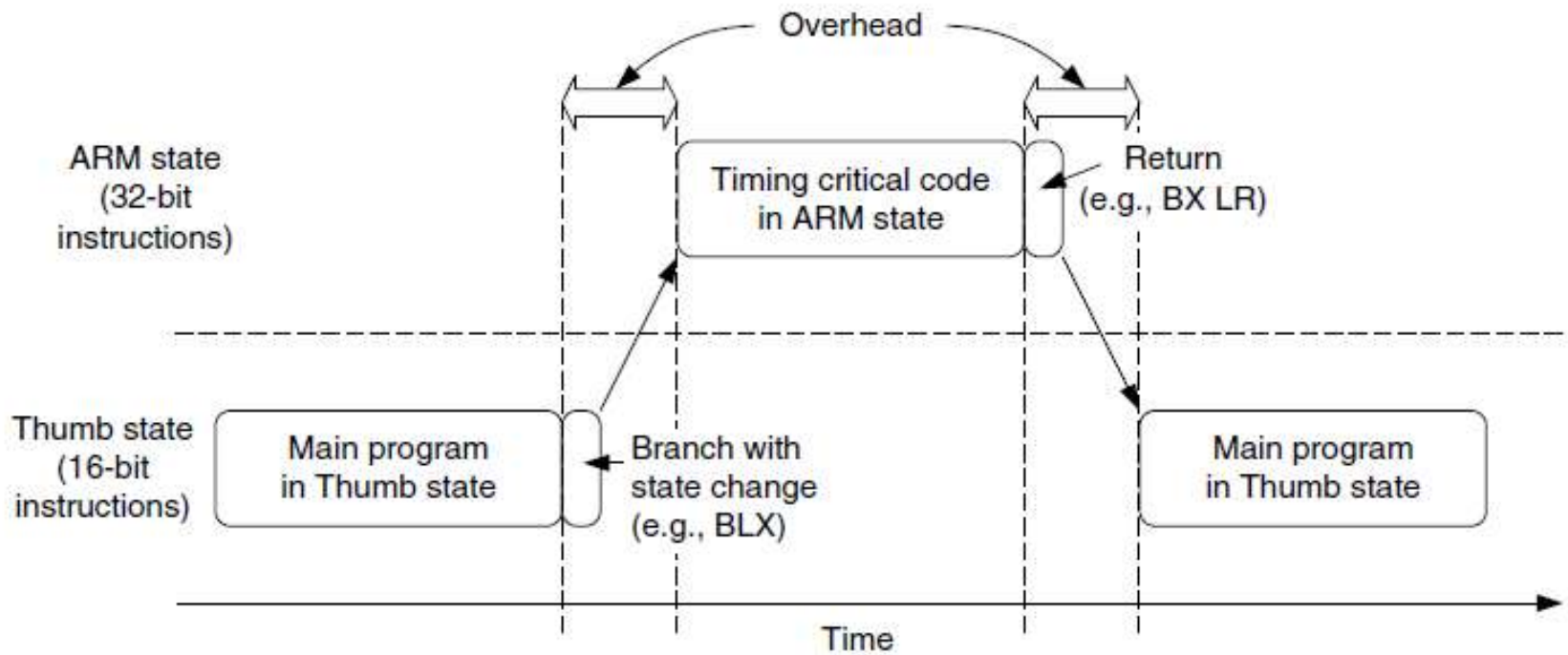
- There are several bus interfaces on cortex-M3 processor which allows processor to carry instruction and data accesses at the same time.
- The main bus interfaces are as follows:
 - Code memory bus
 - Code memory region access is carried out on this
 - Physically consists of two buses, one called I-code and other called D-code
 - These are optimized for instruction fetches for best instruction execution speed
 - System bus
 - Used to access memory and peripherals
 - Provides access to SRAM, peripherals, external RAM, external devices and part of the system level memory regions
 - Private Peripheral bus
 - Provides access to part of the system level memory dedicated to private peripherals such as debugging components

The MPU (Memory Protection Unit)

- Cortex M3 has an optional MPU
- Allows access rules to be set up for the privileged access and user program access
- When an access rule is violated, a fault exception is generated
 - Fault exception handler will be able to analyze the problem and correct it, if possible.
- The MPU can be used in various ways
 - OS can set up the MPU to protect data use
 - Used to make memory regions read-only to prevent accidental erasing of data
- It can help make embedded system more robust and reliable

Instruction Set

- ARM cortex supports the Thumb-2 instruction set
 - One of the most important features of the cortex M3 as it allows both 16-bit and 32-bit instructions to be used together
 - **Advantages**
 - High code density and high efficiency
 - Flexible and powerful yet easy to use
 - Thumb-2 has made it possible to handle all processing requirements in one operation state
 - No state switching overhead, saving both execution time and instruction space
 - No need to separate ARM code and Thumb code source files



- Some powerful and interesting instructions of cortex-M3
 - **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 system
 - **MSR and MRS**
 - Move to special register from general purpose register
 - Move special register to general purpose register

Interrupts and Exceptions

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

Low Power and High Energy Efficiency

- Cortex-M3 processor is designed with various features to allow designers to develop low power and high energy efficient products
 - First ,It has:
 - Sleep mode and deep sleep mode which can reduce power consumption during idle period
 - Second:
 - Its low gate count and design techniques reduce circuit activities in the processor to allow active power to be reduced
- Cortex M-3 has high code density, it has lowered the program size requirement

- Allows the processing task to be completed in a short time so that it can return to sleep mode as soon as possible to cut down energy use
- Energy efficiency of cortex-M3 is better than many 8-bit or 16-bit microcontroller

Debugging Support

- Cortex-M3 processor includes a number of debugging features such as
 - Program execution control
 - Including halting and stepping
 - Instruction breakpoints
 - Data watch points
 - Register and memory accesses
 - Profiling and tracing

- The debugging hardware of ARM Cortex-M3 processor is based on the coresight architecture.
- CPU core itself doesn't have JTAG interface
- Instead a debug interface module is decoupled from the core
- A bus interface called the Debug Access Port (DAP) is provided at the core level
- Through this interface, external debuggers can access control registers to debug hardware as well as the system memory even when the processor is running
- Control of this bus interface is carried out by a debug port (DP) device

- Chip manufacturers can also include an Embedded Trace Macrocell (ETM) to allow instruction trace
- Within the cortex-M3 a number of events can be used to trigger debug actions
 - Debug events can be breakpoints, watchpoints, fault conditions, 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
- Data watch point function is provided by a data watchpoint and trace (DWT) unit in the Cortex-M3 processor

- DWT can be used to trigger or to generate data trace information
- When data is traced, the traced data can be output via the TPIU
- Cortex-M3 also provides a flash patch and breakpoint (FPB)
- An ITM(Instrumentation Trace Macrocell) provides a new way for developers to output data to a debugger.
- All debugging components are controlled via DAP interface bus or by a program running on the processor core.

Characteristic Summary

- Why is the Cortex-M3 such a revolutionary product? What are the advantages and benefits of using the Cortex-M3
 - High performance
 - Advanced Interrupt Handling Feature
 - Low power consumption
 - System features
 - Debug Supports

High Performance

The Cortex-M3 processor delivers high Performance in microcontroller products

- Many instructions, including multiply, are single cycle.
 - Outperforms most microcontroller products
 - Separate data and instruction buses allow simultaneous data and instruction accesses to be performed
 - The Thumb-2 instruction set makes state switching overhead history.
 - The Thumb-2 instruction set provides extra flexibility in programming
 - Many data operations can now be simplified using shorter code
 - Cortex M3 has higher code density and reduced memory requirements

- Instruction fetches are 32 bits.
 - Up to 2 instructions can be fetched in one cycle
 - There's more available bandwidth for data transfer
- The cortex-M3 design allows microcontroller products to operate a high clock frequency
 - Has better CPI (Clock per instruction)
 - Allows more work per Mhz or design can run at lower clock frequency for low power consumption

Advanced Interrupt-Handling Features

Interrupt features on Cortex-M3 processor are easy to use, very flexible and provide high interrupt processing throughput

- The Built-in NVIC supports upto 240 external interrupt inputs
- The Cortex-M3 processor automatically pushes registers R0-R3, Link Register (LR), PSR and PC in the stack at interrupt entry and pops them back at interrupt exit
 - Reduce IRQ handling latency
- NVIC has programmable interrupt priority control for each interrupt
 - Eight levels of priority are supported

Low Power Consumption

Cortex-M3 is suitable for various low-power applications
Suitable for low-power designs because of the low gate count

- It has power saving mode
 - Can enter sleep mode using WFI or WFE
 - Has separated blocks 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

System Features

- Cortex-M3 provides various 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 optional MPU, the processor is more than sufficient to develop robust software and reliable products

Debug Support

- 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 watch points
- 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

Cortex-M3 Basics

- Cortex-M3 processor has register R0-R15 and a number of special registers
 - R0 through R12 are general purpose
 - 16 bit Thumb instructions can only access R0 through R7
 - 32 bit Thumb-2 instructions can access all these registers
 - R0-R7 called as low registers
 - R8-R12 called as high registers

- Stack Pointer R13

- The Cortex M3 has two SP's allowing two separate stack memories to be set up.

- The two SP's are as follows

- Main Stack Pointer

- This is the default SP

- Used by the OS kernel, exception handlers and all application codes that require privileged access

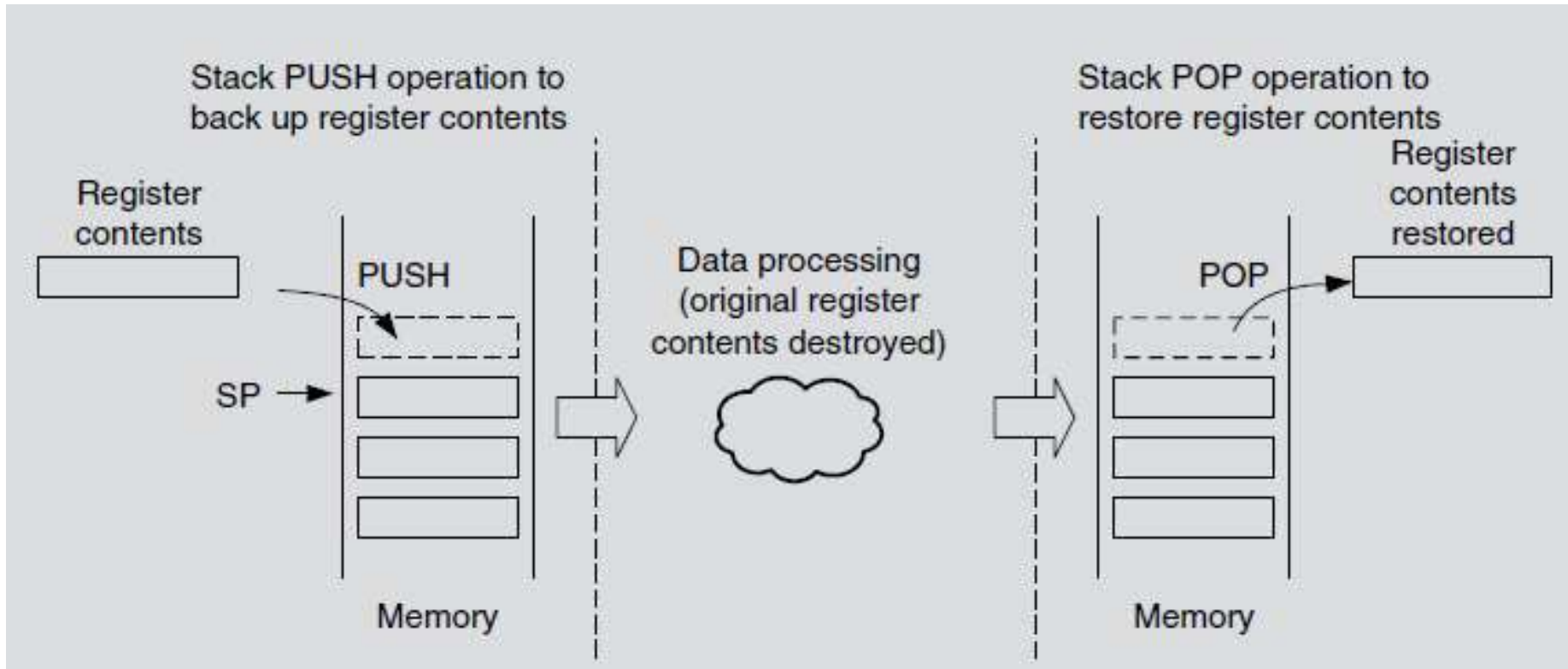
- Process Stack Pointer

- This is used by base level application code

- It is **not necessary to use two SP's** ; simple applications can rely purely on the MSP

- The instructions for accessing stack memory are PUSH and POP

- The Cortex-M3 uses a **full-descending** stack arrangement
- **SP decrements** when new data is stored in the stack
- ***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**
- You can PUSH or POP multiple registers in one instruction
- Instead of using R13 you can use SP in your program codes



```

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
  
```

- **Link Register R14**

- Inside an **assembly program**, you can write it as either **R14** or **LR**
- LR is used to store the return program counter when a subroutine or function is called

- **Program counter R15**

- R15 is the PC. You can access it in assembler code by either R15 or PC

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

Special Registers

- Special registers in Cortex-M3 include the following
 - Program Status Registers
 - Interrupt Mask registers (PRIMASK, FAULTMASK, 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
```


- The PSR's are subdivided into three status registers
 - APSR's (Application Program Status Register)
 - IPSR (Interrupt Program Status Register)
 - EPSRs Execution Program status register.
- The three PSR's 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
- EPSR and IPSR are readonly

```

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

```

	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				

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

Bit Fields in Cortex-M3 PSR

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

PRIMASK, FAULTMASK & BASEPRI

- The PRIMASK, FAULTMASK & BASEPRI registers are used to disable exceptions

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).

- The BASEPRI and PRIMASK 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

```
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
```

Control Register

- The control register is used to define the privilege level and the SP selection
- The register has 2 bits
 - It can either be 0 or 1

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.

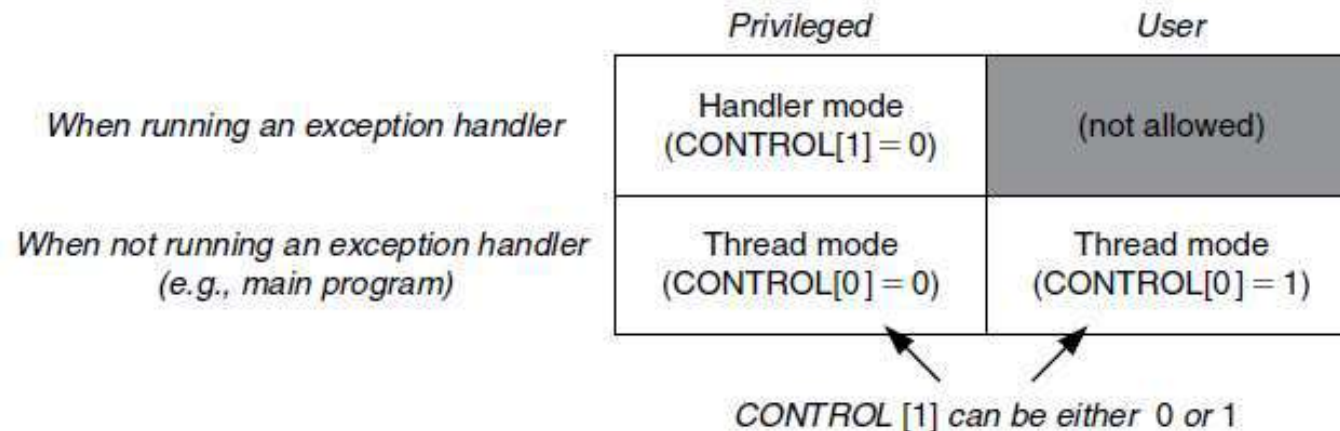
Exceptions and Interrupts

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

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

Operation Mode

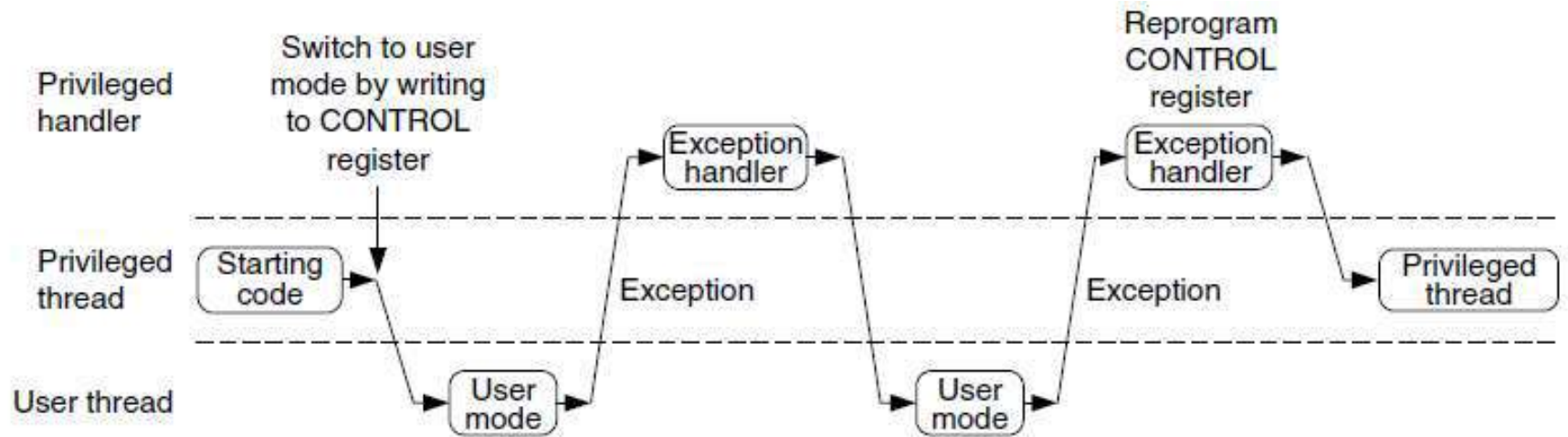
- The Cortex-M3 processor supports two modes and two privilege levels



- When the processor is running in thread mode, it can be in either the privileged or user level
 - But handlers can only be in the privileged level
- When the processor exits reset, it is in thread mode, with privileged access rights

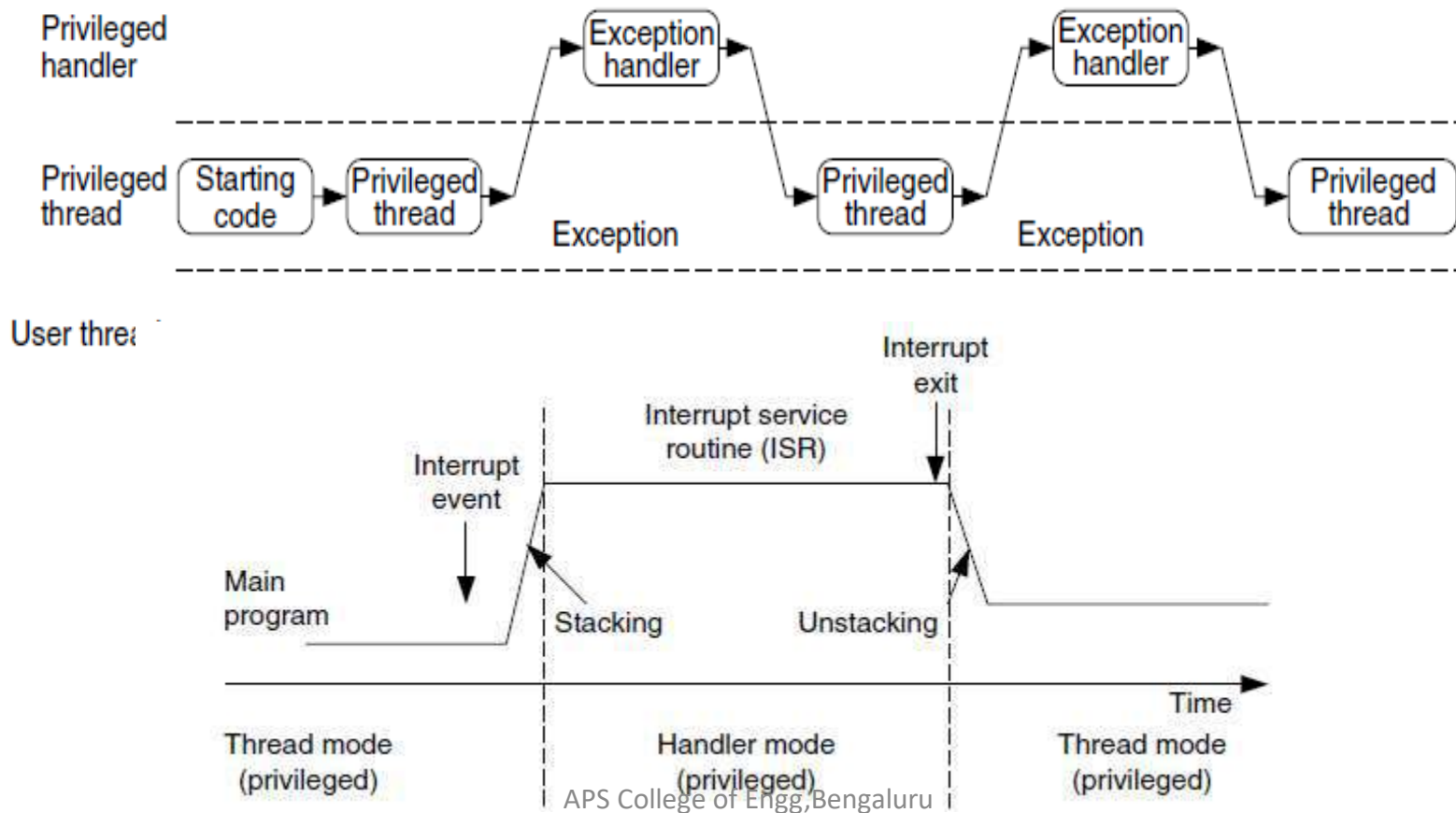
- In the user access level (Thread mode)
 - Access to system control space (SCS) a part of memory region for configuration register and debugging components is blocked
 - Instructions that access special register cannot be used
 - If a program running at the user level tries to access SCS or special registers a fault exception will occur
- Software in the privileged access level can switch the program into 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 the privileged access level when returning to the thread mode

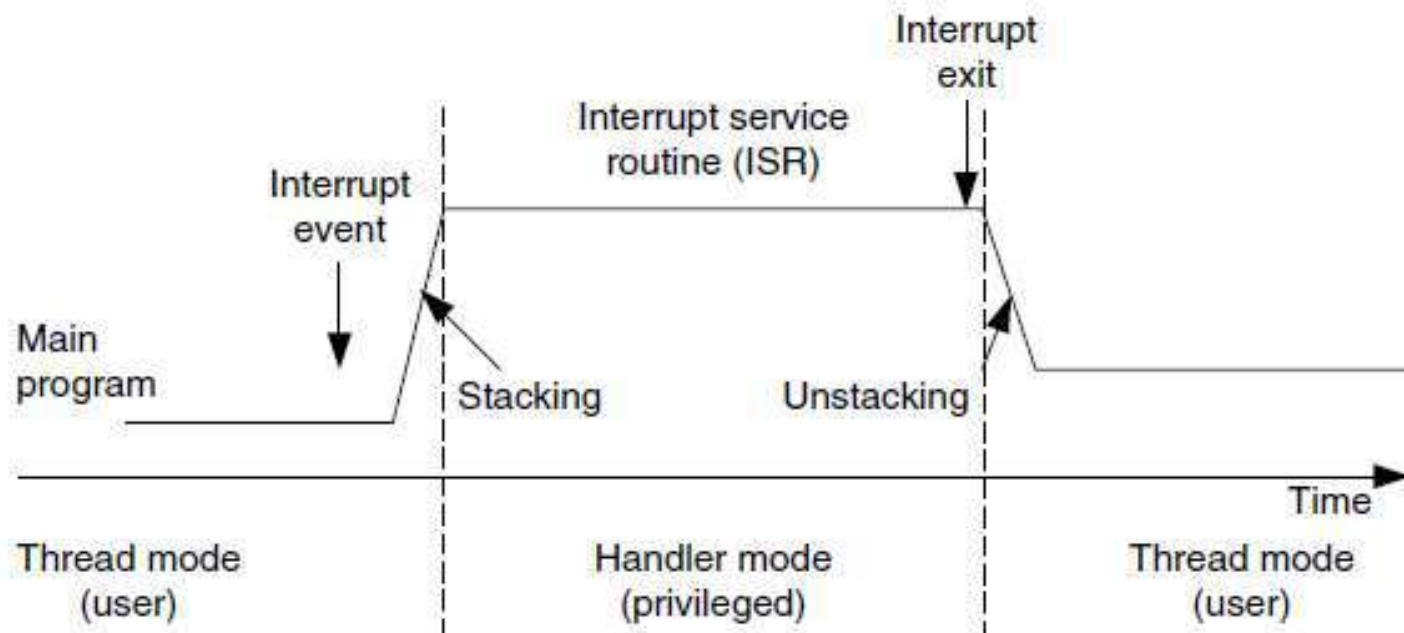


- The support of privileged and user access levels provides a more secure and robust architecture
 - When a user program goes wrong it will not be able to corrupt control registers in the NVIC
 - In addition if the MPU is present, it is possible to block user programs from accessing memory regions used by privileged processes
 - 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

- 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 the control register bit 0 is 1, both processor mode and access level change when an exception takes place



Stack Memory Operations

- Besides normal software-controlled stack PUSH and POP, the stack PUSH and POP operations are also carried out automatically when entering/exiting an exception/interrupt handler
- In general stack operations are
 - Memory write or read operations with address specified by SP
 - Data in register is saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation
 - When PUSH/POP instructions are used, the SP is incremented/decremented automatically

CORTEX-M3 Implementation

Main program

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

Subroutine

function1

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

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

Main program

```
...
; R0 X, R1 Y, R2 Z
BL function 1
```

Subroutine

```
function 1
    PUSH {R0 R2} ; Store R0, R1, R2 to stack
    ... ; Executing task (R0, R1 and R2
        ; could be changed)
    POP {R0 R2} ; restore R0, R1, R2
    BX LR ; Return
```

←

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

Main program

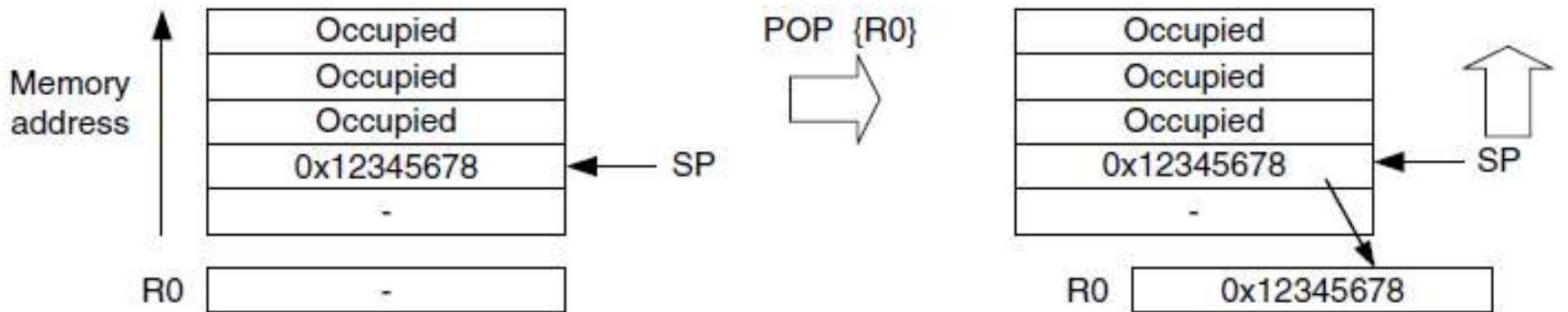
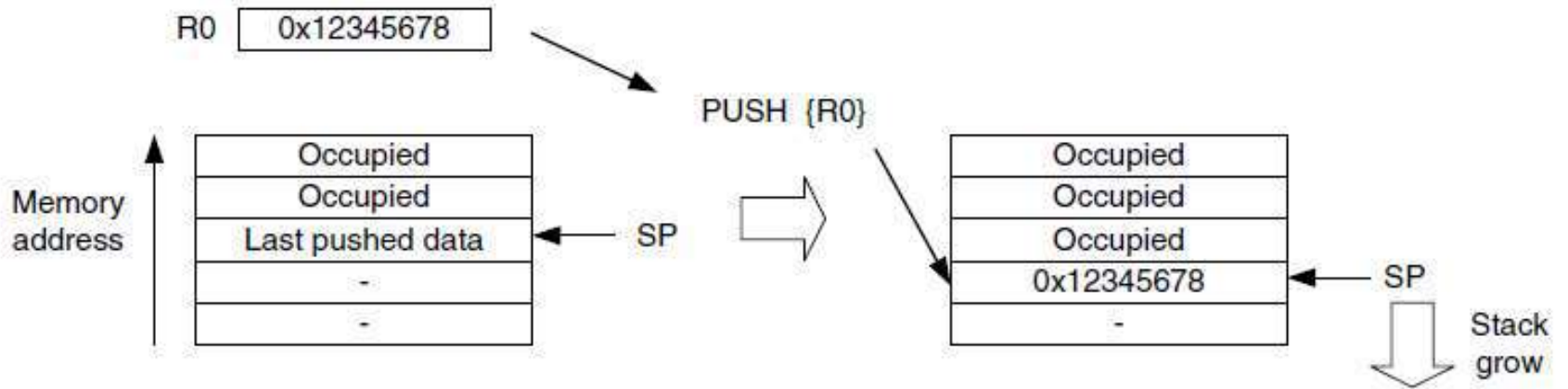
```
...
; R0 X, R1 Y, R2 Z
BL function 1
```

Subroutine

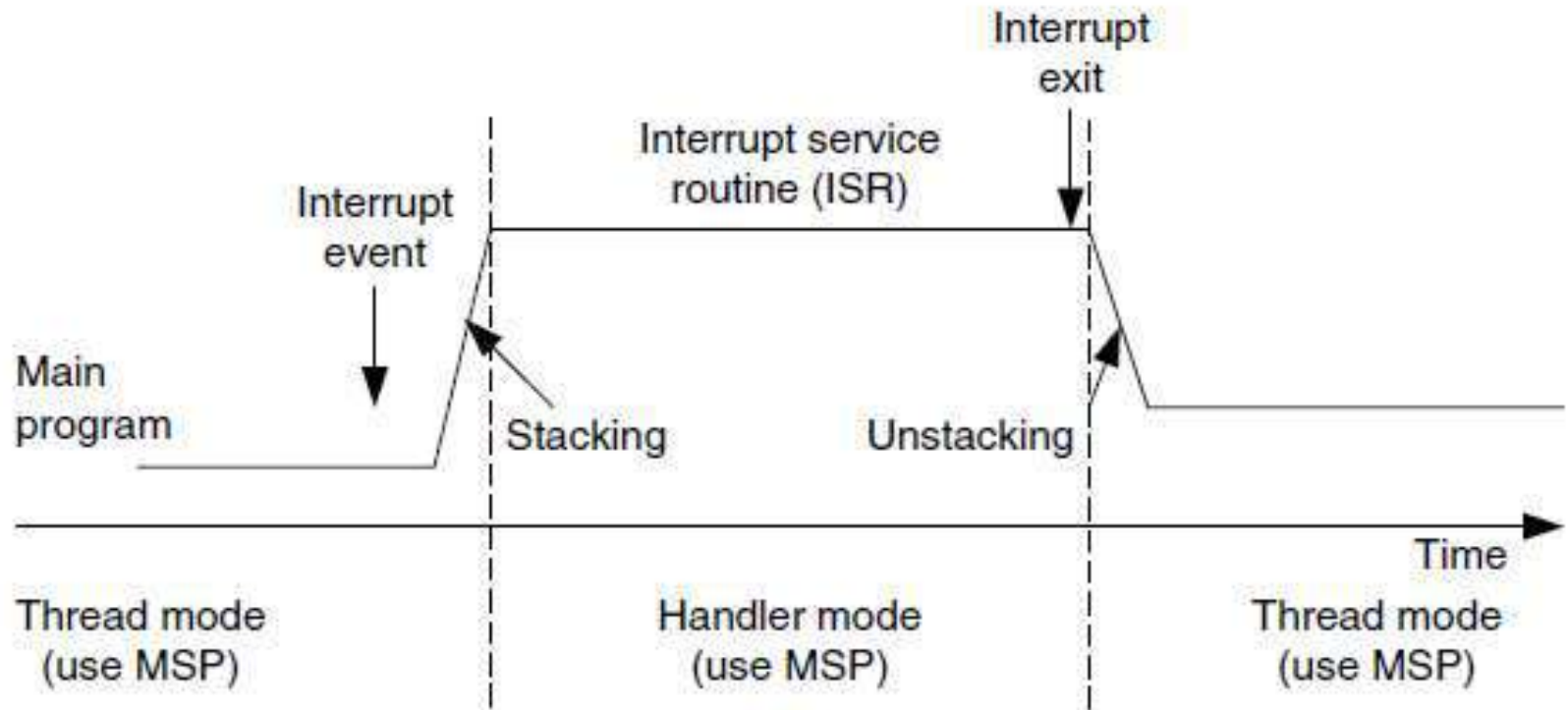
```
function 1
    PUSH {R0 R2, LR} ; Save registers
                        ; including link register
    ... ; Executing task (R0, R1 and R2
        ; could be changed)
    POP {R0 R2, PC} ; Restore registers and
                    ; return
```

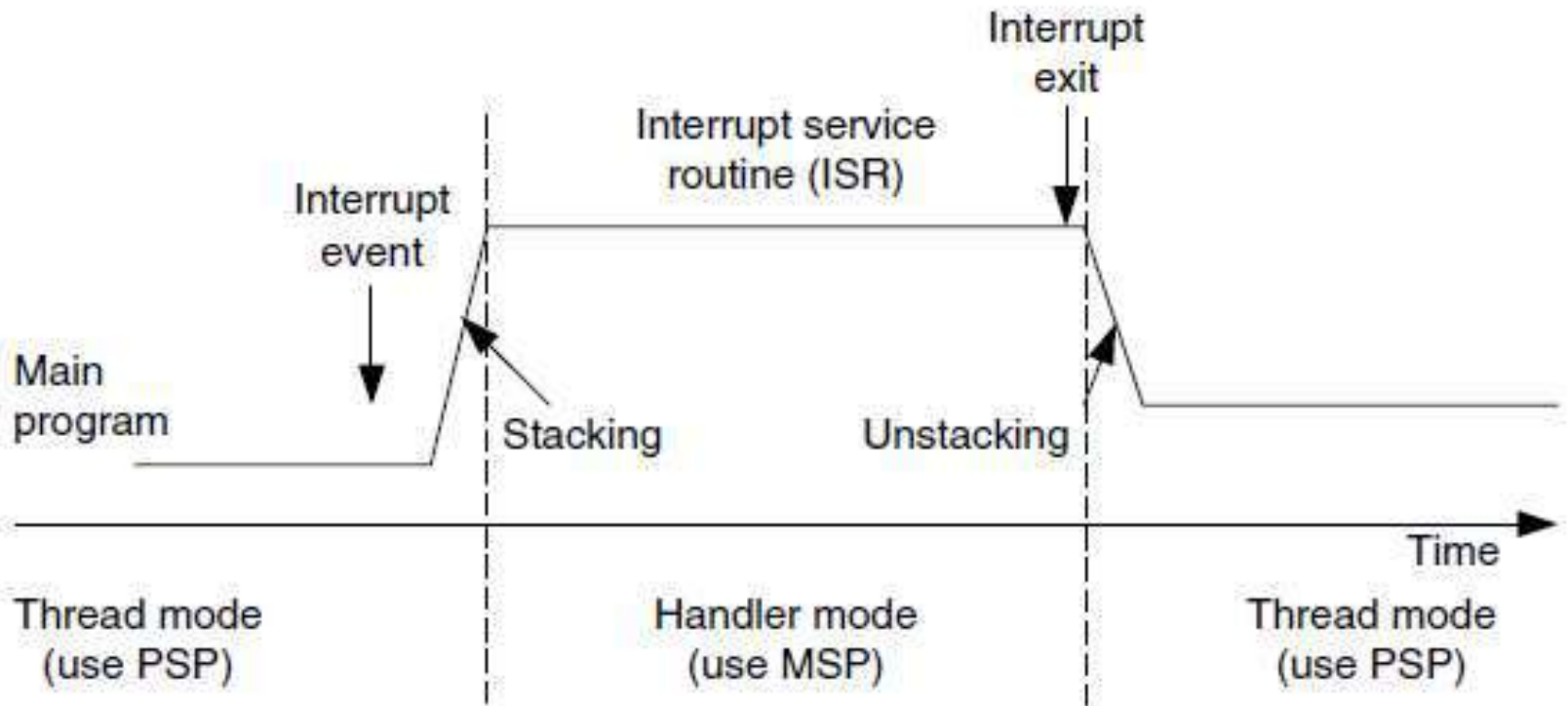
←

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

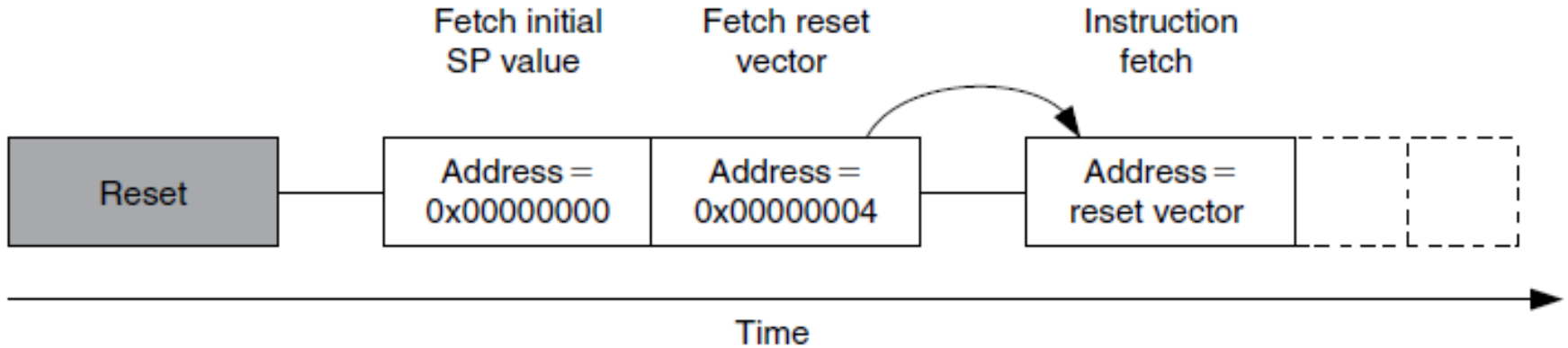


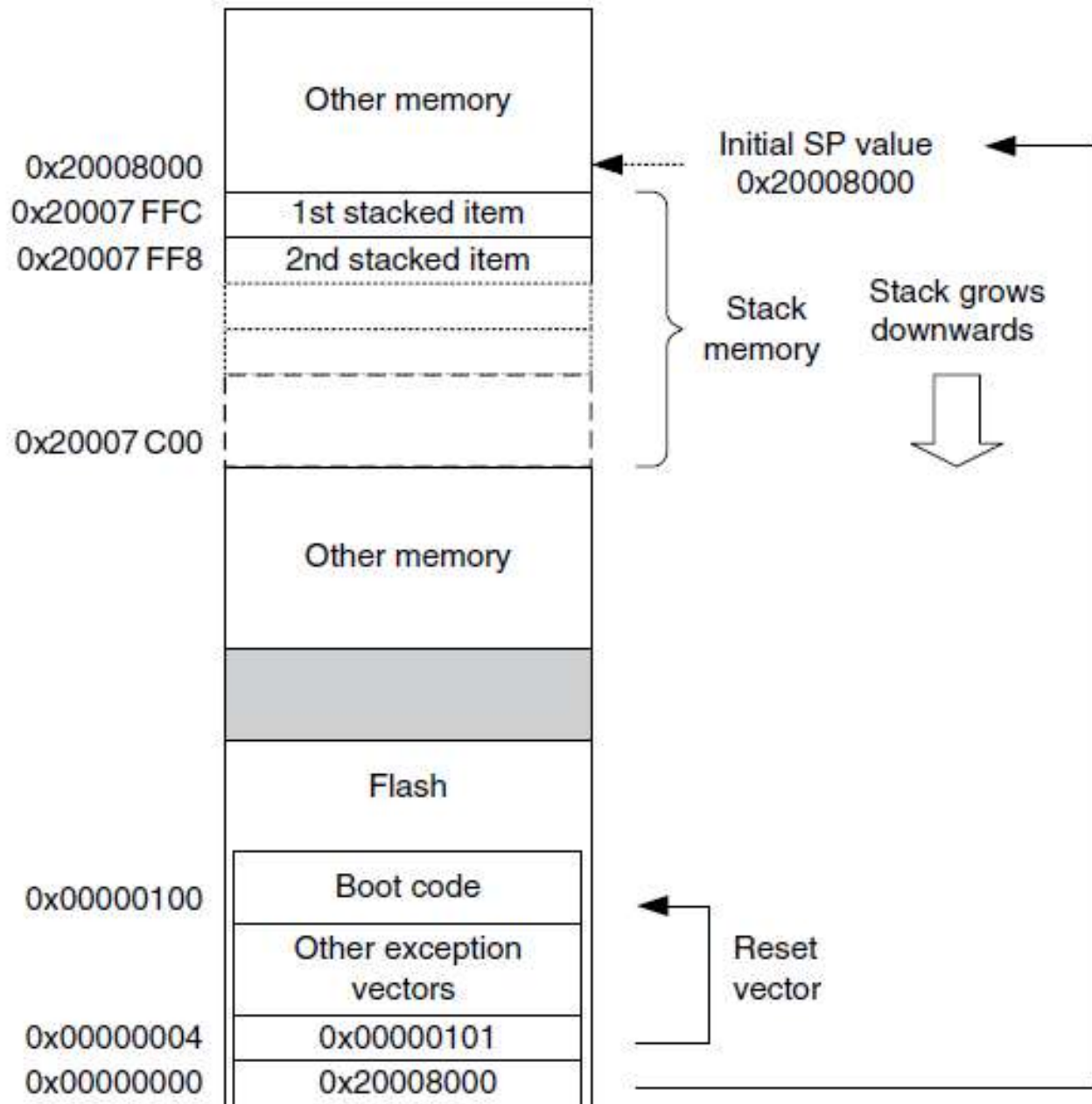
Two Stack Model in Cortex –M3





Reset Sequence



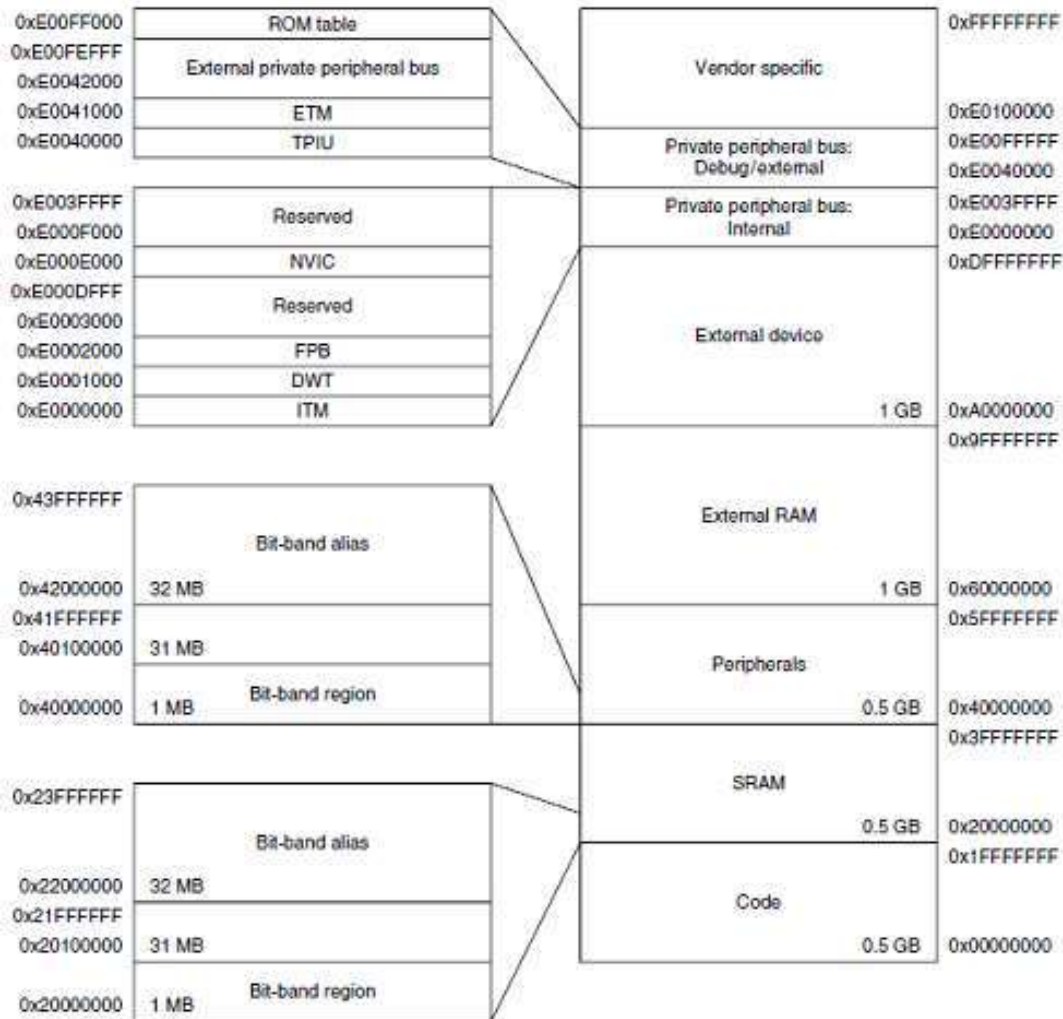




Memory Systems



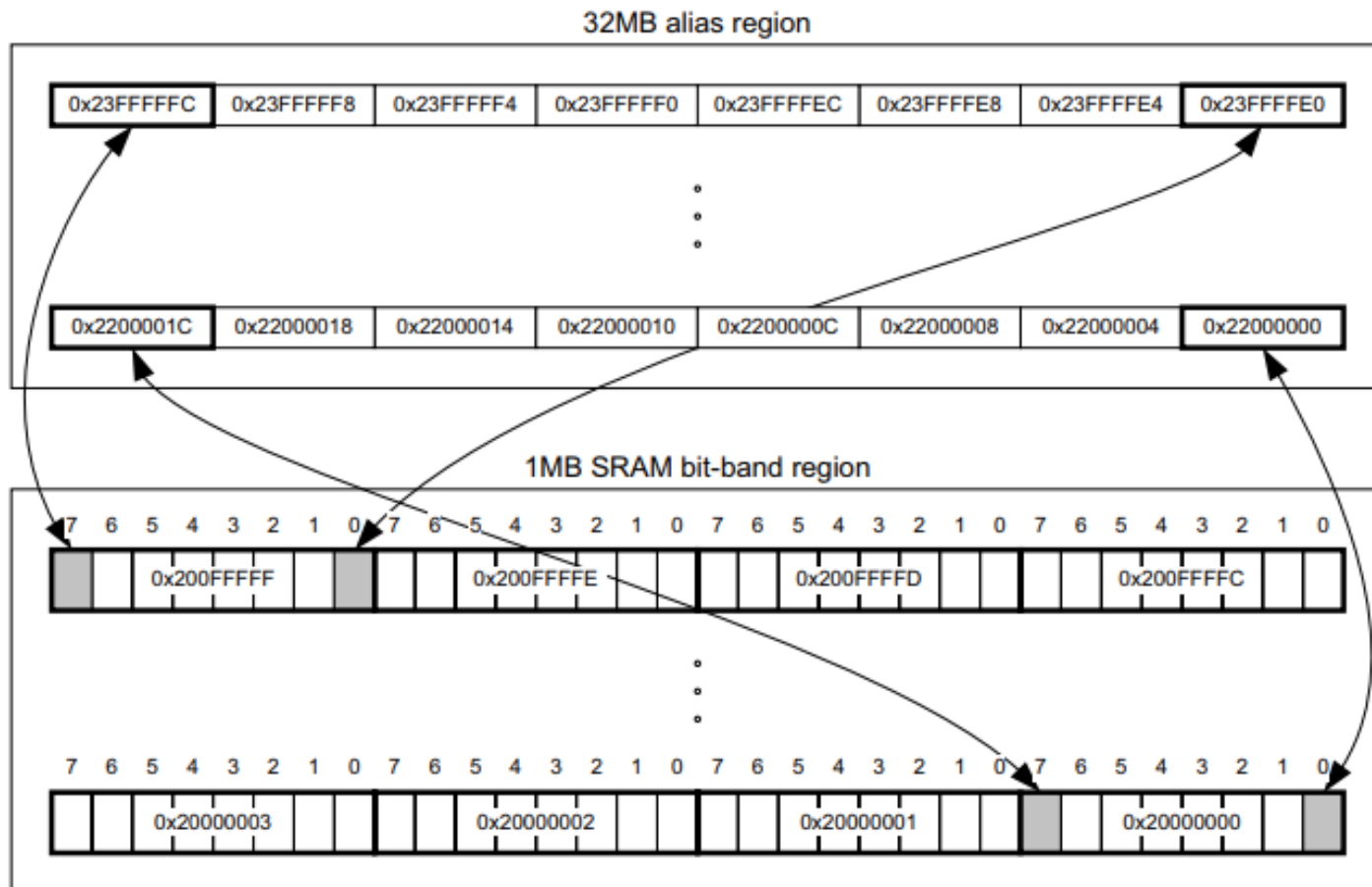
Memory Maps



-
- ▶ Cortex-M3 processor has a fixed memory map.
 - ▶ Makes it easier to port software from one cortex-M3 product to another
 - ▶ Some of the memory locations are allocated for private peripherals such as debugging components
 - ▶ They are located in private peripheral memory region
 - ▶ They include
 - ▶ Fetch patch and breakpoint Unit
 - ▶ Data watchpoint and trace unit
 - ▶ Instrumentation Trace Macrocell
 - ▶ Embedded Trace Macrocell
 - ▶ Trace port Interface Unit
 - ▶ ROM Table

-
- ▶ Cortex-M3 processor has a total of 4GB of address space
 - ▶ Program code can be located in the code region, SRAM region or external RAM region
 - ▶ Best to put the program code in code region
 - ▶ Instruction fetch and data accesses are carried out simultaneously on two separate bus interfaces
 - ▶ SRAM memory range is for connecting internal SRAM
 - ▶ Access to this region is carried out via the system interface bus
 - ▶ In this region a 32MB range is defined as a bit-band alias

▶ Bit Band Operation



▶ Peripherals

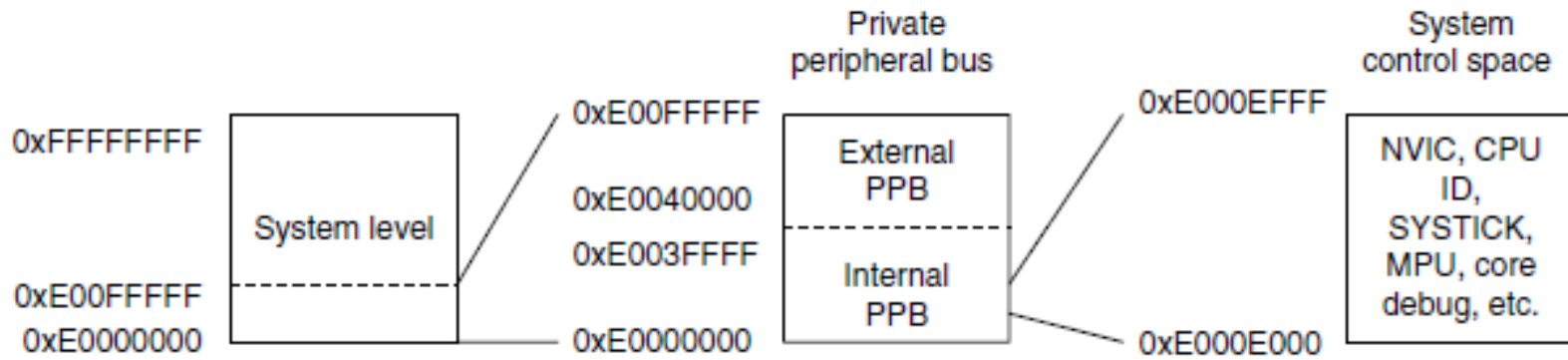
- ▶ Another 0.5 GB block of address range is allocated to on-chip peripherals.
- ▶ Similar to SRAM this region supports Bit-Band alias and is accessed via the system bus interface
- ▶ Instruction execution in this region is not allowed

▶ External RAM and external devices

- ▶ Difference between the two is the program execution in the external device region is not allowed
- ▶ Some differences with the caching behaviors

-
- ▶ Last 0.5 GB memory is for:
 - ▶ System level components
 - ▶ Internal peripheral buses
 - ▶ External peripheral bus
 - ▶ Vendor specific system peripherals
 - ▶ There are two segments of the private peripheral bus (PPB)
 - ▶ AHB(Advanced High performance bus)
 - ▶ For internal AHB peripherals only which includes NVIC, FPB, DWT and ITM
 - ▶ APB(Advanced Peripheral Bus)
 - ▶ For internal APB devices as well as external peripherals

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



Memory Access Attributes

- ▶ **Memory map defines the memory attributes of the access**
 1. **Bufferable:**
 - ▶ Write to memory can be carried out by a write buffer while processor continues on next instruction execution
 2. **Cacheable:**
 - ▶ Data obtained from memory read can be copied to a memory cache
 - ▶ Accessed value can be obtained from cache to speed up program execution
 3. **Executable:**
 - ▶ Processor can fetch and execute program code from this memory region
 4. **Sharable:**
 - ▶ Data in this region can be shared by multiple bus masters
 - ▶ Need to ensure coherency of data between different bus masters

-
- ▶ Cortex M3 bus interfaces output the memory access attributes information to the memory system for each instruction and data transfer.
 - ▶ Default attribute settings can be overridden if MPU is present.
 - ▶ Memory access attributes for each memory region are as follows:
 - ▶ Code memory region (0x00000000-0xFFFFFFFF)
 - ▶ Region is executable
 - ▶ Cache attribute is WT
 - ▶ Can put data memory in this region as well
 - ▶ Data operations will take place via data bus interface
 - ▶ Write transfers are bufferable

-
- ▶ **SRAM memory (0x20000000-0x3FFFFFFF)**
 - ▶ Intended for on-chip RAM
 - ▶ Write transfers are bufferable
 - ▶ Cache attribute is WB-WA
 - ▶ Region is executable
 - ▶ **Peripheral region (0x40000000-0x5FFFFFFF)**
 - ▶ Intended for peripherals
 - ▶ Accesses are non cacheable
 - ▶ Cannot execute instruction code in this region
 - ▶ **External RAM(0x60000000-0x7FFFFFFF)**
 - ▶ Intended for on-chip or off-chip memory
 - ▶ Accesses are cacheable WT
 - ▶ Can execute code in this region

▶ **External RAM(0x80000000-0x9FFFFFFF)**

- ▶ Intended for on-chip or off-chip memory
- ▶ Accesses are cacheable WT
- ▶ Can execute code in this region

▶ **External devices (0xA0000000-0xBFFFFFFF)**

- ▶ Intended for external devices and/or shared memory that needs ordering/non buffered accesses
- ▶ Non executable region

▶ **External devices (0xC0000000-0xDFFFFFFF)**

- ▶ Intended for external devices and/or shared memory that needs ordering/non buffered accesses
- ▶ Non executable region

▶ **System Region (0xE0000000-0xFFFFFFFF)**

- ▶ For private peripherals and vendor specific devices
- ▶ Nonexecutable
- ▶ For PPB memory range, the accesses are strongly ordered(non cacheable, nonbufferable)

Default Memory access permissions

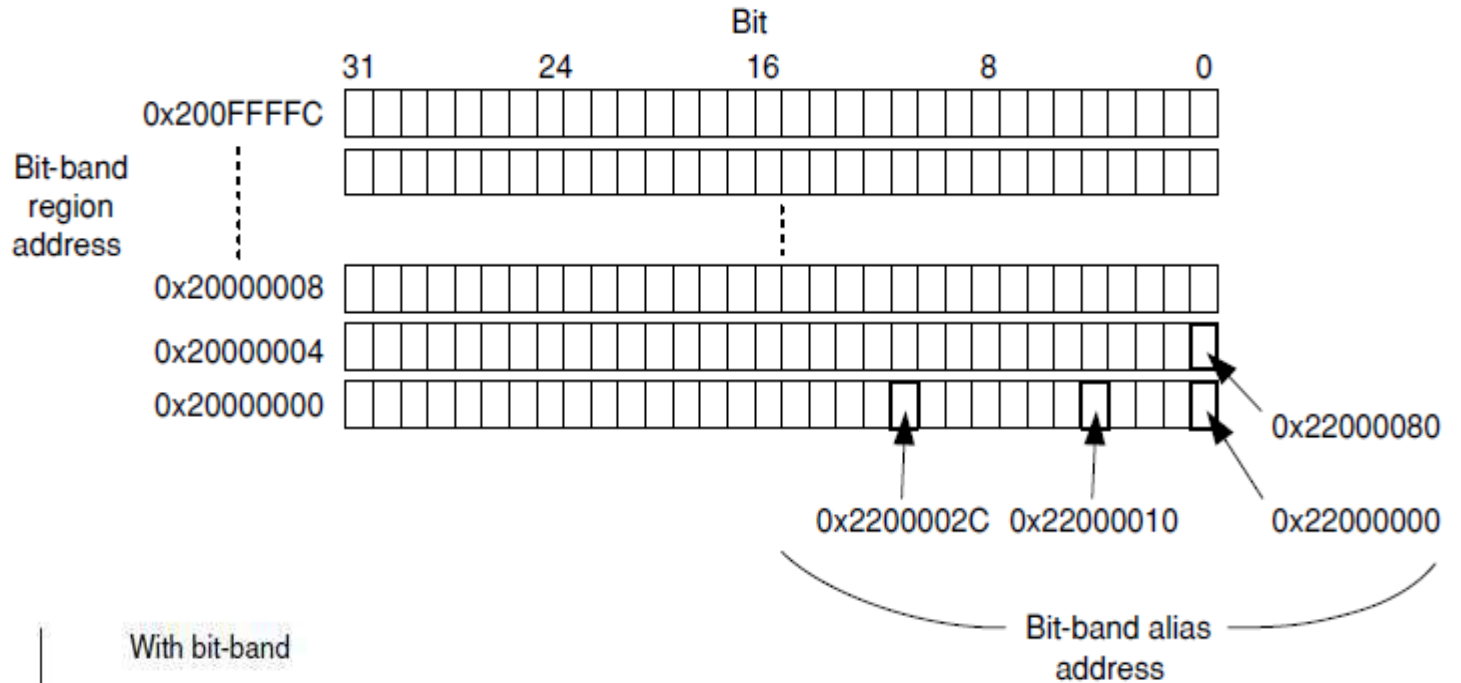
- ▶ Prevents user programs from accessing system control memory spaces
- ▶ Default memory access permission is used when there is no MPU or when MPU is disabled

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



Bit-Band Operations

- ▶ Allows a single load/store operation to access (read/write) to a single data bit
- ▶ In Cortex M3 supported in two predefined memory regions called bit-band regions
 - ▶ 1st located in IMB of SRAM region
 - ▶ 2nd located in IMB of peripheral region
- ▶ These regions can be accessed like normal memory
 - ▶ Can also be accessed via a separate memory region called bit-band alias
 - ▶ When bit-band alias is used each individual bit can be accessed separately in the LSB of each word –aligned address



Without bit-band

Read 0x20000000 to register

Set bit 2 in register

Write register to 0x20000000

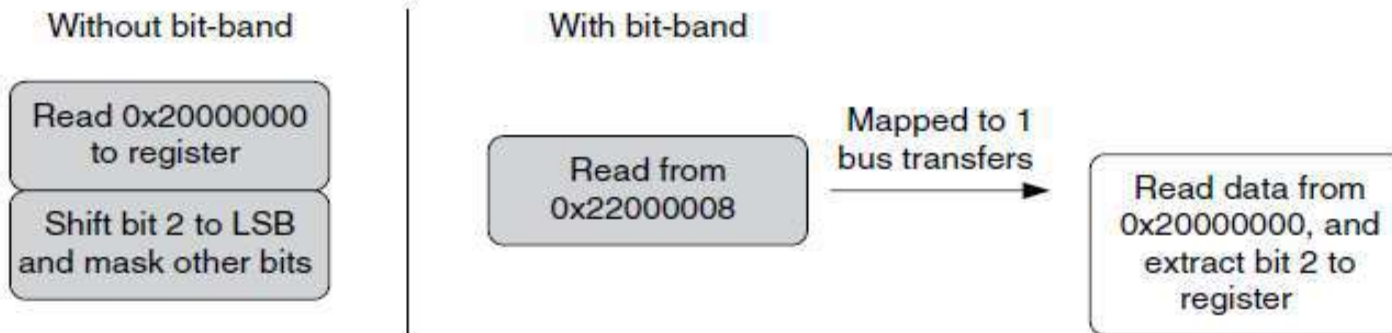
With bit-band

Write 1 to 0x22000008

Mapped to 2 bus transfers

Read data from 0x20000000 to buffer

Write to 0x20000000 from buffer with bit 2 set



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]

▶ Remapping of Bit-Band Addresses in peripheral Memory

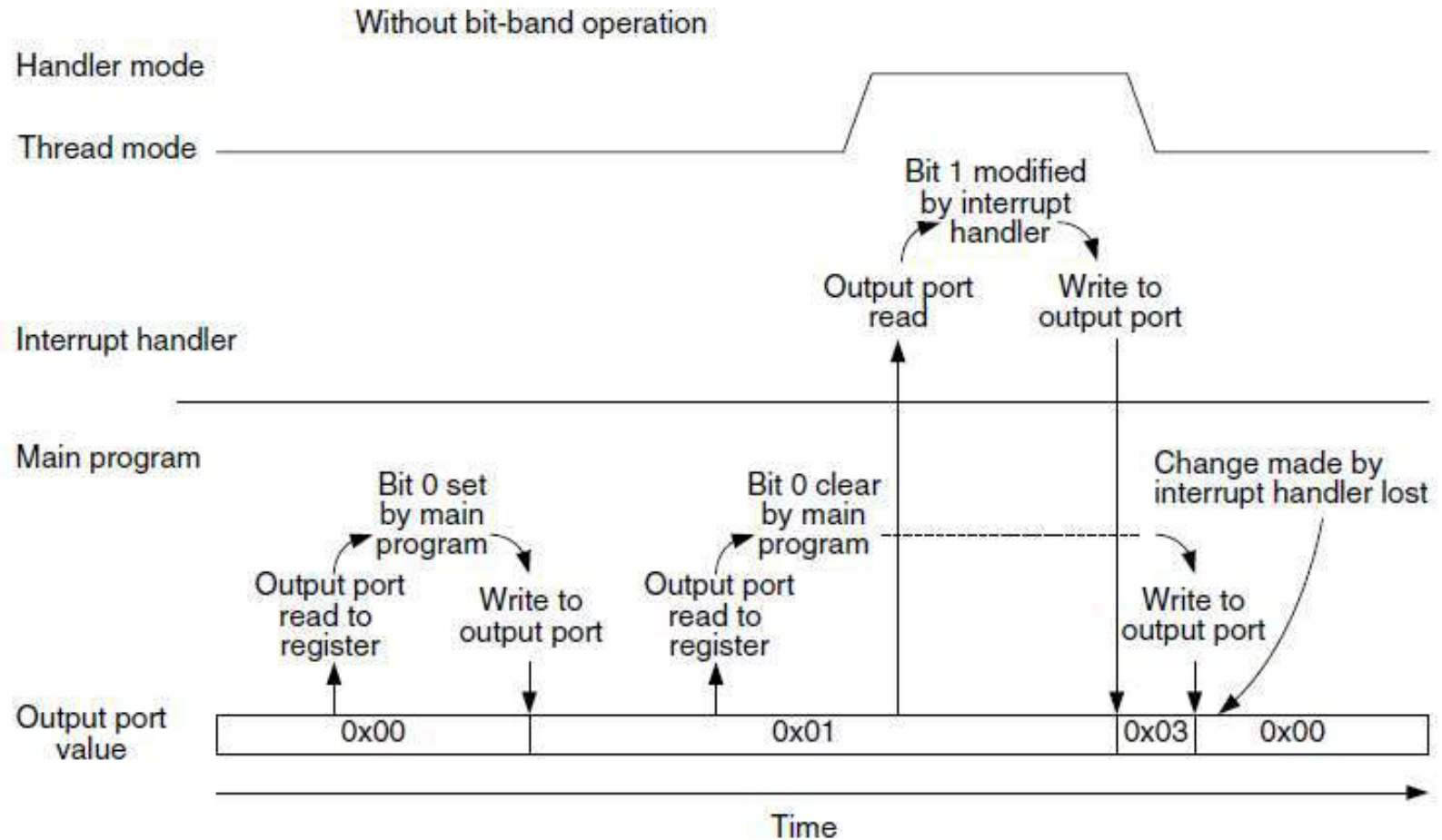
Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
...	...
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
...	...
0x40000004 bit[31]	0x420000FC bit[0]
...	...
0x400FFFFC bit[31]	0x43FFFFFC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

Advantages of Bit-Band operation

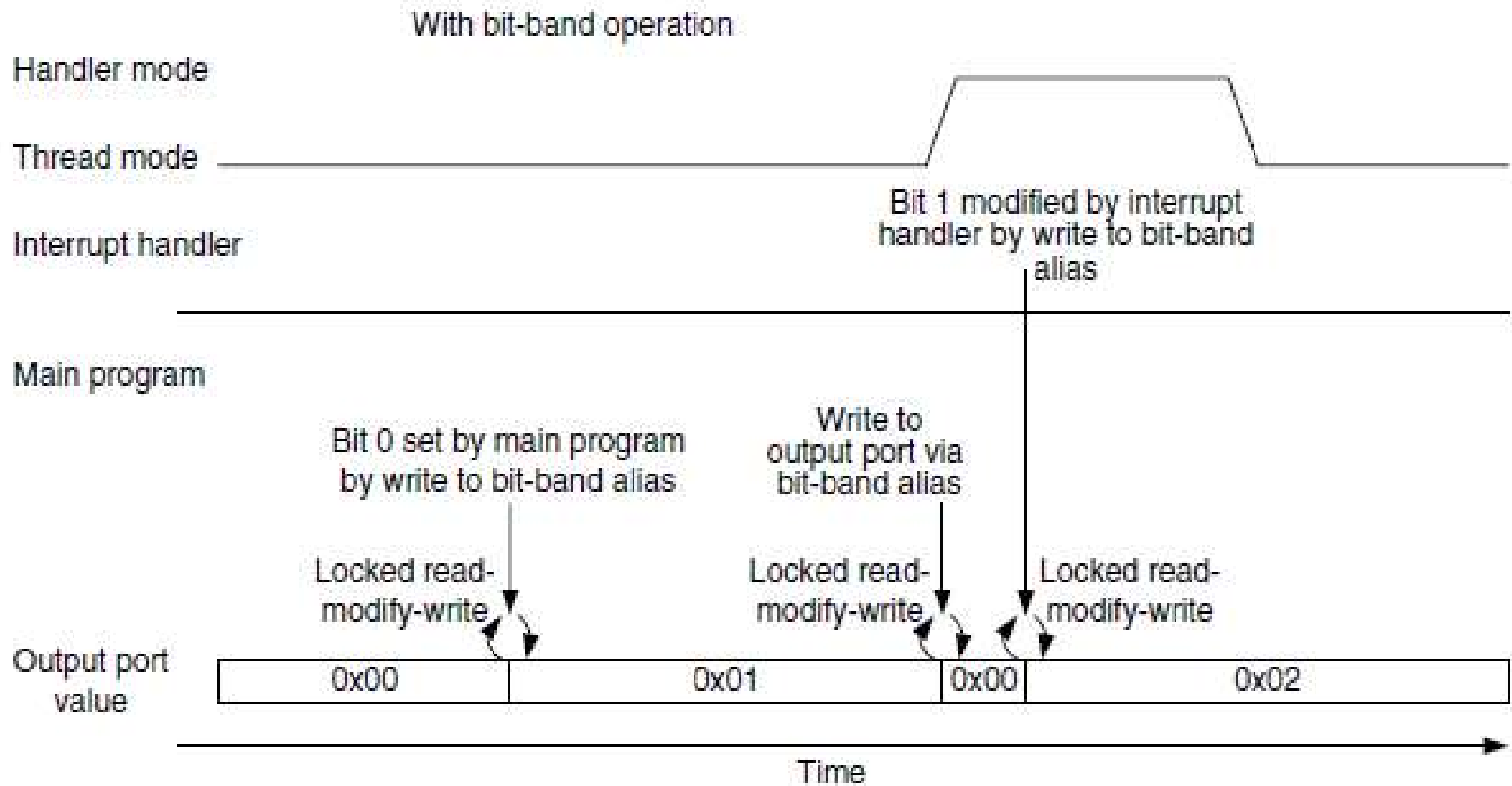
- ▶ We can use them to:
 - ▶ Implement serial data transfers in General Purpose input/output (GPIO) ports to serial devices.
 - ▶ Used to simplify branch decisions
 - ▶ If a branch should be carried out based on 1 single bit in a status register in a peripheral, instead of
 - Reading the whole register
 - Masking the unwanted bits
 - Comparing and branching
 - ▶ Operations can be simplified to
 - Reading the status bit via the bit-band alias
 - Comparing and branching

-
- ▶ Besides providing faster bit operations with fewer instructions, bit-band feature is also essential for situations in which resources are shared by more than one process
 - ▶ One of the most important advantages or properties of bit-band operation is that it is **atomic**

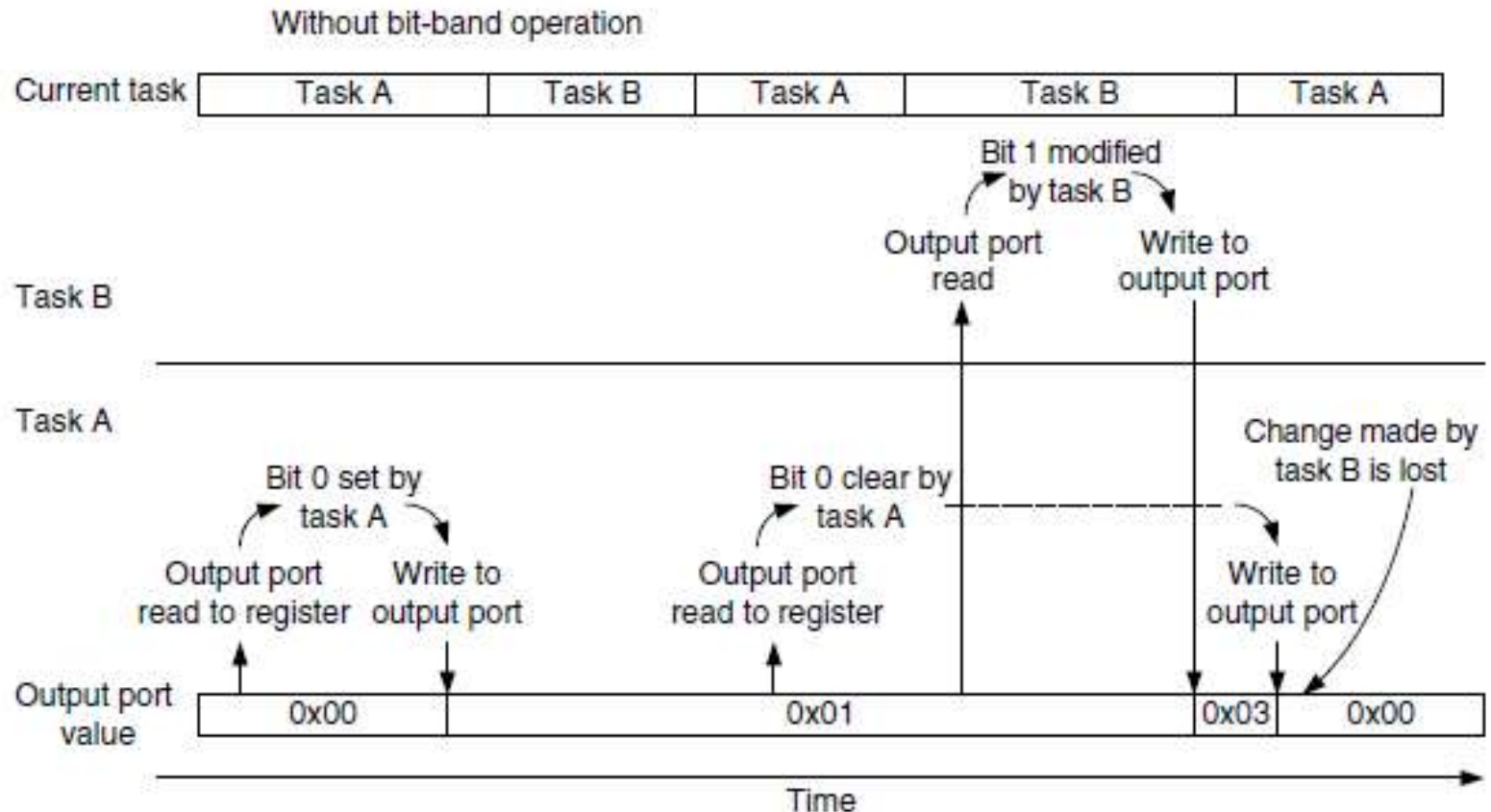
Data are lost when an exception handler modifies a shared memory location



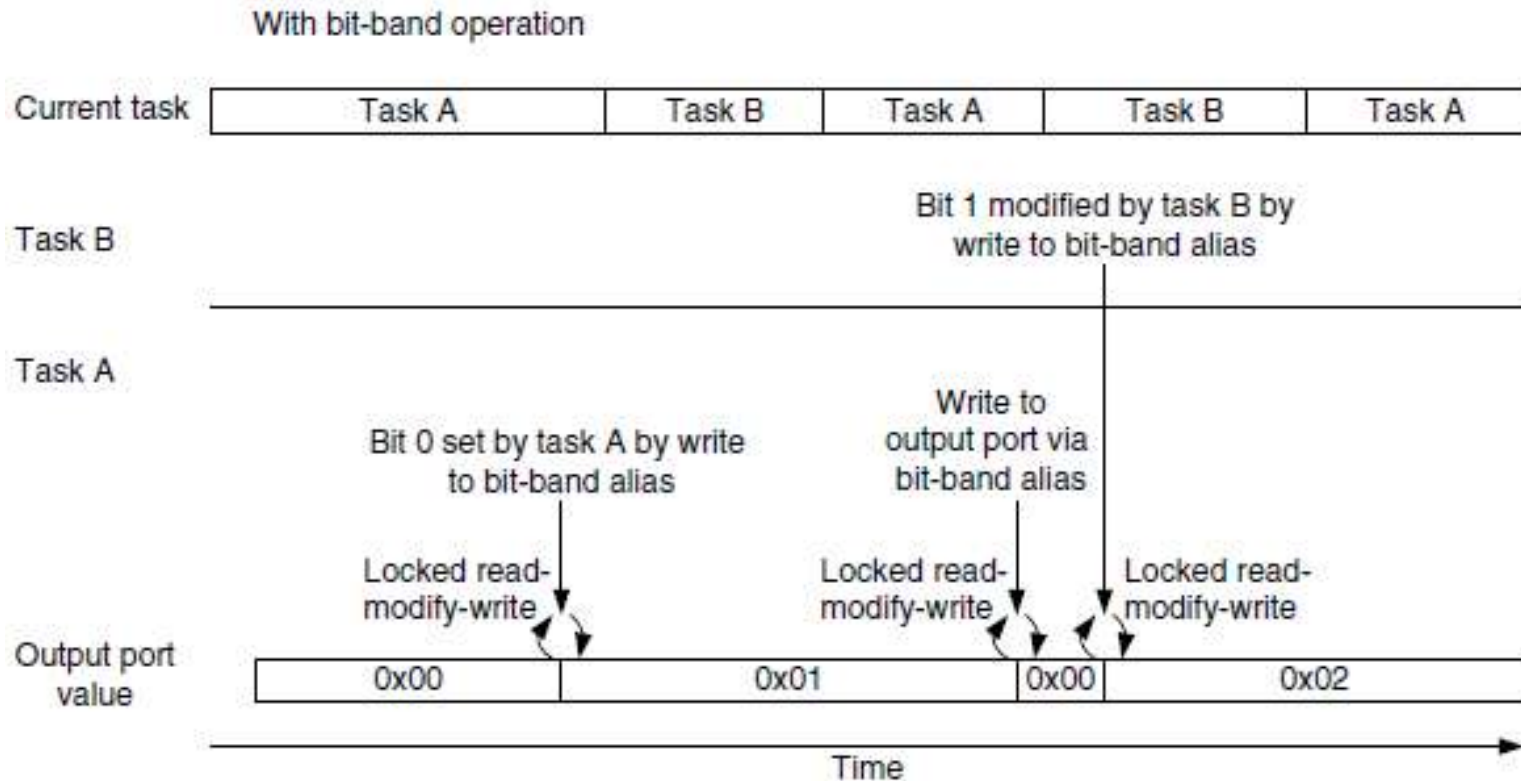
Data loss prevention with locked transfers using the bit-band feature



Data are lost when a different task modifies a shared memory location



Data loss prevention with locked transfers using the bit-band feature



▶ **Bit-Band operation of different data sizes**

- ▶ Bit-Band operation is not limited to word transfers, it can be carried out as byte transfers or half word transfers as well.
- ▶ If LDRB/STRB is used to access a bit-band alias address range, access generated to the bit-band will be in byte size
- ▶ LDRH/STRH makes access in the bit band to be halfword size

▶ **Bit-Band Operations in C**

- ▶ C compilers do not understand that the same memory can be accessed using two different addresses
- ▶ They do not know that accesses to bit-band alias will only access the LSB of the memory location.
- ▶ To use bit-band feature in C, the simplest solution is to separately declare the address and the bit-band alias of a memory location

```
#define DEVICE_REGO      *((volatile unsigned long *) (0x40000000))
#define DEVICE_REGO_BIT0 *((volatile unsigned long *) (0x42000000))
#define DEVICE_REGO_BIT1 *((volatile unsigned long *) (0x42000004))

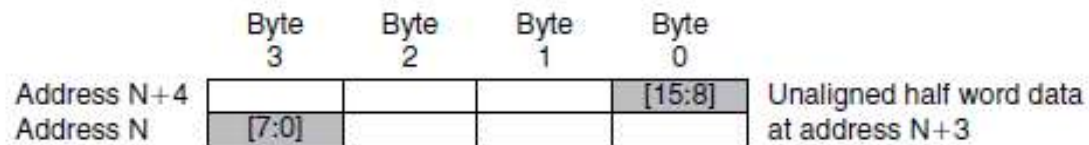
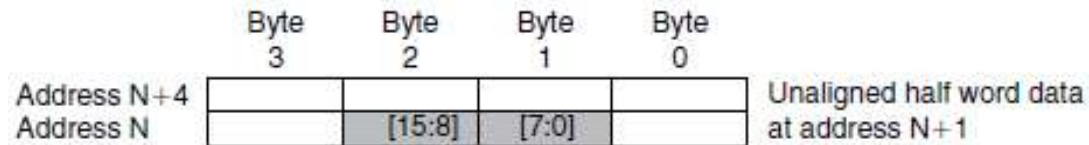
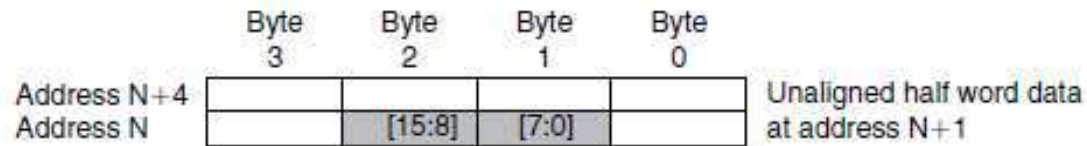
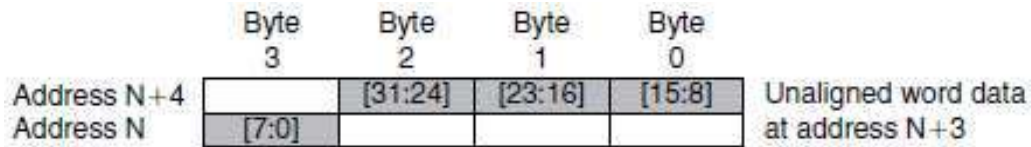
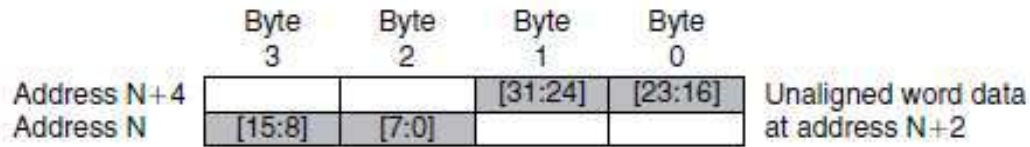
...
DEVICE_REGO = 0xAB; // Accessing the hardware register by normal
                    // address

...
DEVICE_REGO = DEVICE_REGO | 0x2; // Setting bit 1 without using
                                // bitband feature

...
DEVICE_REGO_BIT1 = 0x1; // Setting bit 1 using bitband feature
                       // via the bit band alias address
```

Unaligned Transfers

- ▶ Cortex-M3 supports unaligned transfers on single access
- ▶ Data memory accesses can be defined as aligned or unaligned
- ▶ Traditional ARM processors allow only aligned transfers
 - ▶ A word transfer must have address bit[1] and bit[0] equal to 0
 - ▶ E.g 0x1000 or 0x1004
 - ▶ A half word transfer must have address bit[0] equal to 0
 - ▶ E.g 0x1000 or 0x1002
- ▶ Unaligned can be any word size read/write such that address is not a multiple of 4



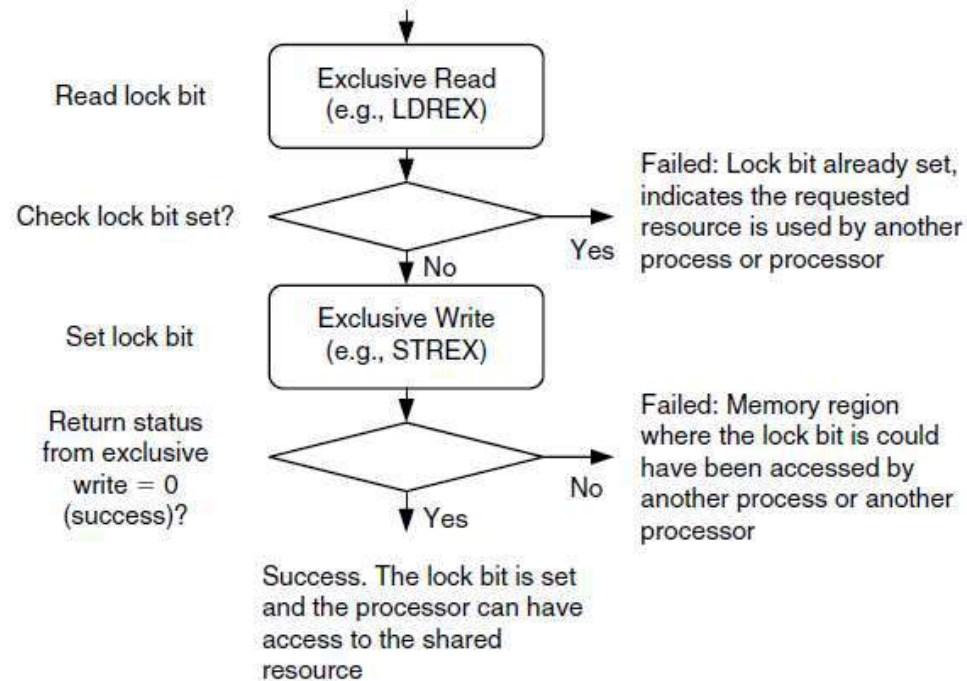
▶ Limitation of Unaligned Transfers

- ▶ Not supported in load/store multiple instructions
 - ▶ Stack operations must be aligned
 - ▶ Exclusive accesses must be aligned; otherwise a fault exception will be triggered
 - ▶ Unaligned transfers are not supported in bit-band
 - ▶ Results will be unpredictable if you attempt to do so
-
- ▶ Unaligned transfers are actually converted into multiple aligned transfers
 - ▶ Takes more clock cycles for a single data access
 - ▶ Not good for situations in which high performance is required

Exclusive accesses

- ▶ **Semaphores** are commonly used for allocating shared resources to applications
- ▶ When a shared resource can only service one client or application processor is called as MUTEX
 - ▶ When a resource is being used by one process, it is locked to that process
 - ▶ It cannot serve another process until the lock is released
- ▶ To set up MUTEX semaphore
 - ▶ A Memory location is defined as the lock flag
 - ▶ Lock flag indicates whether a shared resource is locked by a process

-
- ▶ **When a process or application wants to use the resource**
 - ▶ It needs to check whether the resource has been locked first
 - ▶ If not being used, it can set the lock flag to indicate the resource is now locked
 - ▶ **In traditional ARM processors, the access to the flag is carried out by the SWP instruction**
 - ▶ Allows the lock flag read and write to be atomic
 - ▶ Prevents the resource from being locked by two processes at the same time
 - ▶ **In newer ARM processors the read/write access can be carried out on separate bus**
 - ▶ SWP instructions can no longer be used to make the memory access atomic
 - ▶ Because the read and write in a locked transfer sequence must be on the the same bus
 - ▶ The locked transfers are replaced by exclusive access



- ▶ Exclusive access to work properly in a multiprocessor environment requires “exclusive access monitor”.
- ▶ Monitor checks:
 - ▶ the transfers toward shared address locations
 - ▶ Replies to the processor if an exclusive access is success
 - ▶ Processor bus interface also provides additional control signals to indicate if the transfer is an exclusive access

Endian Mode

▶ **Big Endian**

- ▶ The first byte of a word size data is stored in the most significant byte of the 32 bit address memory location

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
...	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

▶ **Little Endian**

- ▶ The first byte of the word size data is stored in the least significant byte of the 32 bit memory location

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
...	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

Cortex-M3 (Byte Invariant Big endian, BE-8)-Data on the AHB Bus

Address	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000 word	Data bit [7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000 half word	-	-	Data Bit [7:0]	Data bit [15:8]
0x1002, half word	Data bit[7:0]	Data bit[15:8]	-	-
0x1000, byte	Data bit [7:0]	-	-	-
0x1001, byte	-	Data bit[7:0]	-	-
0x1002,byte	-	-	Data bit[7:0]	-
0x1003 byte	-	-	-	Data bit[7:0]

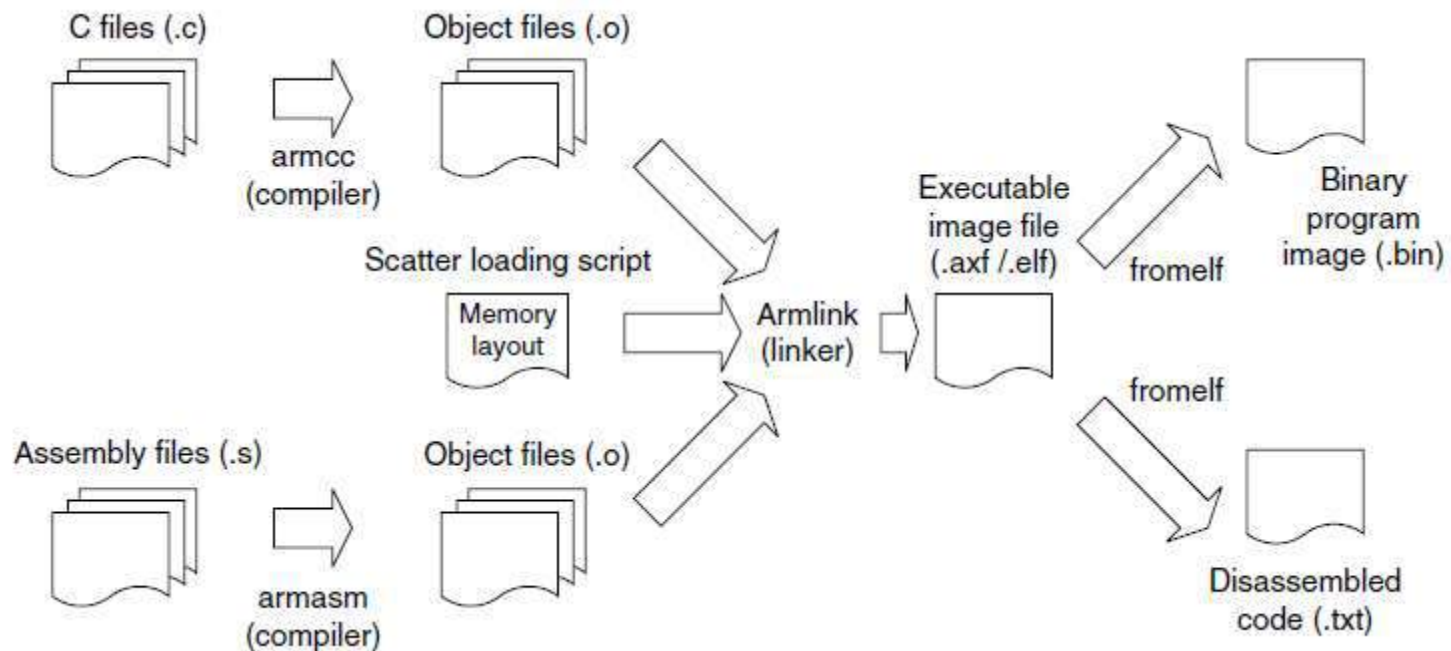
ARM 7 TDMI (Word-Invariant Big Endian, BE-32) Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000, half word	Data bit [7:0]	Data bit [15:8]	—	—
0x1002, half word	—	—	Data bit [7:0]	Data bit [15:8]
0x1000, byte	Data bit [7:0]	—	—	—
0x1001, byte	—	Data bit [7:0]	—	—
0x1002, byte	—	—	Data bit [7:0]	—
0x1003, byte	—	—	—	Data bit [7:0]

Cortex M3 little Endian- Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [31:24]	Data bit [23:16]	Data bit [15:8]	Data bit [7:0]
0x1000, half word	—	—	Data bit [15:8]	Data bit [7:0]
0x1002, half word	Data bit [15:8]	Data bit [7:0]	—	—
0x1000, byte	—	—	—	Data bit [7:0]
0x1001, byte	—	—	Data bit [7:0]	—
0x1002, byte	—	Data bit [7:0]	—	—
0x1003, byte	Data bit [7:0]	—	—	—

- ▶ ARM cortex-M3 can be programmed using either assembly language, C language, or other high level languages like National Instruments Lab View.
- ▶ **Typical Development Flow**



-
- ▶ For beginners in embedded programming, using C language for software development on the Cortex-M3 processor is the best choice.
 - ▶ Programming in C with the Cortex-M3 processor is made even easier as most microcontroller vendors provide device driver libraries written in C to control peripherals.
 - ▶ Since modern C compilers can generate very efficient code, it is better to program in C than spending a lot of time to try to develop complex routines in assembly language, which is error prone and less portable.
 - ▶ C has the advantage of being portable and easier for implementing complex operations, compared with assembly language. Since it's a generic computer language, C does not specify how the processor is initialized.

```

#define LED *((volatile unsigned int *) (0xDFFF000C))

int main (void)
{
    int i;          /* loop counter for delay function */
    volatile int j; /* dummy volatile variable to prevent
                    C compiler from optimize the delay away */

    while (1) {
        LED = 0x00; /* toggle LED */
        for (i=0;i<10;i++) {j=0;} /* delay */
        LED = 0x01; /* toggle LED */
        for (i=0;i<10;i++) {j=0;} /* delay */
    }
    return 0;
}

typedef void(* const ExecFuncPtr)(void) __irq;
extern int __main(void);

/*
 * Dummy handlers Exception Handlers
 */
__irq void NMI_Handler(void)
{ while(1); }
__irq void HardFault_Handler(void)
{ while(1); }
__irq void SVC_Handler(void)
{ while(1); }
__irq void DebugMon_Handler(void)
{ while(1); }
__irq void PendSV_Handler(void)
{ while(1); }
__irq void SysTick_Handler(void)
{ while(1); }
__irq void ExtInt0_IRQHandler(void)
{ while(1); }
__irq void ExtInt1_IRQHandler(void)
{ while(1); }
__irq void ExtInt2_IRQHandler(void)
{ while(1); }
__irq void ExtInt3_IRQHandler(void)
{ while(1); }

```

```

ExecFuncPtr exception_table[] = { /* vector table */
    (ExecFuncPtr)0x20002000,
    (ExecFuncPtr)__main,
    NMI_Handler, /* NMI */
    HardFault_Handler,
    0, /* MemManage_Handler in Cortex-M3 */
    0, /* BusFault_Handler in Cortex-M3 */
    0, /* UsageFault_Handler in Cortex-M3 */

    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler,
    0, /* DebugMon_Handler in Cortex-M3 */
    0, /* Reserved */
    PendSV_Handler,
    SysTick_Handler,

    /* External Interrupts*/
    ExtInt0_IRQHandler,
    ExtInt1_IRQHandler,
    ExtInt2_IRQHandler,
    ExtInt3_IRQHandler
};
#pragma arm section

```

```

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

```

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

```
LOAD_REGION 0x00000000 0x00200000
```

```
{
  VECTORS 0x0 0xC0
  {
    : Provided by the user in vectors.c
    * (exceptions_area)
  }
}
```

```
CODE 0xC0 FIXED
```

```
{
  * (+RO)
}
```

```
DATA 0x20000000 0x00010000
```

```
{
  * (+RW, +ZI)
}
```

:: Heap starts at 4KB and grows upwards

```
ARM_LIB_HEAP HEAP_BASE EMPTY HEAP_SIZE
{
}
```

:: Stack starts at the end of the 8KB of RAM
 :: And grows downwards for 2KB

```
ARM_LIB_STACK STACK_BASE EMPTY -STACK_SIZE
{
}
```

```
}
```

```
$> armlink -scatter led.scat "--keep=vectors.o(exceptions_area)"
      blinky.o vectors.o -o blinky.elf
```

```
/* create binary file */
$> fromelf --bin blinky.elf -output blinky.bin
/* Create disassembly output */
$> fromelf -c blinky.elf > list.txt
```



```

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

LOAD_REGION 0x00000000 0x00200000
{
  VECTORS 0x0 0xC0
  {
    : Provided by the user in vectors.c
    * (exceptions_area)
  }

  CODE 0xC0: FIXED
  {
    * (+RO)
  }

  DATA 0x20000000 0x00010000
  {
    * (+RW, +ZI)
  }

  ;; Heap starts at 4KB and grows upwards
  Heap_Mem HEAP_BASE EMPTY HEAP_SIZE
  {
  }

  ;; Stack starts at the end of the 8KB of RAM
  ;; And grows downwards for 2KB
  Stack_Mem STACK_BASE EMPTY -STACK_SIZE
  {
  }
}

```

```

SET PATH=C:\Keil\ARM\BIN40\;%PATH%
SET RVCT40INC=C:\Keil\ARM\RV31\INC
SET RVCT40LIB=C:\Keil\ARM\RV31\LIB
SET CPU_TYPE=Cortex-M3
SET CPU_VENDOR=ARM
SET UV2_TARGET=Target 1
SET CPU_CLOCK=0x00000000
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM vectors.c
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM blinky.c
C:\Keil\ARM\BIN40\armlink --device DLM "--keep=Startup.o(RESET)"
"--first=Startup.o(RESET)" -scatter led.scats --map vectors.o
blinky.o -o blinky.elf
C:\Keil\ARM\BIN40\fromelf --bin blinky.elf -o blinky.bin

```

Accessing memory mapped registers in C

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

```
/* Setup SYSTICK */
```

```
SYSTICK_LOAD = 0xFFFF; // Set reload value
```

```
SYSTICK_VAL = 0x0; // Clear current value
```

```
SYSTICK_CTRL = 0x5; // Enable SYSTICK and select core clock
```

CALIB

VALUE

RELOAD

CTRL

0xE000E01C

0xE000E018

0xE000E014

0xE000E010

SYSTICK
Timer
registers

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

```
/* Setup SysTick */
```

```
HW_REG(SYSTICK_LOAD) = 0xFFFF; // Set reload value
```

```
HW_REG(SYSTICK_VAL) = 0x0; // Clear current value
```

```
HW_REG(SYSTICK_CTRL) = 0x5; // Enable SYSTICK and select core clock
```

CALIB

VALUE

RELOAD

CTRL

0xE000E01C

0xE000E018

0xE000E014

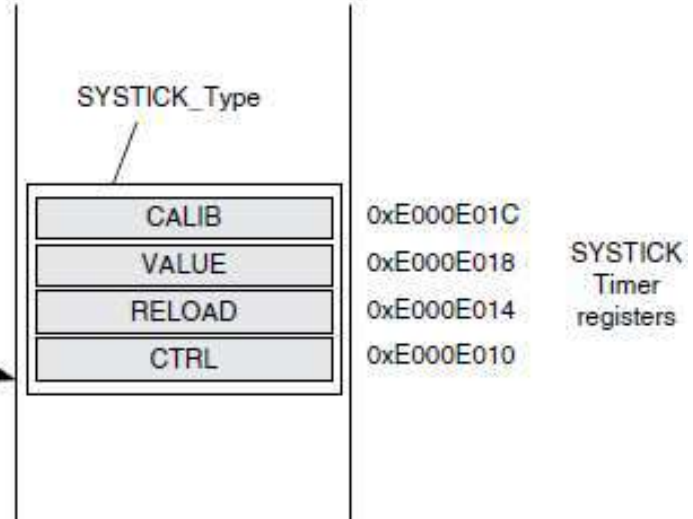
0xE000E010

SYSTICK
Timer
registers

```
typedef struct
{
    volatile unsigned long CTRL; /* SysTick Control and Status register */
    volatile unsigned long LOAD; /* SysTick Reload Value register */
    volatile unsigned long VAL; /* SysTick Current Value register */
    volatile unsigned long CALIB; /* SysTick Calibration register */
} SysTick_Type;

#define SysTick ((SysTick_Type *) 0xE000E010) /* SysTick struct */

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

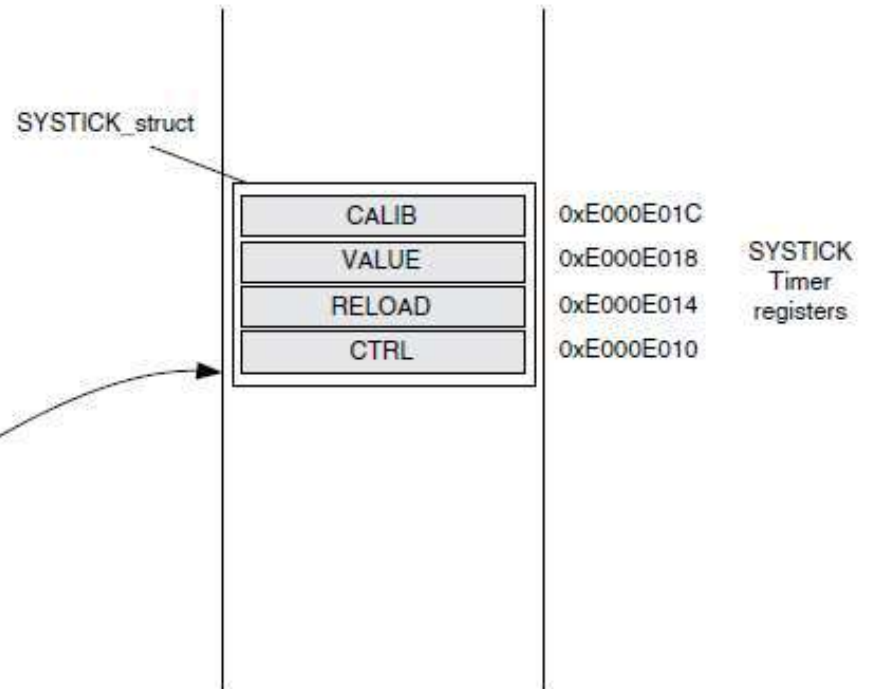


In the C file, define the data structure as

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

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

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



Intrinsic function

- ▶ Use of the C language can often speed up application development
- ▶ In some cases, we need to use some instructions that cannot be generated using normal C-code
- ▶ Some C compilers provide intrinsic functions for accessing these special instructions
- ▶ Intrinsic functions are used just like normal C functions.

Assembly Instructions

CLZ
CLREX
CPSID I
CPSIE I
CPSID F
CPSIE F
LDREX/LDREXB/LDREXH
LDRT/LDRBT/LDRSBT/LDRHT/LDRSHT
NOP
RBIT
REV
ROR
SSAT
SEV
STREX/STREXB/STREXH
STRT/STRBT/STRHT
USAT
WFE
WFI
BKPT

ARM Compiler Intrinsic Functions

unsigned char __clz(unsigned int val)
void __clrex(void)
void __disable_irq(void)
void __enable_irq(void)
void __disable_fiq(void)
void __enable_fiq(void)
unsigned int __ldrex(volatile void *ptr)
unsigned int __ldrt(const volatile void *ptr)
void __nop(void)
unsigned int __rbit(unsigned int val)
unsigned int __rev(unsigned int val)
unsigned int __ror(unsigned int val, unsigned int shift)
int __ssat(int val, unsigned int sat)
void __sev(void)
int __strex(unsigned int val, volatile void *ptr)
void int __strt(unsigned int val, const volatile void *ptr)
int __usat(unsigned int val, unsigned int sat)
void __wfe(void)
void __wfi(void)
void __breakpoint(int val)

Embedded Assembler and Inline Assembler

- ▶ As an alternative to using intrinsic functions, we can also directly access assembly instructions in C code
- ▶ This is often necessary in low-level system control or when you need to implement a timing critical routine and decide to implement it in assembly for the best performance.
- ▶ Most ARM C compilers allow you to include assembly code in form of *inline assembler*.

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

Using Assembly

- ▶ For small projects, it is possible to develop the whole application in assembly language.
- ▶ However, this is often much harder for beginners.
- ▶ Using assembler, you might be able to get the best optimization you want, though it might increase your development time, and it could be easy to make mistakes.
- ▶ Handling complex data structures or function library management can be extremely difficult in assembler.
- ▶ Even when the C language is used in a project, in some situations part of the program is implemented in assembly language as follows:
 - ▶ Functions that cannot be implemented in C, such as direct manipulation of stack data or special instructions that cannot be generated by the C compiler in normal C-code
 - ▶ Timing-critical routines
 - ▶ Tight memory requirements, causing part of the program to be written in assembly to get the smallest memory size

Interface between Assembly and C

- ▶ When embedded assembly (or inline assembler, in the case of the GNU tool chain) is used in C program code
- ▶ When C program code calls a function or subroutine implemented in assembler in a separate file
- ▶ When an assembly program calls a C function or subroutine
- ▶ It is important to understand how parameters and return results are passed between the calling program and the function being called

First Step in Assembly programming

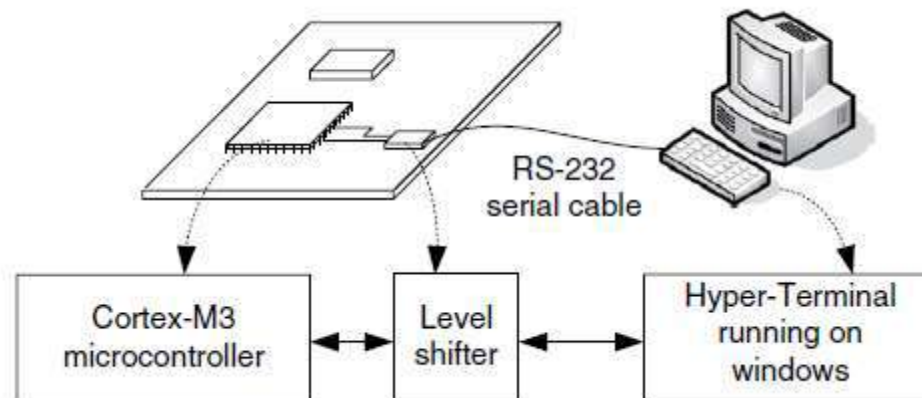
```
STACK_TOP EQU 0x20002000; constant for SP starting value

AREA |Header Code |, CODE
DCD STACK_TOP ; Stack top

DCD Start      ; Reset vector
ENTRY          ; Indicate program execution start here
Start ; Start of main program
; initialize registers
MOV r0, #10    ; Starting loop counter value
MOV r1, #0     ; starting result
; Calculated 10+9+8+...+1
loop
ADD r1, r0     ; R1 = R1 + R0
SUBS r0, #1    ; Decrement R0, update flag ("S" suffix)
BNE loop      ; If result not zero jump to loop
; Result is now in R1
deadloop
B deadloop    ; Infinite loop
END           ; End of file
```

Producing Outputs

- ▶ Its more fun to connect microcontroller to the outside world
 - ▶ Simplest way to do this is to turn on/off the LEDs
 - ▶ Represents very limited information
 - ▶ Most common output method is to send text messages to a console
 - ▶ Often handled by a UART interface connecting to a personal computer



-
- ▶ The cortex-M3 processor does not contain a UART interface
 - ▶ Most cortex-M3 microcontrollers come with UART provided by the chip manufacturers
 - ▶ Assuming UART is available
 - ▶ It has a status flag to indicate whether the transmit buffer is ready to send out new data
 - ▶ Level shifter is needed
 - ▶ UART is not the only solution to output text messages
 - ▶ A number of features are implemented on the cortex-M3 processor to help output debugging messages

▶ Semihosting

- ▶ Depending on the debugger and code library support semihosting can be done via debug register in NVIC

▶ Instrumentation Trace

- ▶ Instead of using UART to output messages, we can use ITM
- ▶ Trace port works much faster than UART and can offer more data channels

▶ Instrumentation trace via Serial-Wire Viewer (SWV)

- ▶ Allows outputs from ITM to be captured using low cost hardware instead of a TPA
- ▶ Bandwidth provided is limited, hence not ideal for large amounts of data

Using Data Memory

```
STACK_TOP EQU 0x20002000 ; constant for SP starting value
          AREA | Header Code|, CODE
          DCD STACK_TOP ; SP initial value
          DCD Start ; Reset vector
          ENTRY
Start ; Start of main program
      ; initialize registers
      MOV r0, #10 ; Starting loop counter value
      MOV r1, #0 ; starting result
      ; Calculated 10+9+8+...+1
Loop
      ADD r1, r0 ; R1 = R1 + R0
      SUBS r0, #1 ; Decrement R0, update flag ("S"
                  ; suffix)
      BNE loop ; If result not zero jump to loop
      ; Result is now in R1
      LDR r0, =MyData1 ; Put address of MyData1 into R0
      STR r1, [r0] ; Store the result in MyData1
deadloop
      B deadloop ; Infinite loop
          AREA | Header Data|, DATA
          ALIGN 4
MyData1 DCD 0 ; Destination of calculation result
MyData2 DCD 0
          END ; End of file
```

EMBEDDED SYSTEM COMPONENTS

Module 3

What is an Embedded System???

2

- An electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software)
- Embedded systems are becoming an inevitable part of any product or equipment in all fields
 - Household appliances
 - Telecommunications
 - Medical equipment
 - Industrial control
 - Consumer products

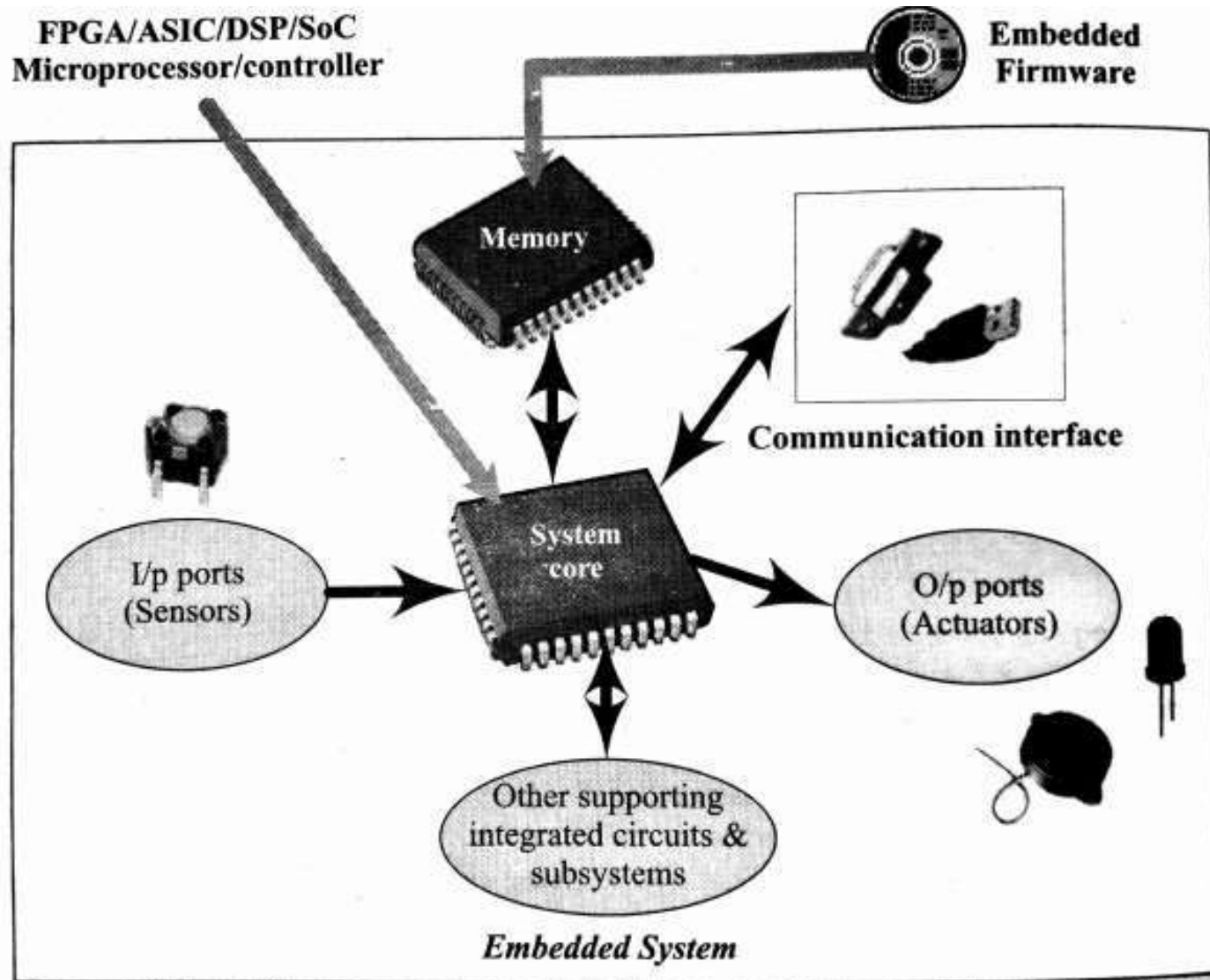
General Purpose Computing System

3

- A system which is a combination of a generic hardware and a general purpose operating system for executing a variety of applications
- Contains a GPOS
- Applications are alterable
- Performance is the key deciding factor in the selection of the system
- Less/not at all tailored towards reducing operating power requirements
- Response requirements are not critical
- Need not be deterministic in execution behavior

Embedded System

- System which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
- May or may not contain an OS for functioning
- Firmware of the embedded system is pre-programmed and it is non alterable by the end user
- Application specific requirements are the key deciding factors
- Highly tailored to take advantage of the power saving modes supported by hardware and OS
- For certain systems, the response time requirement is highly critical
- Execution behavior is deterministic for certain types of embedded systems



- Embedded hardware/software systems are basically designed
 - ▣ To regulate a physical variable
 - ▣ To manipulate the state of some devices by sending some control signals to the actuators or device connected
 - ▣ The above are done in response to the input signals provided by the end users or sensors
- Embedded system can be viewed as a reactive system
- Some embedded systems do not require any manual intervention for their operation.
 - ▣ They automatically sense the variations in input parameters in accordance with the changes in the real world through sensors

- ▣ Sensor information is passed to the processor
- ▣ The processor or the brain of the embedded system performs pre-defined operation with the help of firmware embedded in the system
- ▣ Sends some actuating signals to the actuator connected to the output port
 - This in turn acts on the control variable to bring the variable to the desired level
 - Make the embedded system work in a desired manner
- ▣ Memory of the ES is responsible for holding the control algorithm and other important configuration details

Core of the Embedded System

7

- Embedded systems are domain and application specific and built around a central core.
 1. General purpose and domain specific
 1. Microprocessors
 2. Microcontrollers
 3. Digital Signal Processors
 2. ASICS
 3. PLDS
 4. Commercial off the shelf components (COTS)

General Purpose and Domain Specific Processors

8

- Almost 80% of the embedded systems are processor/controller based.
- Microprocessors
 - ▣ It's a silicon chip representing a CPU
 - Contains ALU and working registers
 - ▣ It's a dependent unit
 - Memory, Timer Unit and Interrupt Controller
- Intel claims the credit for developing the first microprocessor unit Intel 4004
 - ▣ 8080,8085, 8086 etc

- Different Instruction set and system architecture are available for the design of a microprocessor
- System Architectures
 - Harvard
 - Contains separate buses for program and data memory
 - Von-Neumann
 - Shares a single bus for program and data memory
- Instruction set Architecture (ISA)
 - RISC
 - CISC

GPP vs Application specific instruction set processor

10

□ GPP

- Designed for general computational tasks
- Produced in large volume targeting the general market
 - Per unit cost is low
- Contains ALU and CU

□ ASIP

- Architecture and instruction set optimized to specific-domain or application requirements
 - Network processing, automotive, telecom, media applications, DSP, control applications
- Fill the architectural spectrum between GPP and ASIC
- Need arises when GPP are unable to meet the increasing application needs
- Automotive AVR, USB AVR from Atmel
- Incorporate processor and on-chip peripherals, demanded by the application requirement, program and data memory

□ **Microcontrollers**

- Highly integrated circuit that consists of
 - CPU
 - scratch pad RAM
 - General purpose register arrays
 - On chip ROM/FLASH for program storage
 - Timer and interrupt control units
 - Dedicated I/O ports
- Considered as superset of microprocessors
- Finds greater place in embedded domain
- Cheap, cost effective and readily available in the market
- ISA can be either RISC or CISC
- Designed for either general purpose application requirement or domain specific application requirement

Microprocessor

- A silicon chip representing a CPU, capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions
- It's a dependent Unit
- General purpose in design and operation
- Doesn't contain a built-in I/O port, functionality needs to be implemented with help of 8255 PPI
- Targeted for high end market where performance is important
- Limited power saving options

Microcontroller

- Highly integrated chip that contains CPU, scratch pad RAM, special and general purpose register arrays, on chip ROM/FLASH memory, timer, interrupt controller and dedicated I/O ports
- It is a self contained Unit
- Mostly application oriented or domain specific
- Most of them contain multiple built-in I/O ports which can be operated as single 8 or 16 or 32 bit port or as individual port pins
- Targeted for embedded market where performance is not so critical
- Includes lot of power saving features

□ Digital Signal Processors

- Powerful 8/16/32 bit microprocessors
- Designed specifically to meet the computational demands and power constraints of embedded audio, video and communications applications
- DSPs are 2 to 3 times faster than general purpose microprocessor in signal processing applications
 - This is because of the architectural difference
- DSPs implement algorithm in hardware
 - Speeds up the execution
- GPP implement algorithm in firmware
 - Speed of execution depends primarily on the clock for the processors

□ Typical DSP incorporates the following units

■ **Program Memory**

- For storing 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 signal processing in accordance with the stored program memory
- Incorporates many specialized arithmetic units and each of them operate simultaneously to increase the execution speed
- Incorporates multiple hardware shifters for shifting operands and thereby saves execution time

■ IO Unit

- Acts as an interface between outside world and DSP
 - Responsible for capturing signals and delivering the processed signals
 - DSP employs a large amount of real-time calculations
 - SOP
 - Convolution
 - FFT
 - Discrete Fourier Transform
- Blackfin processors from analog devices is an example of DSP

RISC

- ❑ Less number of instructions
- ❑ Instruction pipelining and increased execution speed
- ❑ Orthogonal instruction set
- ❑ Operations are performed on registers only, the only memory operations are load and store
- ❑ A large number of register are available
- ❑ Programmer needs to write more code to execute a task
- ❑ Single, fixed length instructions
- ❑ Less silicon usage and pin count
- ❑ With Harvard architecture

CISC

- ❑ Greater number of instructions
- ❑ No instruction pipelining feature
- ❑ Non-orthogonal instruction set
- ❑ Operations are performed on register or memory depending on the instruction
- ❑ Limited number of general purpose registers
- ❑ Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction
- ❑ Variable length instruction
- ❑ More silicon usage since more additional decoder logic is required to implement complex instruction decoding
- ❑ Can be harvard or Von-Neumann architecture

Harvard Architecture

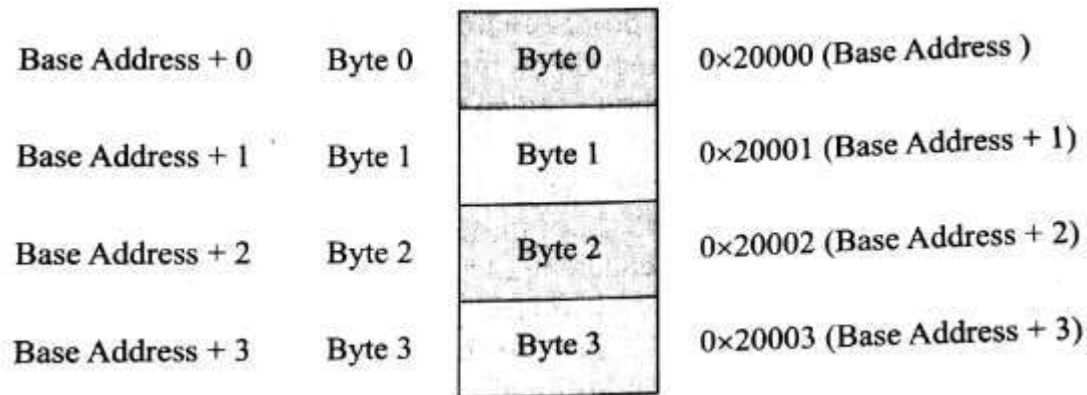
- Separate buses for instruction and data fetching
- Easier to pipeline, so high performance can be achieved
- Comparatively high cost
- No memory alignment problems
- Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory

Von-Neuman Architecture

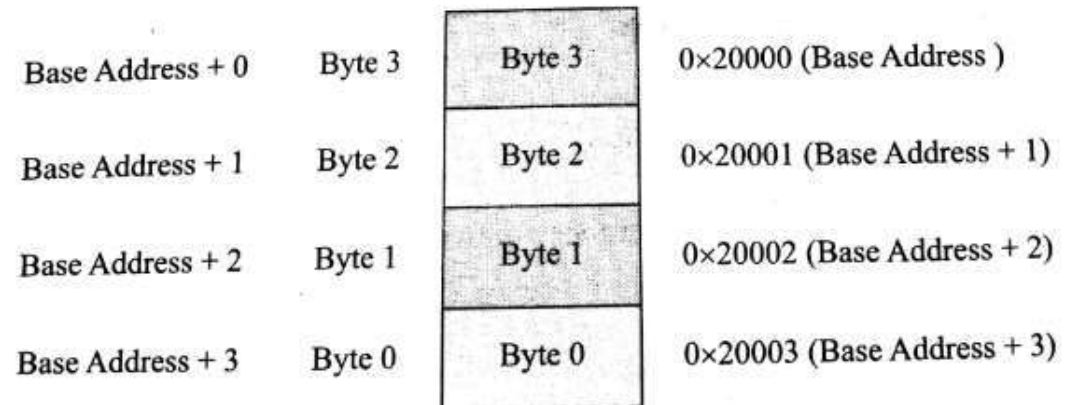
- Single shared bus for instruction and data fetching
- Low codes
- Low performance compared to harvard architecture
- Cheaper
- Allows self modifying codes
- Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory

- 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 multibyte system
 - Suppose the word length is 2 bytes then two ways in which data can be stored in the memory
 - Higher order of data byte at high memory and lower order of data byte at location just below the higher memory
 - Lower order of data and byte at the higher memory and higher order of data byte at the location just below the higher memory

□ Little Endian



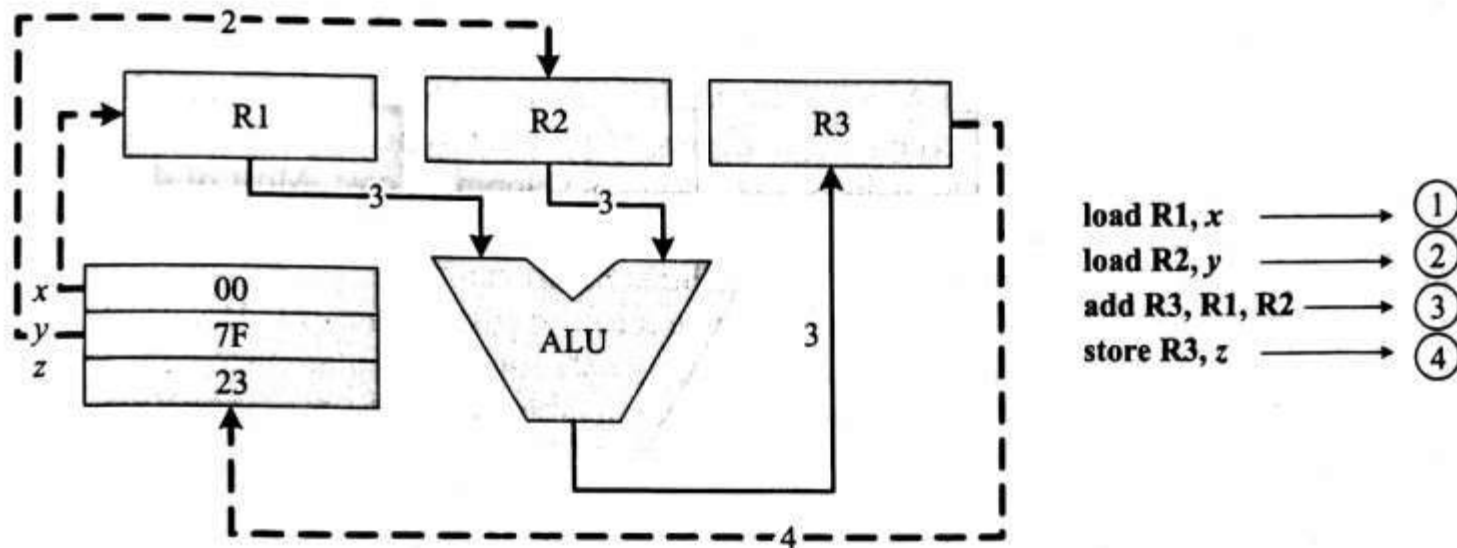
□ Big Endian



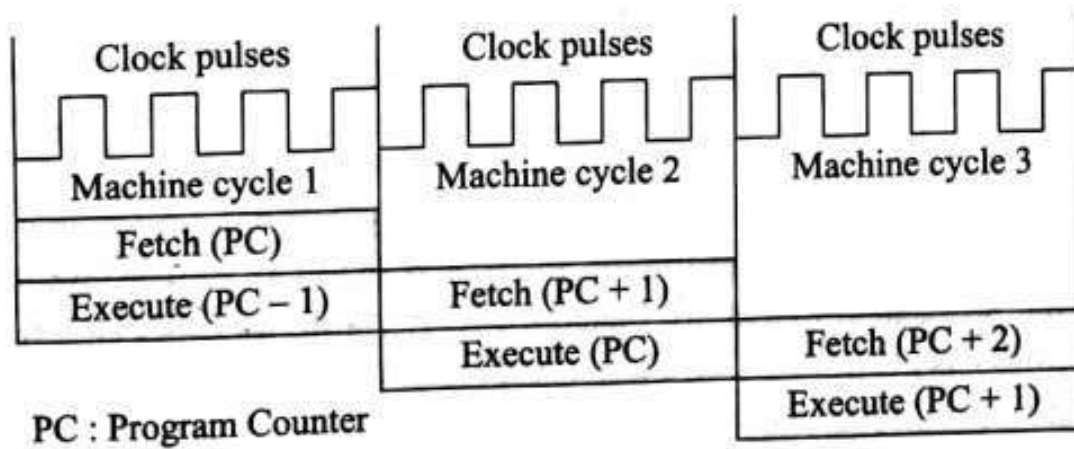
Load Store operations and Instruction Pipelining

20

- RISC processor instruction set is orthogonal
- Memory access related operations are performed by the special instructions load and store.



- The conventional instruction execution by the processor follows fetch-decode-execute sequence
- During decode operation the memory address bus is available
 - ▣ It can be possible to effectively utilize it for an instruction fetch.
 - Processing speed can be increased
 - ▣ Instruction pipelining refers to overlapped execution of instructions
 - It is meaningful to fetch the next instruction to execute, while decoding and execution of the current instruction is in progress
 - If the current instruction in progress is a program control flow transfer instruction, no point in fetching the next instruction



Application Specific Integrated Circuits

23

- ASIC is a microchip designed to perform a specific or unique application
- Integrates several functions into a single chip and thereby reduces the development cost
- Most of the ASICs are proprietary products
- As a single chip, ASIC consumes a very small area in the total system
 - ▣ Helps in the design of smaller systems with high capabilities/functionality

- ASICs can be pre-fabricated for a special application or it can be custom fabricated from a re-usable 'building block' library components
- Fabrication of ASIC requires a non-refundable initial investment for the process technology and configuration expenses (NRE)
- Since ASICs are proprietary products, developers may not be interested in revealing the internal details of it

Programmable Logic Devices

25

- Logic devices provide specific functions
 - ▣ Device-to-device interfacing
 - ▣ Data communication
 - ▣ Signal processing
 - ▣ Data display
 - ▣ Timing and control operations
- Logic devices can be classified into two broad categories
 - ▣ Fixed and programmable
- With PLDs designers use inexpensive software tools to quickly develop, simulate and test their designs.
 - ▣ E.g network router, A DSL modem, DVD player, automotive navigation system
- During the design phase the customer can change the circuitry as often they want

□ **CPLDs and FPGAs**

- Two major types of programmable logic devices are
 - FPGA and CPLD
- FPGAs offer the highest amount of logic density, most features and highest performance
- The largest FPGA now shipping
 - Xilinx virtex
 - Provides 8 million system gates
 - Advanced devices also offer features such as built-in hardwired processors
 - Substantial amounts of memory
 - Clock management systems
- FPGAs are used in wide variety of applications ranging from
 - Data processing and storage to instrumentation, telecommunications and digital signal processing

- ❑ CPLDs offer much smaller amounts of logic-up to about 10,000 gates
- ❑ CPLDs offer very predictable timing characteristics and are ideal from critical control applications
- ❑ Require low amounts of power and are very inexpensive
- ❑ Ideal for cost-sensitive, batter-operated, portable applications such as mobile phones and digital hand held assistants

Advantages of PLD

28

- Offer much more flexibility for customers
- Do not require long lead times for prototypes or production parts-
 - ▣ PLDS are already on a distributors shelf and ready for shipment
- Do not require customers to pay for large NRE costs and purchase expensive mask sets
- PLDs allows customers to order just the number of parts they need allowing them to control the inventory
- PLDs can be reprogrammed even after a piece of equipment is shipped to a customer.
- Advanced process technologies help PLDs in number of key areas
 - ▣ Faster performance, integration of more features, reduced power consumption and lower cost

Commercial off the shelf components(COTS)

29

- COTS product is one which is used 'as-is'
- Designed in such a way to provide
 - ▣ Easy integration
 - ▣ Interoperability with existing system components
- COTS component itself may be developed around a general purpose or domain specific processor or an ASIC or a PLD
- Examples
 - ▣ Remote controlled toy car control units
 - ▣ High performance, high frequency microwave electronics
 - ▣ High bandwidth analog to digital converters
- Major advantage is that
 - ▣ they are readily available in market
 - ▣ Cheap
 - ▣ Developer can cut down his/her development time to a great extent



- This network plug-in module gives the TCP/IP connectivity to the system being developed
- No need to design this module by ourself and write the firmware for TCP/IP protocol and data transfer
- Everything is ready supplied by the COTS manufacturer
- The major drawback of using COTS
 - Manufacturer may withdraw product or discontinue the production of COTS at any time if a rapid change in technology occurs
 - Adversely affects the commercial manufacturer of the embedded system

Memory

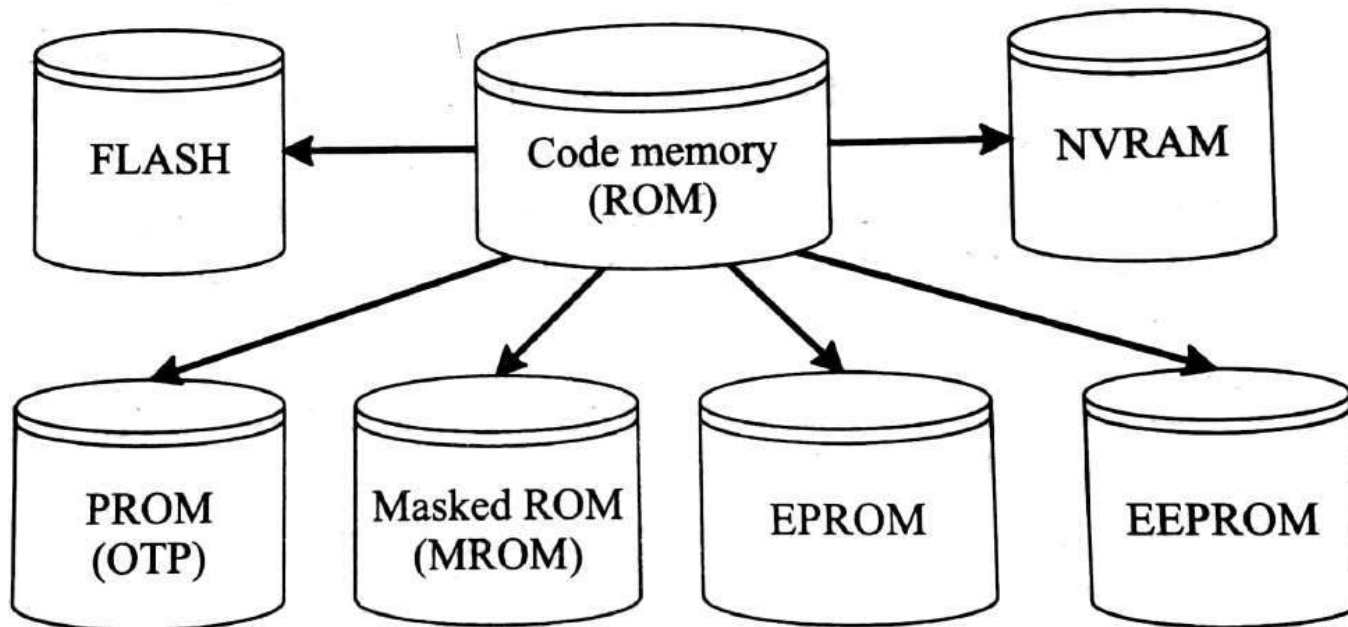
31

- Is an important part of a processor/ controller based embedded systems
- Some of the processors/ controllers contain built in memory referred as **on-chip memory**
 - ▣ Others do not contain memory inside the chip and requires external memory to be connected 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)

32

- Stores the program instructions



□ **Masked ROM**

- Is a one time programmable device
- Makes use of the hardwired technology for storing data
- Factory programmed by masking and metallization process at the time of production itself
 - According to the data provided by the end user
- Advantage is low cost for high volume production
- Different process used for the masking process of the ROM
 - 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
- Limitation is the inability to modify the device firmware against firmware upgrades

Programmable ROM/OTP

34

- One time programmable memory or PROM is not pre-programmed by the manufacturer
 - ▣ End user responsible for programming these devices
- This memory has a nichrome or polysilicon wires arranged in a matrix
 - ▣ Wires can be functionally viewed as fuses
 - ▣ Programmed by a PROM programmer which selectively burns the fuses according to the bit pattern to be stored.
 - ▣ Fuses blown/burned represent logic '0' else logic '1'
- OTP widely used for commercial production of embedded systems
- OTPs cannot be reprogrammed

□ EPROM

- OTPs not useful and worth for development purpose
- During development phase code is subject to continuous changes and using OTP each time to load the code is not economical
- EPROM gives the flexibility to reprogram the same chip
- EPROM stores the information by charging the floating gate of an FET
- EPROM contains a quartz crystal window for erasing the stored information
- Erasing the device is a tedious and time consuming process

□ 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
- The only limitation is their capacity when compared with standard ROM

□ FLASH

- Latest and most popular ROM technology
- FLASH is a variation of EEPROM technology
- Combines the re-programmability of EEPROM and the high capacity of standard ROMS
- Memory is organized as sectors(blocks) or pages
 - Erasing of memory can be done at sector level or page level without affecting other sectors or pages
- Each sector/page should be erased before re-programming

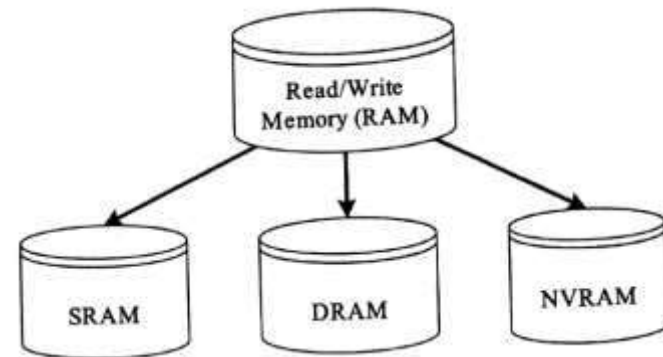
□ **NVRAM**

- Non-Volatile RAM is a RAM with battery backup
- Contains static RAM based memory and a minute battery for providing supply to the memory in the absence of external power supply
- Memory and battery are packed together in a single package
- Life span expected to be around 10 years

Read-Write Memory/ RAM

39

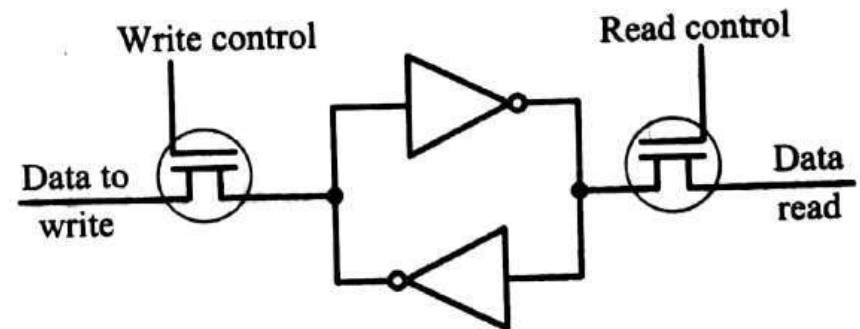
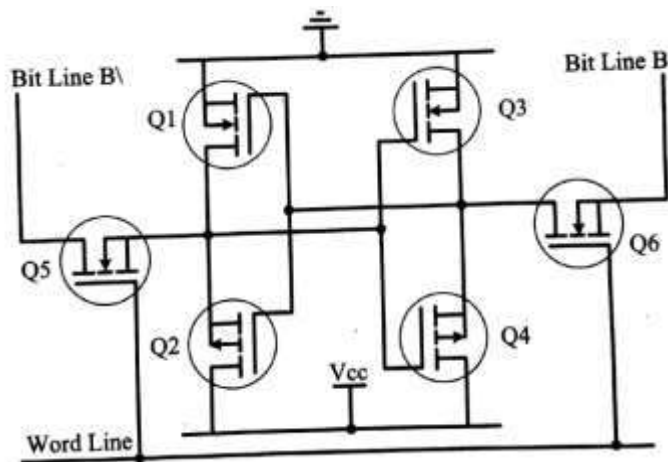
- RAM is the data memory or working memory of the controller/processor
 - ▣ Controller/processor can read from it and write to it
- RAM is
 - ▣ Volatile
 - ▣ Direct Access memory
- RAM falls into three categories
 - ▣ SRAM, DRAM, and non-volatile RAM



Static RAM

40

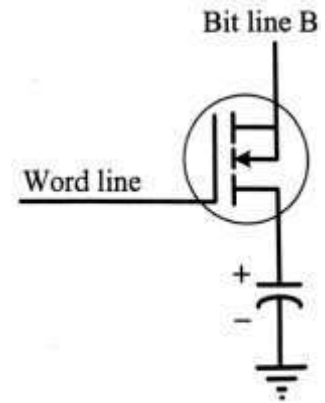
- ❑ Stores data in the form of voltage
- ❑ Made up of flipflops
- ❑ Is the fastest form of RAM available
- ❑ SRAM is fast in operation due to its resistive networking and switching capabilities



Dynamic RAM

41

- Stores data in the form of charge
- Made up of MOS transistors
- Advantages of DRAM
 - ▣ High density and low cost compared to SRAM
- Disadvantage
 - ▣ Information is stored as charge it gets leaked off with time
 - ▣ To prevent this they need to be refreshed periodically
 - Refresh operation is done periodically in milliseconds interval



SRAM Cell

- ❑ Made up of 6 CMOS transistors (MOSFET)
- ❑ Doesn't require refreshing
- ❑ Low capacity (Less dense)
- ❑ More expensive
- ❑ Fast in operation. Typical access time is 10ns

DRAM Cell

- ❑ Made up of a MOSFET and a capacitor
- ❑ Requires refreshing
- ❑ High Capacity (Highly dense)
- ❑ Less expensive
- ❑ Slow in operation due to refresh requirements. Typical access time is 60 ns. Write operation is faster than read operation

□ **NVRAM**

- Non-Volatile RAM is a random access memory with battery up
- Contains static RAM based memory and a minute battery for providing supply to the memory in the absence of external power supply.
- Memory and battery packed together in a single package

Memory according to the type of interface

- Interface of memory with processor/controller can be of various types
 - ▣ Parallel data lines
 - ▣ Serial interface like I2C
 - ▣ SPI (serial peripheral interface, 2+n line interface)
 - ▣ Can also be a single wire interconnection
- Serial interface is commonly used for data storage memory like EEPROM
- Memory density
 - ▣ serial memory is usually expressed in kilobits
 - ▣ Parallel interface memory is expressed in terms of kilobytes

Memory Shadowing

- Generally execution of a program or a configuration from ROM is very slow (120 to 200 ns) compared to the execution from RAM (40 to 70 ns)
- Shadowing is a technique used 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 (BIOS)
- In personal computer systems BIOS stores hardware configuration information
 - ▣ Usually BIOS is read and the system is configured according to it during boot-up and it is time consuming

- Manufacturers include a RAM behind the logical layer of BIOS at its same address as a shadow to the BIOS
 - ▣ 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

Memory Selection for Embedded Syst

47

- 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 which are modifiable by the application
- Memory requirement for an embedded system is solely dependent on
 - ▣ The type of embedded system
 - ▣ Applications for which it is designed

- Lot of factors need to be considered when selecting the type and size of the memory for embedded system
 - ▣ If ES is designed using SOC or a microcontroller with on-chip RAM and ROM
 - ▣ Thumb rule
 - Identify system requirement based on the type of the processor
 - Decide whether on-chip memory is sufficient or external memory is required.
- Example: A simple electronic Toy
- If ES is based on RTOS, RTOS requires
 - ▣ Certain amount of RAM for execution
 - ▣ ROM for storing the RTOS image
- Normally binary code for RTOS kernel containing all the services is stored in a non-volatile memory as either compressed or non-compressed data
 - ▣ During boot up the RTOS files are copied from storage memory, decompressed if required and loaded to the RAM for execution

- There are two parameters for representing a memory
 - ▣ Size of the memory chip
 - ▣ Word size of the memory
- **Size of the memory chip**
 - ▣ There's no option to get the memory chip with exact required number of bytes
 - ▣ Chips come in standard sizes
 - 512 bytes, 1Kbytes, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K, 1024K etc
- **Word size**
 - ▣ Refers to the number of bits that can be read/write at a time
 - ▣ 4,8,12,16,24,32 etc are the word sizes supported by memory chips
 - ▣ Word size should match with data bus width of the processor/controller

- **FLASH memory** is the popular choice for ROM in embedded applications
- Powerful and cost-effective solid state storage technology for mobile electronics devices and other consumer applications
- FLASH comes in two major variants
 - **NAND flash**
 - High density low cost non-volatile storage memory
 - **NOR flash**
 - Less dense and slightly expensive
 - Supports XIP
- Good practice to use a combination of NOR and NAND memory for storage requirements

- **EEPROM** data storage memory is available as either serial or parallel interface chip
- if the processor/controller of the device supports serial interface
 - ▣ Amount of data to write and read to and from the device is less
 - ▣ Better to have a serial EEPROM chip
 - ▣ Saves the address space of the total system
 - ▣ Memory capacity of serial EEPROM is expressed in bits or kilobits
- For embedded systems with low power requirements choose low power memory devices
- Certain ES may be targeted for operation at extreme environmental conditions
 - ▣ Select industrial grade memory chip in place of commercial grade chip

Sensors and Actuators

52

- Embedded system is in constant interaction with the real world
 - ▣ Controlling and monitoring
- Changes in the system environment or variables are detected by the sensors connected to the input port of the embedded system
- Embedded system may be designed for
 - ▣ Controlling purpose
 - Will produce some changes in the controlling variable to bring the controlled variable to the desired value
 - ▣ Monitoring purpose
 - No need to include an actuator

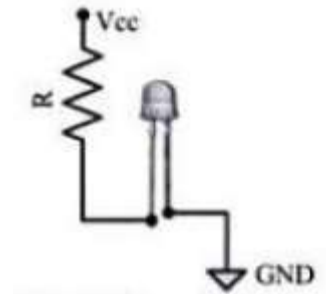
- A **sensor** is a transducer that converts energy from one form to another for any measurement or control purpose
- **Actuator** is a form of transducer device which converts signals to corresponding physical action.
 - Acts as an output device

I/O subsystem

54

- I/O subsystem facilitates the interaction of the embedded system with the external world.
- Interaction happens through sensors and actuators
- **Light Emitting Diode**

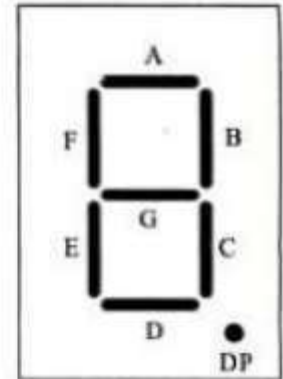
- Important output device for visual indication
- Used as an indicator for the status of various signals and situations
 - Device ON, Battery LOW, Charging of Battery
- LED is a p-n junction diode
- LED can be directly interfaced to the port pin of a processor/controller
 - Anode is directly connected
 - Cathode is connected

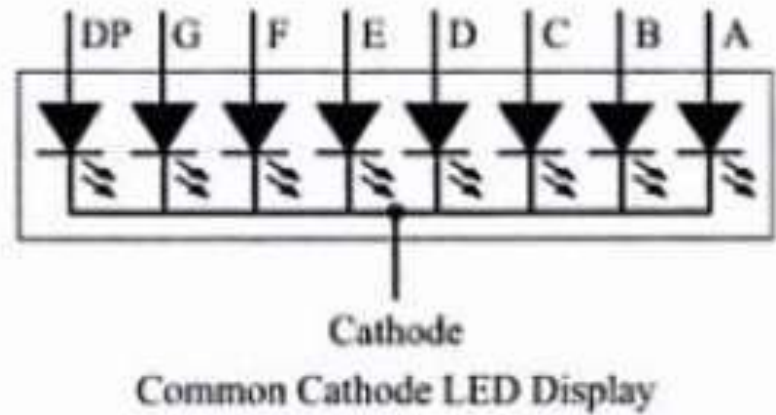
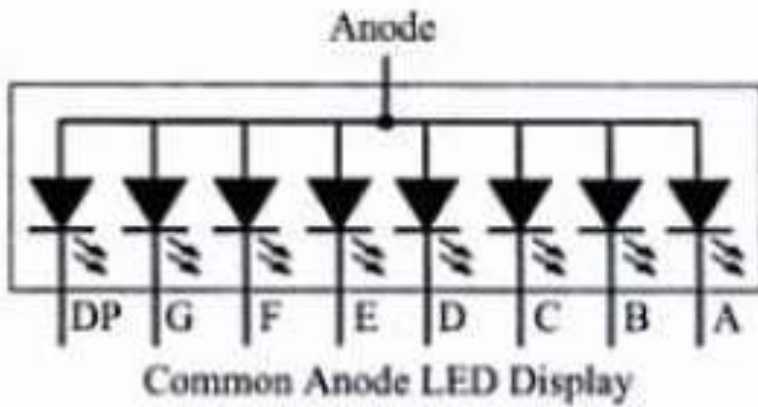


7 segment LED display

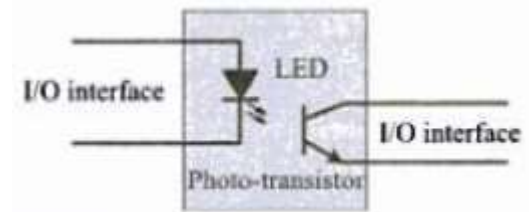
55

- Output device for displaying alphanumeric characters
- Contains 8 LED segments
 - ▣ 7- for alphanumeric display
 - ▣ 1- for representing decimal point
- All 8 LED segments need to be connected to one port of the processor/controller for displaying alphanumeric digits
- Two configurations for display
 - ▣ Common anode
 - ▣ Common cathode
- Current through each segment is limited by current limiting resistor
- Popular choice for low cost embedded applications



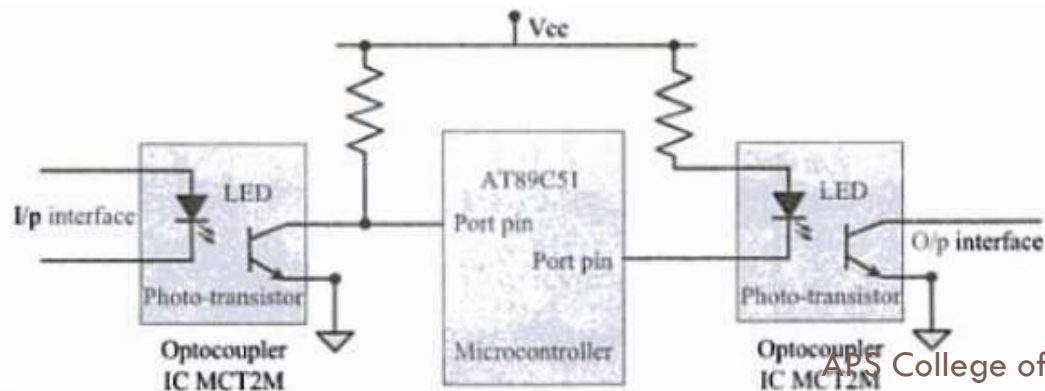


Optocoupler



57

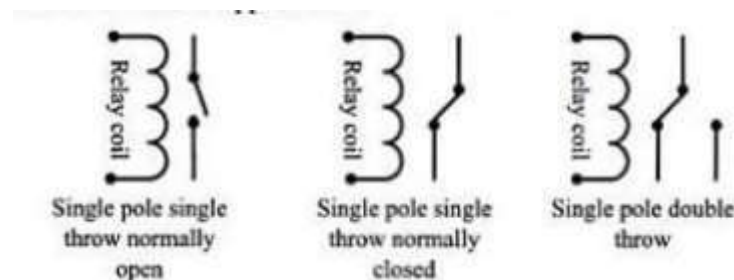
- Solid state device which isolates two parts of the circuit
- Combines an LED and a photo transistor in a single housing
- Used for suppressing interference in data communication, circuit isolation
- Can be used either in input circuits or in output circuits



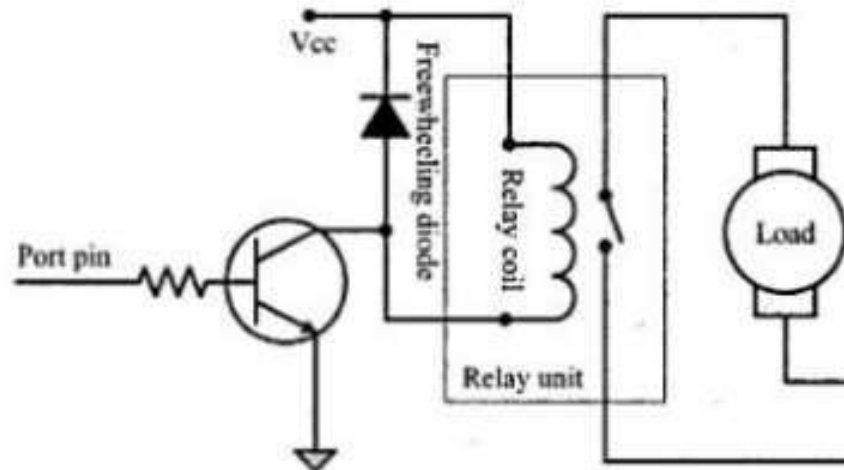
Relay

58

- It's a electromechanical device
- Acts as a dynamic path selectors for signals and power in an embedded application
- Contains a relay coil made up of insulated wire on a metal core and a metal armature with one or more contacts
- Works on electromagnetic principle
- Available in different configurations



- Single pole single throw
 - Has only one path for information flow
 - Path is either closed or open in normal condition
 - Relay is controlled using a relay driver circuit connected to the port pin of the processor/controller



Peizo Buzzer

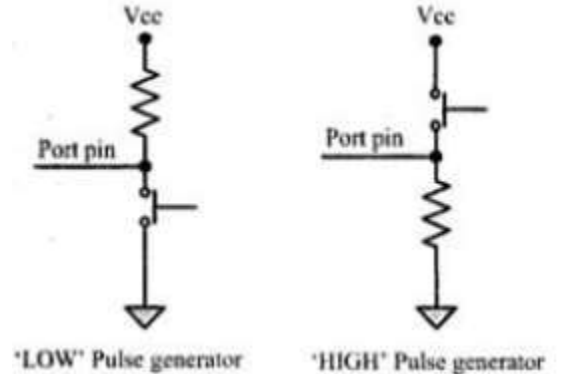
60

- Peizo electric device for generating audio indications
- Contains a piezoelectric diaphragm which produces audible sound in response to the voltage applied to it
- Available in two types
 - ▣ Self driving
 - Contains all the necessary components to generate sound at a predefined tone
 - ▣ External driving
 - Supports the generation of different tones
 - Tone can be varied by applying a variable pulse train
- Can be directly interfaced to the port pin of the processor/controller

Push button switch

61

- Its an input device
- Comes in two configurations
 - ▣ Push to make
 - Switch is normally in the open state
 - Makes a contact when it is pressed or pushed
 - ▣ Push to break
 - Switch is normally closed
 - Breaks the circuit contact when it is pushed or pressed
- Used for generating a momentary pulse
- Used as reset and start switch and pulse generator



Communication Interface

62

- Essential for communicating with various subsystems of the embedded system and with the external world
- Communication interface can be viewed in two different perspectives
 - ▣ Device/board level communication interface (onboard)
 - ▣ Product level communication interface (External)
- Embedded product is a combination of different types of components arranged on a PCB
- device/board level communication interface
 - ▣ Serial interfaces- I2C, SPI, UART, 1-wire
 - ▣ Parallel bus interface

Onboard Communication Interfaces

63

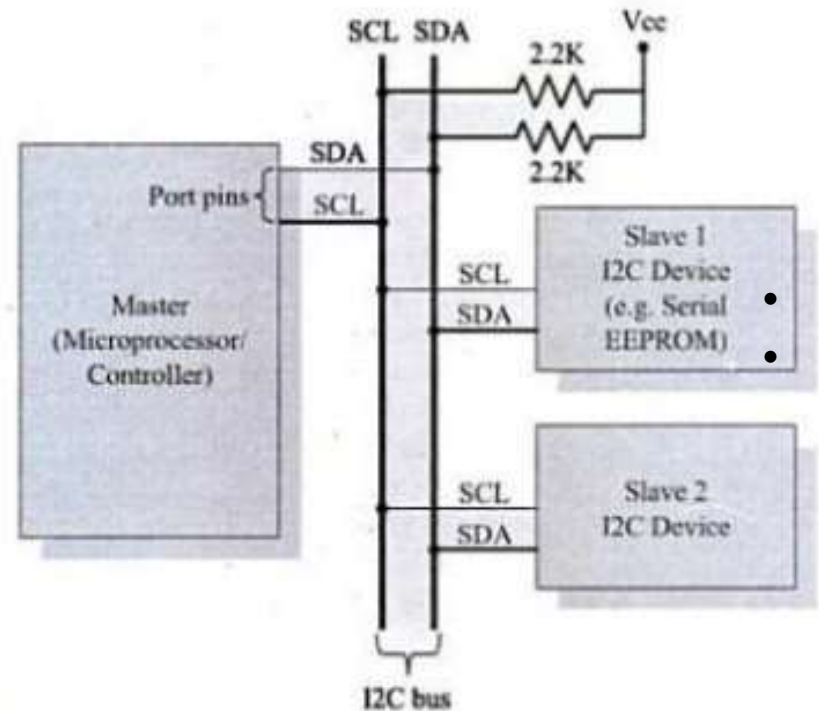
- Refers to the different communication channels/buses for interconnecting the various integrated circuits and other peripherals within the embedded system

Inter Integrated Circuit Bus (I2C)

- ▣ synchronous bi-directional half duplex two wire serial interface bus
- ▣ Intention was to provide an easy way of connection between a microprocessor/controller system and the peripheral chips in television sets
- ▣ Comprises of two bus lines
 - SCL
 - SDA

- Devices connected to the I2C bus can act as either 'Master' device or 'Slave' device.
- Role of a Master
 - ▣ responsible for controlling the communication by initiating/terminating data
 - ▣ Sending data and generating necessary synchronization clock pulses
- Role of a Slave
 - ▣ Wait for the master and respond upon receiving the commands
- Master and slave devices can act as either transmitter or receiver
- I2C supports multi masters on the same bus

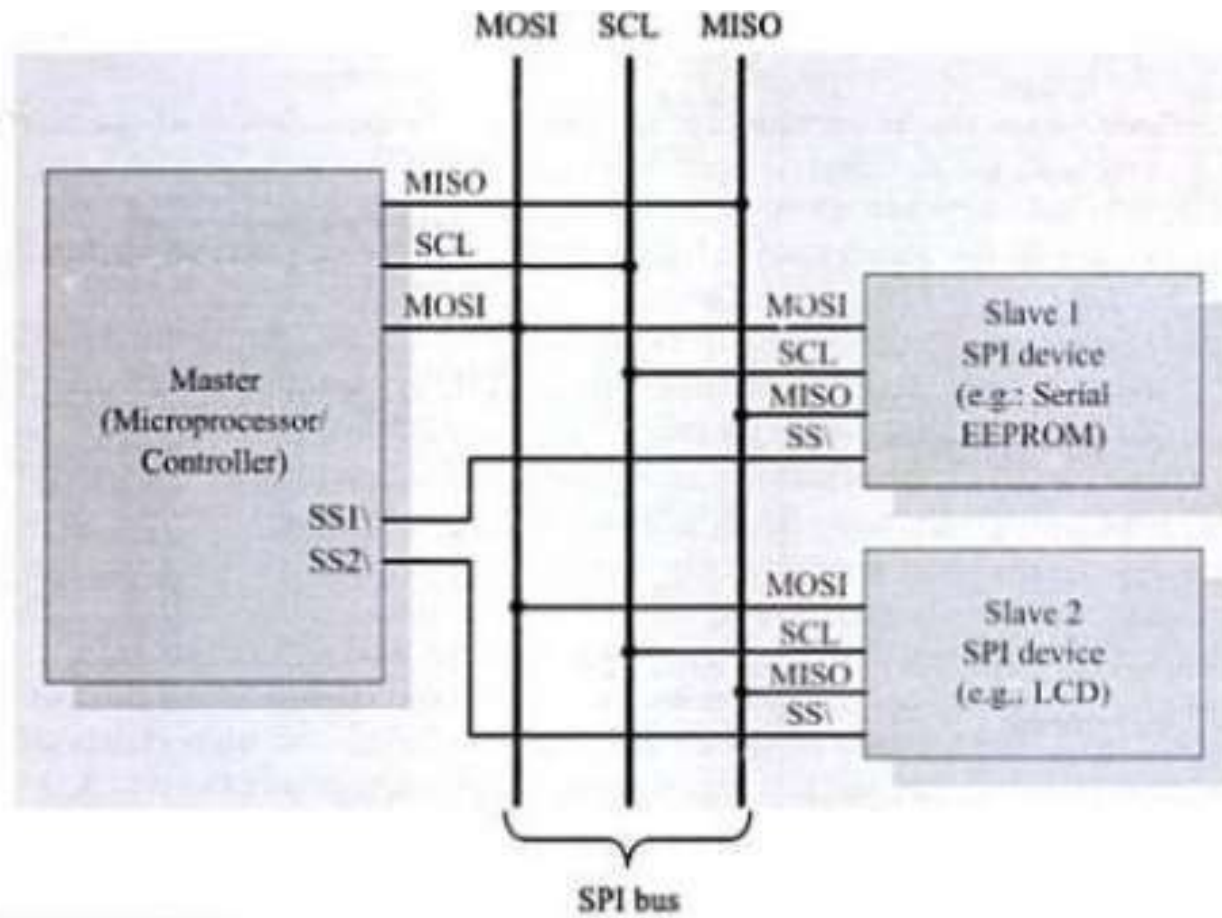
- I2C bus interface is built around an input buffer and an open drain or collector transistor
- When bus is in idle state
 - ▣ Open drain/ collector transistor will be in the floating state
 - ▣ SDA and SCL line switch to the 'high impedance' state
- Address is assigned by hardwiring the address lines of the device to the desired logic level



SPI(Serial Peripheral Interface) Bus

66

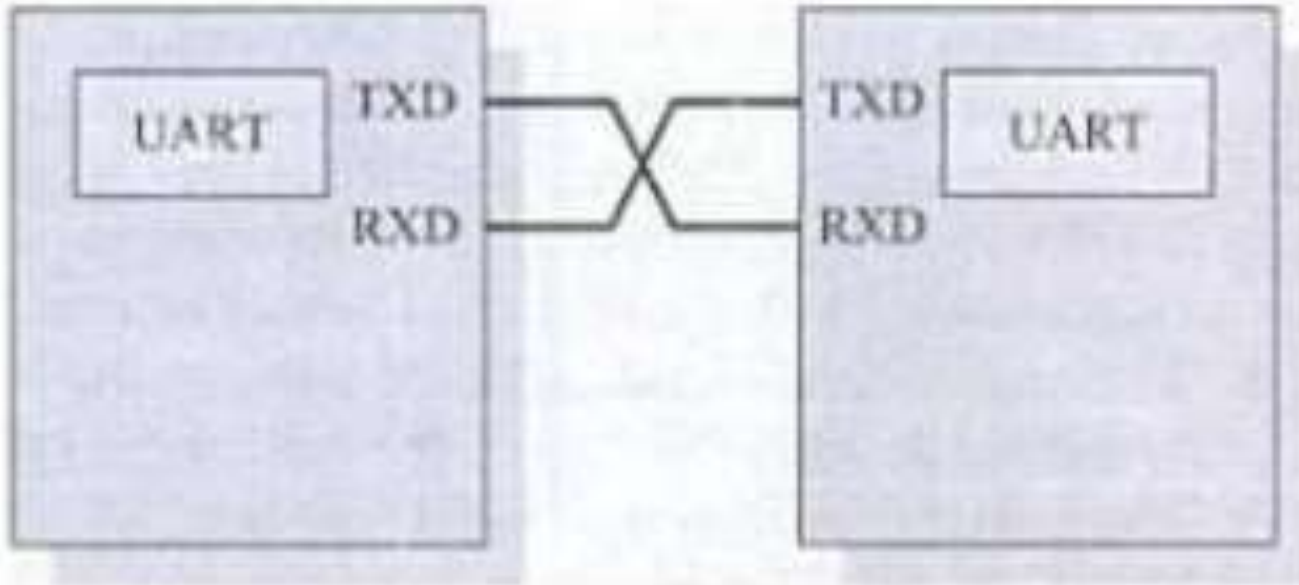
- Synchronous bidirectional full duplex four-wire serial interface bus
- SPI is a single master multi-slave system
 - ▣ Possible to have a system with more than one master
 - ▣ Condition that only one master device is active at any given point of time
- Requires four signals for communication
 - ▣ MOSI – Master out Slave In
 - ▣ MISO - Master In Slave Out
 - ▣ SCLK – Serial Clock
 - ▣ SS – Slave Select
- Most suitable for applications which require transfer of data in streams
- Doesn't support an acknowledgement mechanism



UART

68

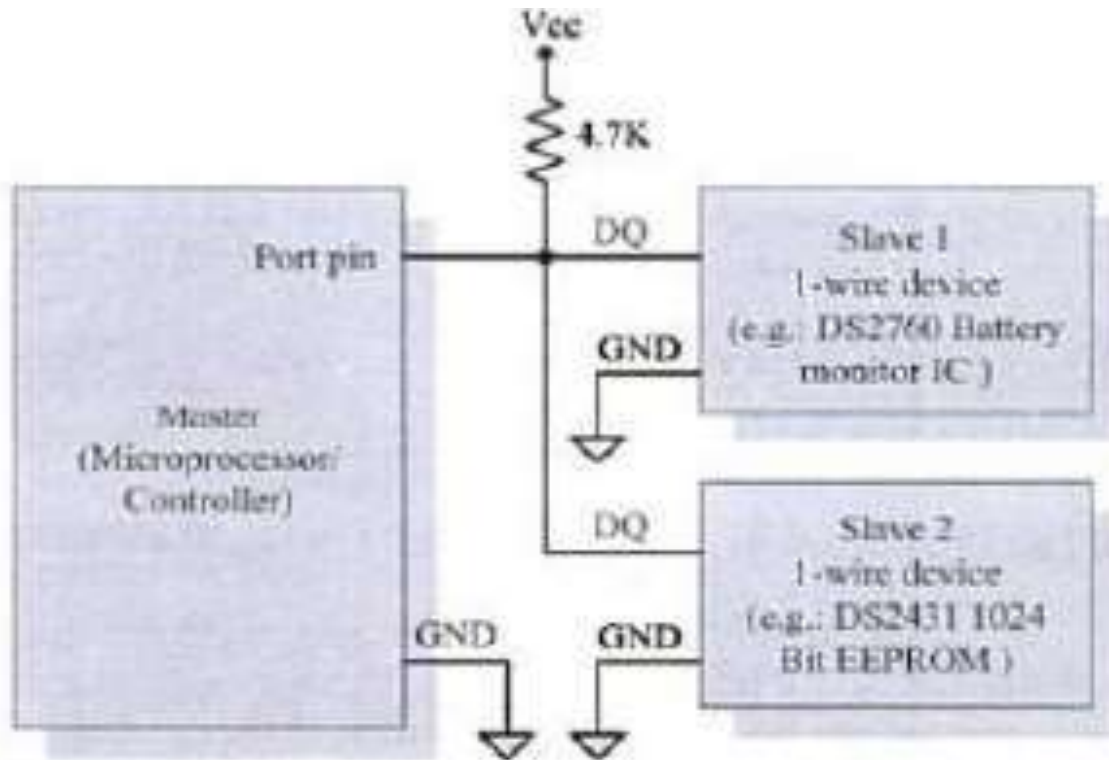
- Asynchronous form of serial data transmission
- Doesn't require a clock signal to synchronize transmitting end and receiving end for transmission
 - ▣ Relies on the pre-defined agreement between the transmitting and receiving device
 - Baud rate
 - Number of bits per byte
 - Parity
 - Number of start bits and stop bit and flow control
- Start and Stop of communication is indicated through special bits in the data stream



1-wire Interface

70

- Asynchronous half duplex communication protocol
 - ▣ Developed by maxim dallas semiconductor
- Makes use of a single signal line (wire) called DW for communication
 - ▣ Follows master-slave communication model
- Allows power to be sent along the signal wire as well
- Supports single master and one or more slave devices on the bus





EMBEDDED SYSTEMS DESIGN CONCEPT

Module 4

1

CHARACTERISTICS OF AN ES

1. Application and Domain specific

- Each embedded system is having certain functions to perform and are designed to do the intended functions only
- Cannot replace ES designed for a particular domain with another

2. Reactive and Real Time

- ES are in constant interaction with real world through sensors and defined input devices
- Control algorithm reacts in a designed manner
- Event may be a periodic or an unpredicted one
 - Unpredicted events are captured by scheduling the systems
- Real time systems, timing behavior is deterministic

3. Operates In harsh environments

- ES could be installed in a dusty or a high temperature zone, subject to vibrations and shock

4. **Distributed**

- ES may be a part of larger system
- Many such distributed ES form a single large ECU
- E.g. Automatic vending machine, Automatic Teller Machine (ATM)

5. **Small size and weight**

- Product aesthetics is an important factor in choosing a product
- People believe in phrase “ Small is beautiful”

6. **Power concerns**

- Power management is an important factor in designing ES.
- ES should be designed to minimize the heat dissipation by the system
- Ultra low power components are available in the market

QUALITY ATTRIBUTES OF ES

- Non functional requirements that need to be documented properly
- Broadly classified into two
 - Operational QA & Non-operational QA

Operation QA

1. Response

- Measure of quickness of the system
- Gives an idea on how fast the system is tracking the changes in input variables
- E.g ES deployed in flight control system




2. Throughput

- Deals with efficiency of the system
- Rate of production or operation of a defined process over a stated period of time
- Generally measured in terms of 'Benchmark'

3. Reliability

- Measure of how much % you can rely upon the proper functioning of the system.
- % susceptibility of the system to failures
- Mean time Between Failure (MTBF) and Mean Time to Repair (MTTR) are the terms used in defining system reliability
- MTBF
 - Gives the frequency of failures in hours/weeks/months
- MTTR
 - Specifies how long the system is allowed to be out of order following a failure
 - It should be in terms of minutes in case of critical application need

4. Maintainability

- Deals with support and maintenance to the end user or client
 - in case of technical issues and product failures or on the basis of routine system check up
- A more reliable system means a system with less corrective maintainability requirements
- As the reliability of the system  the chances of failure and non-functioning also  and thereby need for maintainability is also 
- Maintainability is broadly classified into two categories
 - Scheduled or periodic maintenance (Preventive Maintenance)
 - E,g Replacement of cartridge in a printer
 - Maintenance to unexpected failures (Corrective Maintenance)
 - E.g Paper feeding of the printer fails
- Ideal value for availability is expressed as

$$A_i = \text{MTBF}/(\text{MTBF}+\text{MTTR})$$

4. Security

- Confidentiality, Integrity and availability are the three major measures of information security
- A good example of the security aspect in an embedded product is a PDA
 - PDA can either be a shared resource or an individual one
 - If its shared, then there should be some mechanism in the form of user name and password to access into a particular persons profile

5. Safety

- Safety and Security are two confusing terms
- They represent two unique aspects in quality attributes
- Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products
- Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action

NON OPERATIONAL QUALITY ATTRIBUTES

- The quality attributes that needs to be addressed for the product 'not' on the basis of operational aspects
 - Testability and Debug-ability
 - Evolvability
 - Portability
 - Time to prototype and market
 - Per unit and total cost
- **Testability and Debug-ability**
 - Deals with how easily one can test his/her design application and by which means he/she can test it
 - Testability is applicable to both the embedded hardware and firmware
 - Debug-ability is a means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system

○ **Evolvability**

- It's a term closely related to Biology
- Evolvability is referred as the non-heritable variation
- For an embedded system evolvability refers to the ease with which the embedded product can be modified to take advantage of new firmware or hardware technologies

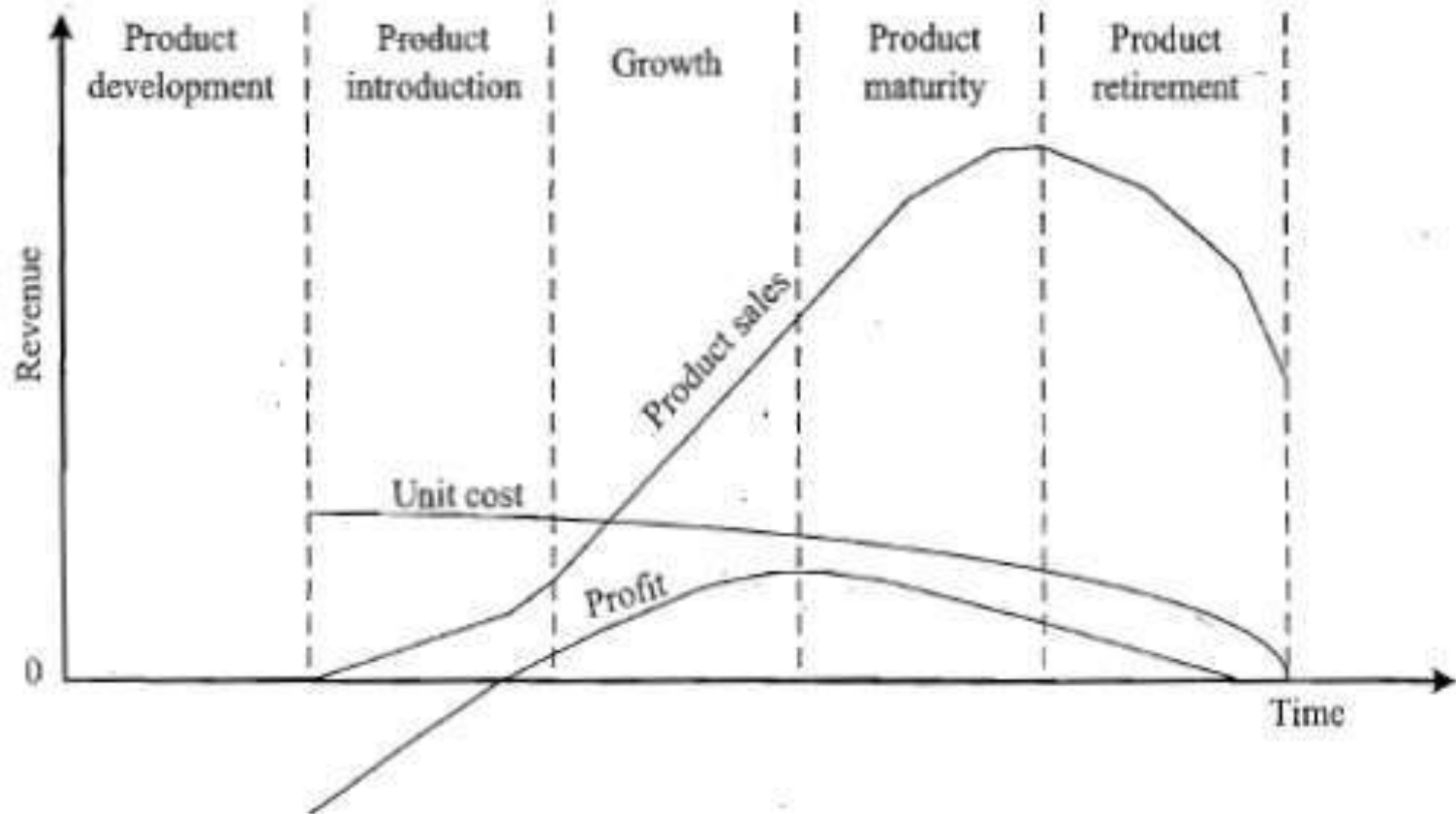
○ **Portability**

- It's a measure of system independence
- An embedded product is said to be portable if its capable of functioning as such in various environments, target processor/controllers and embedded operating systems
- A standard embedded product should always be flexible and portable
- Portability with respect to migration of embedded firmware written for one target processor to a different target processor
- Program written in high level language vs assembly level language

○ Time to prototype and market

- It is the time elapsed between the conceptualization of product and the time at which the product is ready for selling or use.
- The commercial embedded product market is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product
 - There may be multiple players in the industry who develop products of the same category e.g mobile phone
- If you come up with the design and it takes long time to develop and market it
 - The competitor product may take advantage of it with their product
- Embedded technology is one where rapid technology change is happening
 - Start designing using new technology and it takes longtime to develop and market the product.
 - By the time product reaches the market, the technology might have superseded with new technology

- Product prototyping helps in reducing time-to-market
- Time to prototype is also another critical factor
 - If prototype developed faster the development time can be brought down significantly
 - In order to shorten the time to prototype, make use of options such as off-the –shelf components, reusable assets etc
- 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
 - From a designer/product development company perspective the ultimate aim of a product is to generate marginal profit.



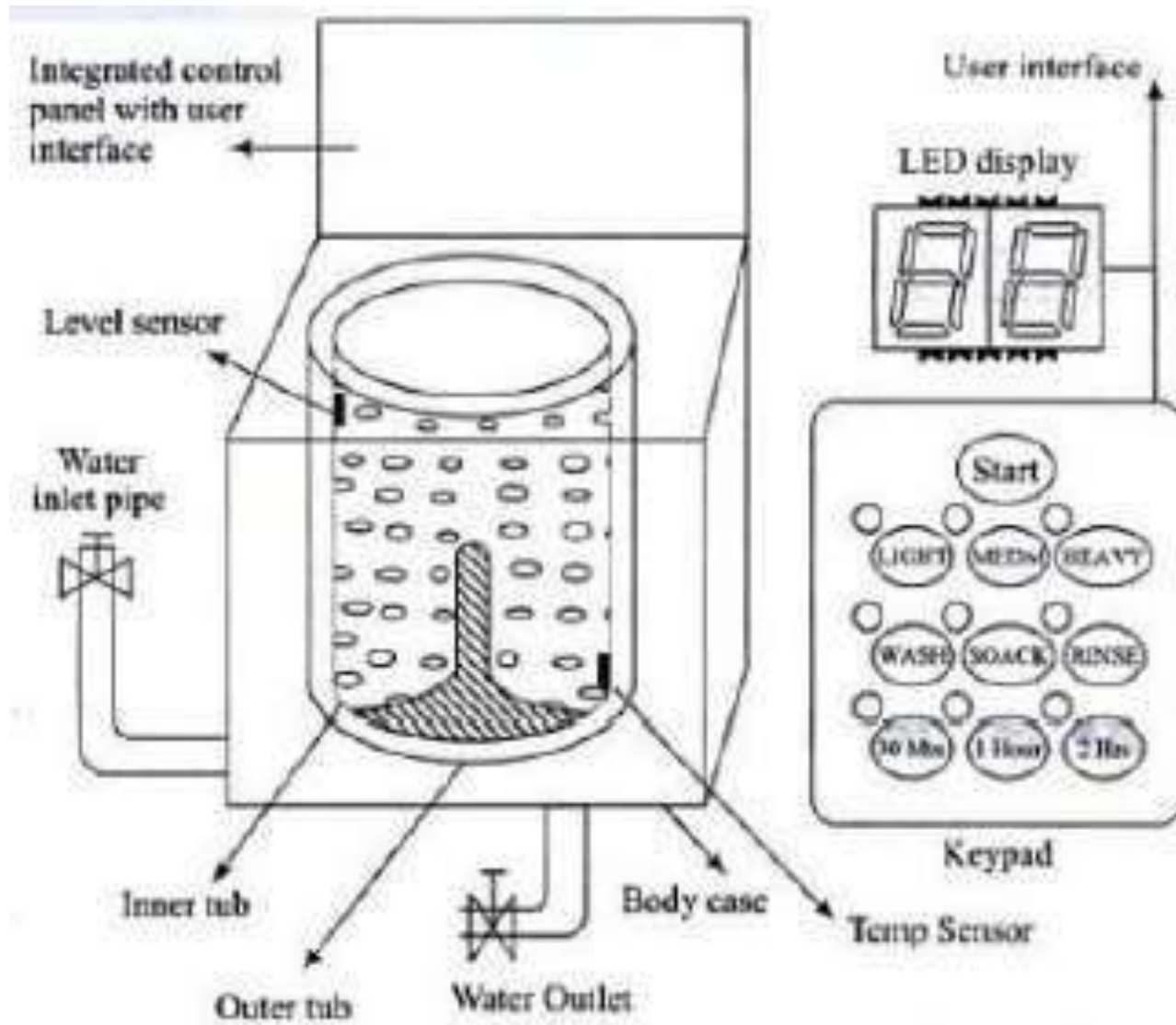
EMBEDDED SYSTEM-APPLICATION AND DOMAIN SPECIFI

- Embedded Systems are highly specialized in functioning and are dedicated for a specific application
- It is therefore not possible to replace an embedded system developed for a specific application in a specific domain with another embedded sys designed for some other application in some other domain
- People experience the power of embedded systems and enjoy the features and comfort provided by them
- They are totally unaware or ignorant of the intelligent embedded players working behind the products providing enhanced features and comfort

WASHING MACHINE- APPLICATION SPECIFIC EMBEDDED SYSTEM

- Components of washing machine in which some are visible and some are invisible
 - Actuator
 - Motorized agitator
 - Tumble tub
 - Water drawing pump
 - Inlet valve to control the flow of water into the unit
 - Sensor
 - Water temperature sensor
 - Level sensor
 - Control
 - Microprocessor/controller based board with interfaces to sensors and actuators
 - Provides connectivity to user interfaces like
 - Keypad for setting the washing time, type of material to be washed
 - User feedback is reflected through
 - LED's and display unit connected to the control board





AUTOMOTIVE-DOMAIN SPECIFIC EXAMPLES OF EMBEDDED SYSTEM

- The major application domains of embedded systems are consumer, industrial, automotive, telecom etc..
- Telecom and automotive holds a big market share for embedded systems

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 vehical varies from simple mirror and wiper controls to complex air bag controller and anitlock brake system(ABS)
- Automotive embedded systems are normally built around microcontrollers or DSPs or a hybrid of the two and are generally known as **ECUs**

- The number of embedded controllers in an ordinary vehicle varies from 20 to 40
 - Mercedes and BMW may contain 75 to 100 ECU
- Government regulations on fuel economy, environmental factors, emission standards and increasing demands on safety, comfort and infotainment forces automotive manufactures to opt for sophisticated ECUs



- ECUs used in the automotive industry can be broadly classified into
 - High Speed embedded control units
 - Low Speed embedded control units
- High Speed Electronic Control Units
 - Deployed in critical control units requiring fast response
 - It includes
 - Fuel injection Systems
 - Antilock brake Systems
 - Engine control
 - Electronic Throttle
 - Steering Controls
 - Transmission control
 - Central Control Unit
- Low Speed ECUs
 - Deployed in applications where response time is not so critical
 - Built around low cost microprocessor/microcontrollers and DSPs
 - Audio controllers, passenger and driver door locks, door glass controls, wiper control, mirror control, seat control systems, head lamp, tail lamp controls, sun roof control unit etc

- Automotive Communication Buses
 - Makes use of simple serial buses for communication
 - Greatly reduces the amount of wiring inside a vehicle.
- Different types of serial interface buses deployed in automotive embedded applications

1. CAN (Controller Area Network)

- Originally proposed by Robert Bosch
- Supports medium speed (ISO11519-Class B with data rates upto 125 Kbps) and High speed(ISO11898 Class C with data rates upto 1 Mbps) data transfer
- CAN is an event driven protocol interface with support for error handling in data transmission
- Generally employed in safety system like
 - Airbag control, power train systems like engine control and ABS and navigation system like GPS

2. LIN (Local Interconnect Network)

- LIN bus is a single master multiple slave communication interface
- LIN is a low speed, single wire communication interface with support for data rates upto 20Kbps 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 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

○ MOST (Media-Oriented System Transport) Bus

- Targeted for automotive audio/video equipment interfacing, used primarily in European cars
- A MOST bus is a multimedia fibre-optic point-to-point network implemented in
 - Star, Ring or daisy chained topology over OFC
- Then MOST bus-specifications define Physical layer as well as application layer, network layer and media access control.
- MOST bus is an optical fiber cable connected between Electrical Optical Converter (EOC) and Optical Electrical Converter (OEC) which would translate into the optical cable MOST bus

Key Players of the Automotive Embedded Market

- The key players of the automotive embedded market can be visualized in three verticals
 - Silicon Providers
 - Solution Providers
 - Tools and platform Providers
- Silicon Providers
 - Responsible for providing necessary chips which are used in the control application development
 - Chip may be a standard product like microcontroller or DSP or ADC/DAC chips
 - Some applications require specific chips and they are manufactured as ASIC.

- The leading silicon providers in the automotive industry are
 - Analog Devices
 - Provider of world class digital signal processing chips
 - Precision analog microcontrollers
 - Programmable inclinometer/accelerometer
 - LED drivers
 - For automotive signal processing applications, driver assistance systems, audio system, GPS/Navigation system
 - Xilinx
 - Supplier of high performance FPGAs, CPLDs and automotive specific IP cores for
 - GPS navigation systems
 - Driver Information Systems
 - Distance Control
 - Collision Avoidance
 - Rear Seat entertainment
 - Adaptive Cruise Control
 - Voice Recognition etc

- **Atmel**
 - Supplier of cost effective high density flash controllers and memories
 - Provides a series of high performance microcontrollers
 - A wide range of ASSP (Application Specific Standard Products)
 - For chasis, body electronics, security, safety and car infotainment and automotive networking products for CAN, LIN and Flex Ray
- **Maxim/Dallas**
 - Supplier of world class analog, digital and mixed signal products R F components for all kinds of automotive solutions
- **NXP Semiconductor**
 - Flash microcontrollers
- **Renesas**
 - Provider of high speed microcontrollers and LSI technology for car navigation systems accommodating three transfer speeds
 - High, Medium and Low
- **Texas Instruments**
 - Supplier of microcontrollers, digital signal processors and automotive communication control chips for LIN bus protocol

- **Fujitsu**
 - Supplier of finger print sensors for security applications, graphic display controller for instrumentation application
 - AGPS/GPS for vehicle navigation system and different types of microcontrollers for automotive control applications
- **Infineon**
 - Supplier of high performance microcontrollers and customized application specific chips
- **NEC**
 - Provider of high performance microcontrollers
- Tools and Platform providers

HARDWARE SOFTWARE CO-DESIGN AND PROGRAM MODELING

- In traditional embedded system development approach, the hardware software partitioning is done at an early stage
 - Software engineers take care of the software architecture development and implementation
 - Hardware engineers responsible for building the hardware required for the product.
 - Less interaction between teams and development happens either serially or in parallel
 - Integration is the next step
- Need for novel approach for embedded system design in order to reduce the 'time-to-market'

○ During co-design process

- The product requirements are captured from the customer are converted into **system level** needs or processing requirements
- At this point it is not segregated as either hardware or software requirement
 - Instead specified as functional requirement
- The system level processing requirements are then transferred into **functions** which can be simulated and verified against performance and functionality
- **Architecture** design follows the system design
 - Partitioning takes place here
 - System level requirements are mapped into hardware and/or software
 - Partitioning performed based on the hardware-software trade-offs

FUNDAMENTALS ISSUES IN HARDWARE SOFTWARE CO-DESIGN

- The hardware software co-design is a problem statement
 - When tried to solve this statement in real life we may come across multiple issues in the design
- Fundamental issues in hardware software co-design
 1. Selecting the model
 - **Models** are used for capturing and describing the system characteristics
 - Model is a formal system consisting of objects and composition rules
 - Hard to make design as to which model should be followed in a particular system design
 - Designers switch between a variety of models from the requirements specification to the implementation aspect
 - Objective varies with each face

2. Selecting Architecture

- A model only captures the system characteristics and does not provide information on ‘how the system can be manufactured?’
- Architecture specifies how a system is going to implement in terms of the number and types of different components and the interconnection among them
- Controller Architecture, Datapath Architecture, CISC, RISC, Very long Instruction word computing (VLIW)
- Single Instruction Multiple Data (SIMD), Multiple instruction Multiple Data (MIMD) etc are the commonly used architecture in system design
- Some of them fall into application specific architecture class while others fall into either general purpose architecture class or parallel processing class

○ 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

○ Datapath Architecture

- Best suited for implementing the data flow graph model where 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
- In datapath architecture the datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units.
- Ports connect the datapath to multiple buses

○ Finite State Machine Datapath (FSMD)

- 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
 - The second I/O port interfaces the datapath with external world for data input and data output
- Normally 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

○ The complex instruction set computing (CISC)

- This architecture uses an instruction set representing complex operations
- It is possible for a CISC instruction set to perform a large complex operation with a single instruction
- The use of a single complex instruction in place of multiple simple instructions greatly reduces the program memory access and program memory size requirement
 - It requires additional silicon for implementing microcode decoder for decoding the CISC instruction
- The datapath for the CISC processor is complex
- RISC architecture uses instruction set representing simple operations and 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 supports extensive pipelining

- **The Very Long Instruction Word (VLIW)**
 - Architecture implements multiple functional units in the datapath
 - The VLIW instruction packages one standard instruction per functional unit of the datapath
- **Parallel Processing Architecture**
 - Implements multiple concurrent processing elements and each processing element may associate a datapath containing register and local memory
 - Single Instruction Multiple Data (SIMD) and Multiple instruction multiple data (MIMD) are examples of parallel processing architecture
 - SIMD
 - 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
 - SIMD architecture forms the basis of reconfigurable processor

○ MIMD

- The MIMD architecture forms the basis of multiprocessor systems.
- The PEs in a multiprocessor system communicates through mechanisms like shared memory and message passing

Select 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
 - Languages like VHDL, System C, Verilog etc for hardware implementations
- A single model can be used for capturing a variety of models
- Certain models are good in capturing certain computational models
- The only pre-requisite is selecting a programming language for capturing a model is that the language should capture the model easily

Partitioning System Requirements into hardware and software

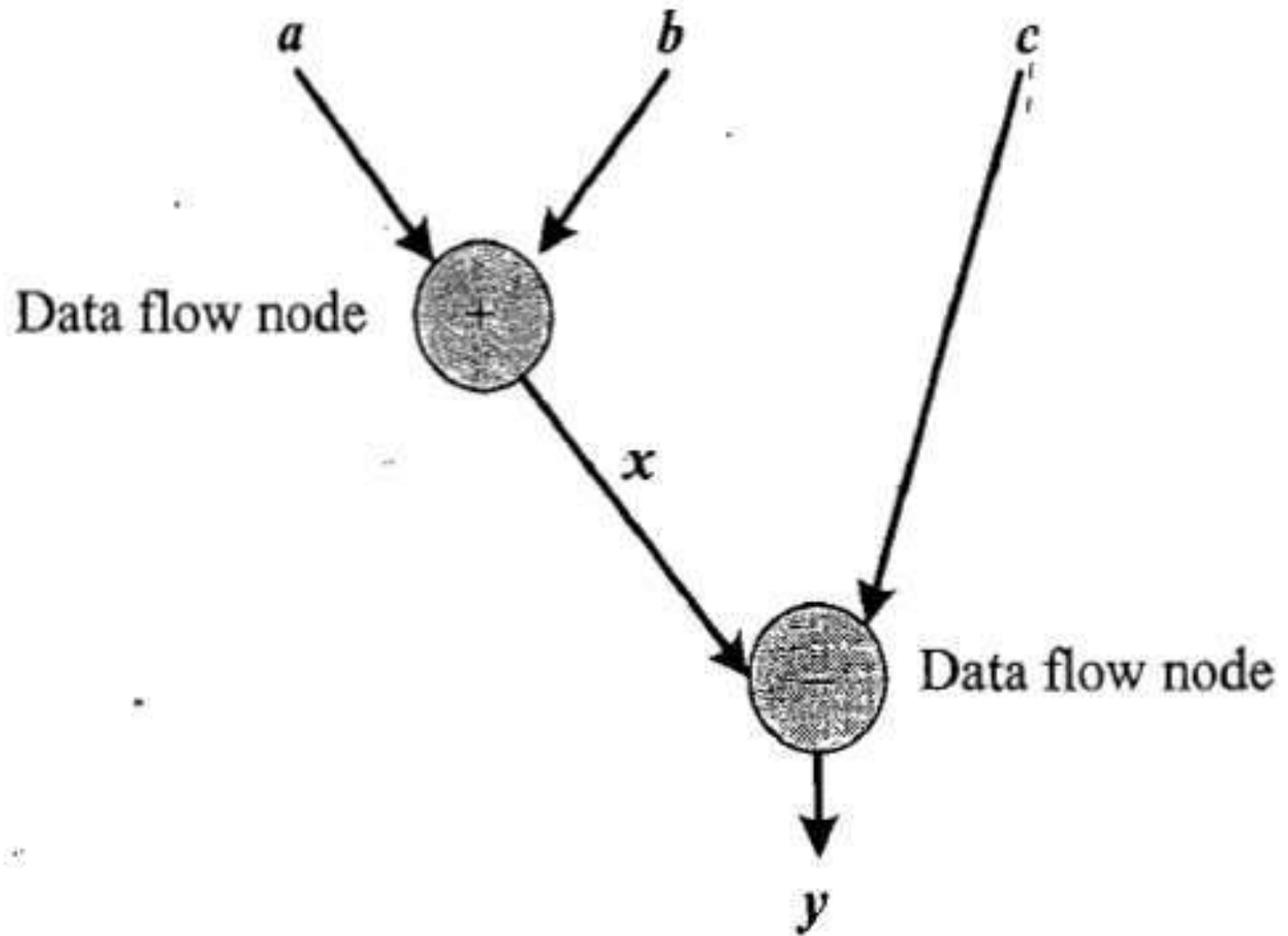
- From the implementation perspective, it may be possible to implement the system requirements in either hardware or software.
- It's a tough decision making task to figure out which one to opt.
- Various hardware-software trade-offs are used for making decision on the hardware-software partitioning.

COMPUTATIONAL MODELS IN EMBEDDED DESIGN

- Some of the commonly used computational models
 - Data Flow Graph (DFG) model
 - State machine model
 - Concurrent process model
 - Sequential program model
 - Object oriented model

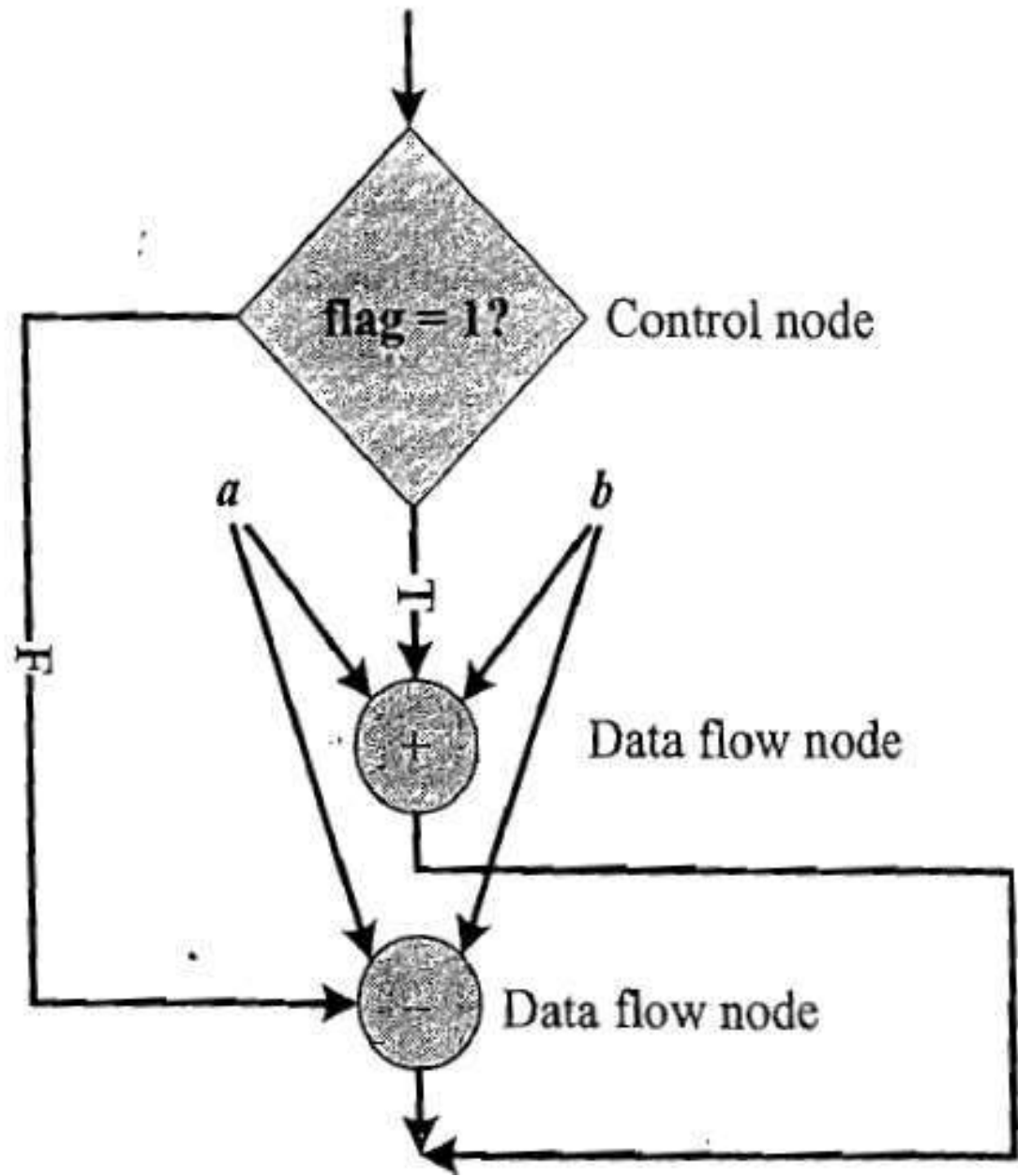
Data Flow Graph Model

- DFG model translates the data processing requirements into a data flow graph
- DFG graph is a data driven model in which program execution is determined by data
- This model emphasizes on the data and operation on the data which transforms the input data to output data
- 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 represents input data and an outward arrow represents output data
- **Embedded applications which are computational intensive and data driven are modeled using DFG**
- DSP applications are typical examples for it



Control Data Flow Graph/Diagram (CDFG)

- It seems that DFG model is a data driven model in which execution is controlled by data and it doesn't involve any control operations (Conditional)
- The CDFG model is used for modeling applications involving conditional program execution.
 - CDFG models contain both data operations and control operations
- The CDFG uses data flow graph (DFG) as elements and conditional constructs as decision makers
- CDFG contains both data flow nodes and decision nodes



- **Real world example** modeling the embedded application using CDFG is the capturing and saving of the image to a format set by the user in a digital still camera where everything is data driven starting from the Analog front end which converts the CCD sensor generated analog signal to digital signal and the task which stores the data from ADC to a frame buffer for the use of a media processor which performs various operations like auto correction, while balance adjusting etc.
- The decision on in which format the image is stored is controlled by the camera setting configured by the user

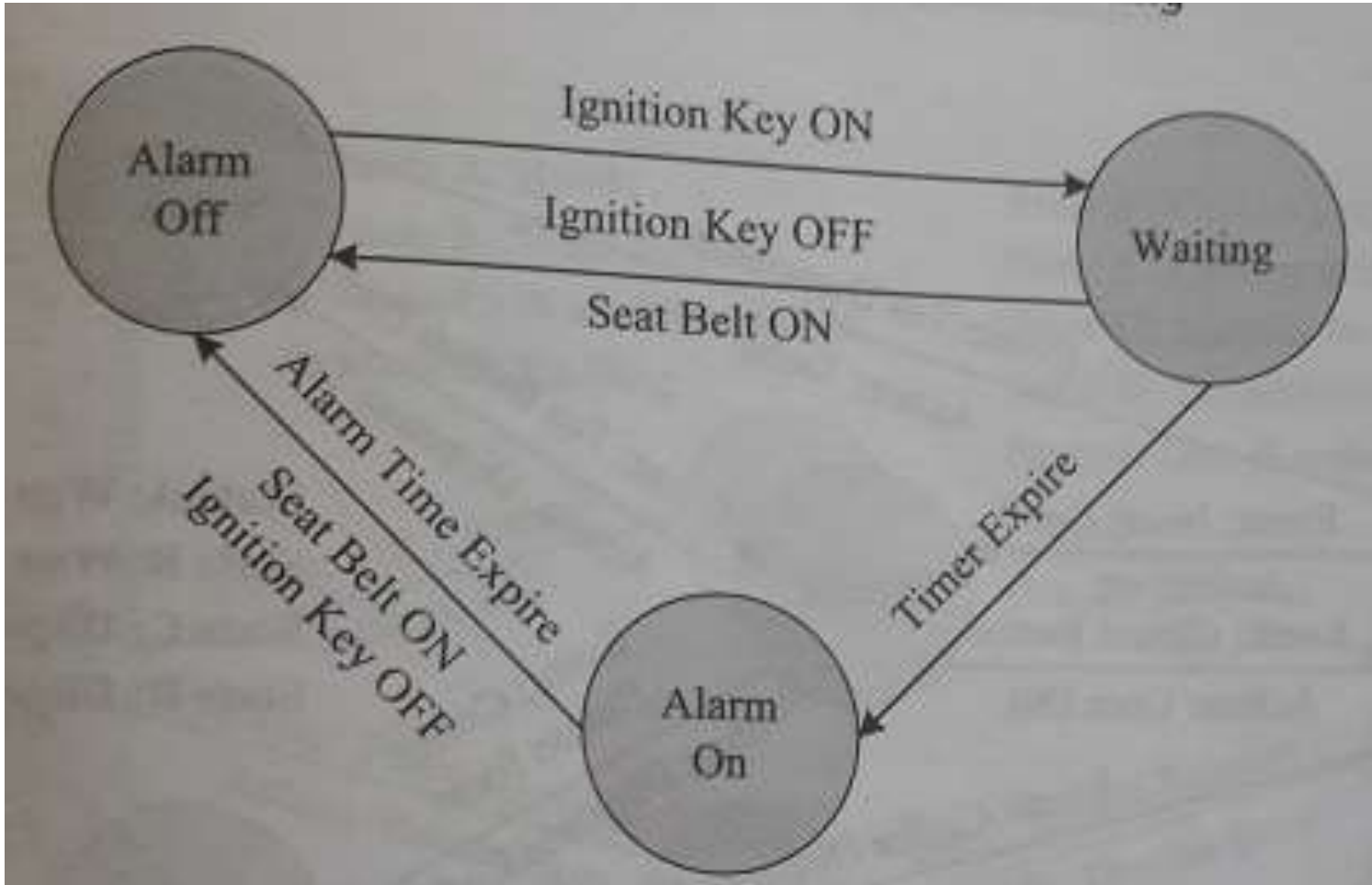
State Machine Model

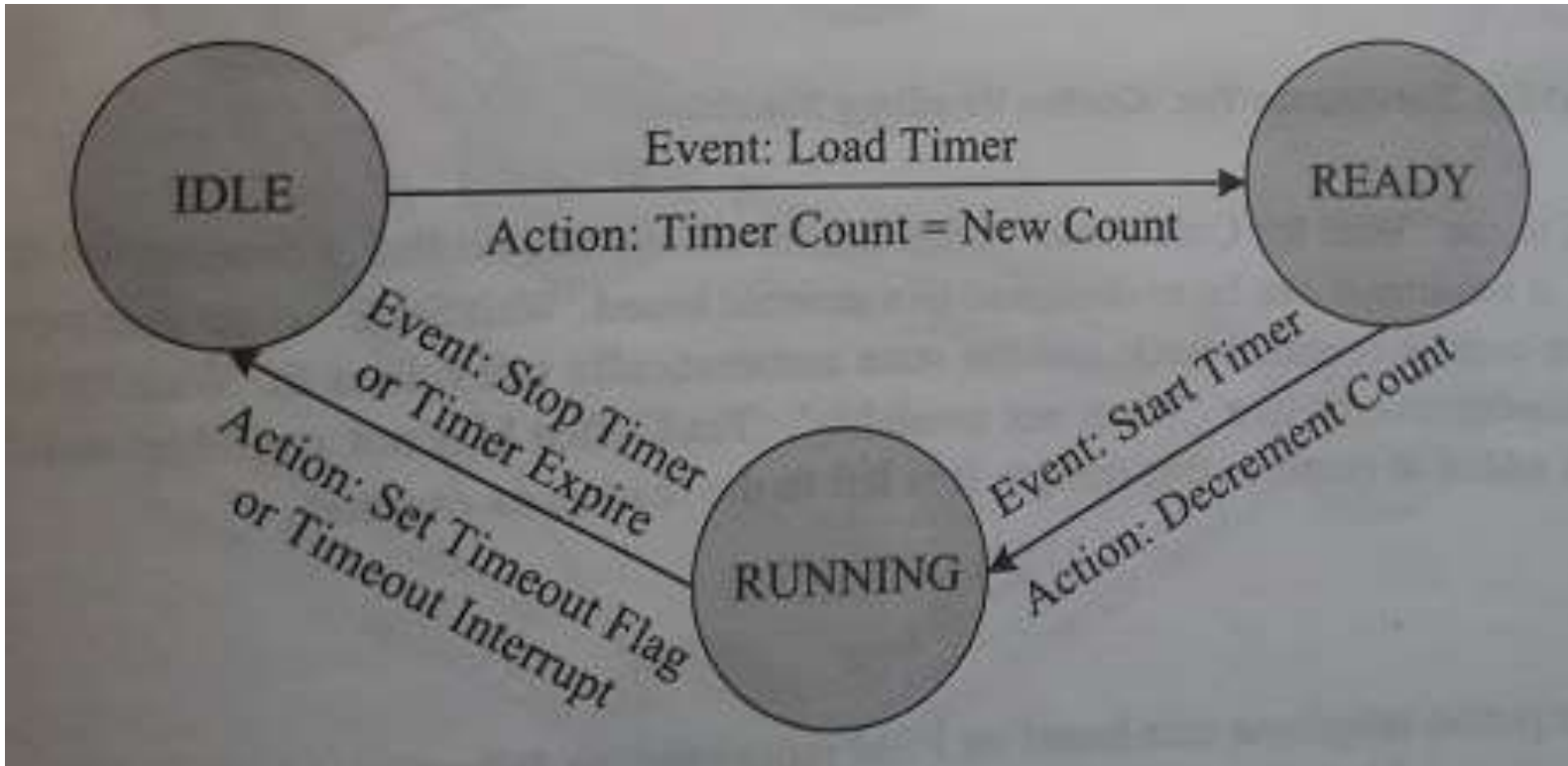
- The state machine model is used for modeling reactive or event driven embedded systems whose processing behavior are dependent on state transitions
- Embedded systems used in the control and industrial applications are typical examples for event driven systems
- The state machine model describes the system behavior with **states**, **events** , **actions** and **transitions**.
 - *The state is a representation of a current situation.*
 - *An event is an input to the state. It acts as a stimuli for state transition*
 - *Transition is the movement from one state to another.*
 - *Action is activity performed by the state machine*

- A FSM model is one in which the number of states are finite.
 - The system is described using a finite number of possible states

Driver/ Passenger seat belt warning

- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds
- The alarm is turned off when the alarm time (5 secs) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first
- Here states are
 - ‘Alarm off’, Waiting, Alarm on,
- Events are
 - Ignition key on, Ignition key off, Timer expire, Alarm time expire and seat belt on



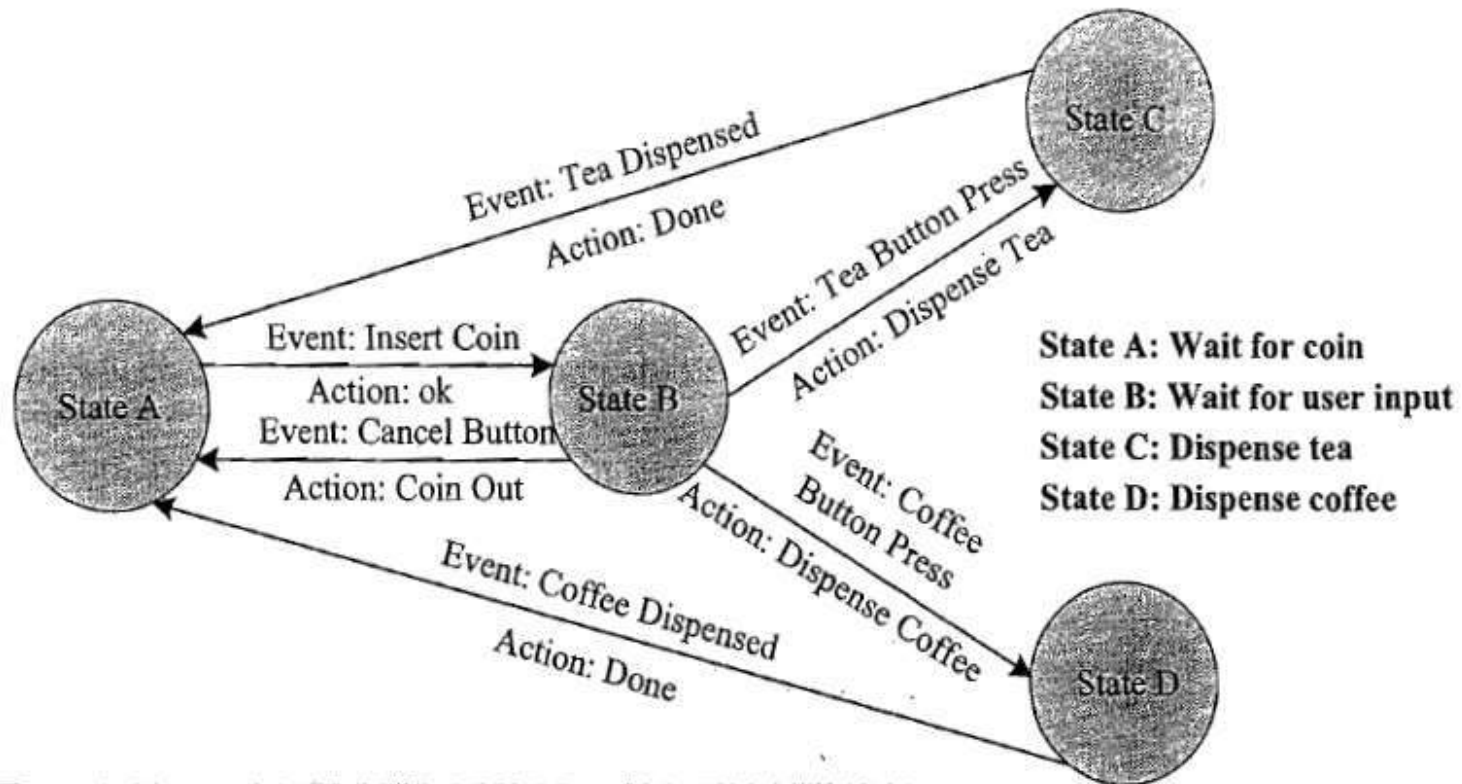


Design an automatic tea/coffee vending machine based on FSM model for the following requirement.

The tea/coffee vending is initiated by user inserting a 5 rupee coin. After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order and take back the coin.

The FSM representation for the above requirement is given in Fig. 7.5.

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



Sequential Program Model

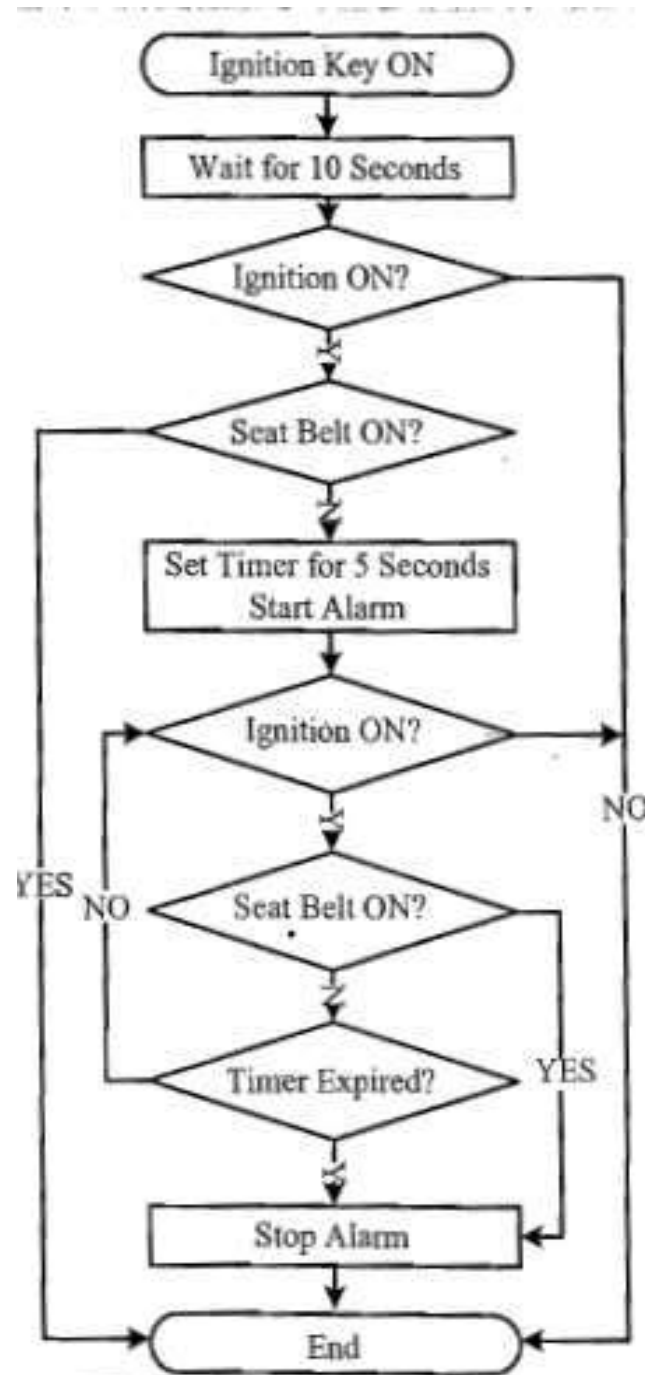
- The functions or processing requirements are executed in sequence
- It is same as the conventional procedural programming
 - Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations
- FSMs are a good choice for sequential program modelling
- Another important tool used for modelling sequential program is flow charts

Flow chart approach for seat belt warning system

Concurrent/ Communicating Process Model

- This model models concurrent executing tasks/processes
- It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution
- Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilization, when the task involves I/O waiting, Sleeping for specified duration
- If the task is divided into multiple subtasks, it is possible to tackle the CPU usage effectively, when the subtask under execution goes to a wait or sleep mode, by switching the task execution
- Concurrent process model requires additional overheads in task scheduling, task synchronization and communication

- Implementing Seat belt warning system in concurrent process model
- We can split the task into
 - Timer task for waiting 10 seconds
 - Task for checking the ignition key status
 - Task for checking seat belt status
 - Task for starting and stopping the alarm
 - Alarm time task for waiting 5 seconds



Object Oriented Model

- The Object Oriented model is an object based model for modeling system requirements
- It disseminates a complex software requirement into a simple well defined pieces called objects
- Object-oriented model brings
 - Re-usability, maintainability and productivity in system design
- In object-oriented modeling object is an entity used for representing or modeling a particular piece of the system
- Each object is characterised by a set of unique behavior and state
- A class is an abstract description of a set of objects and it can be considered as a blue print of an object

Create and initialize events

wait_timer_expire, ignition_on, ignition_off,

seat_belt_on, seat_belt_off,

alarm_timer_start, alarm_timer_expire

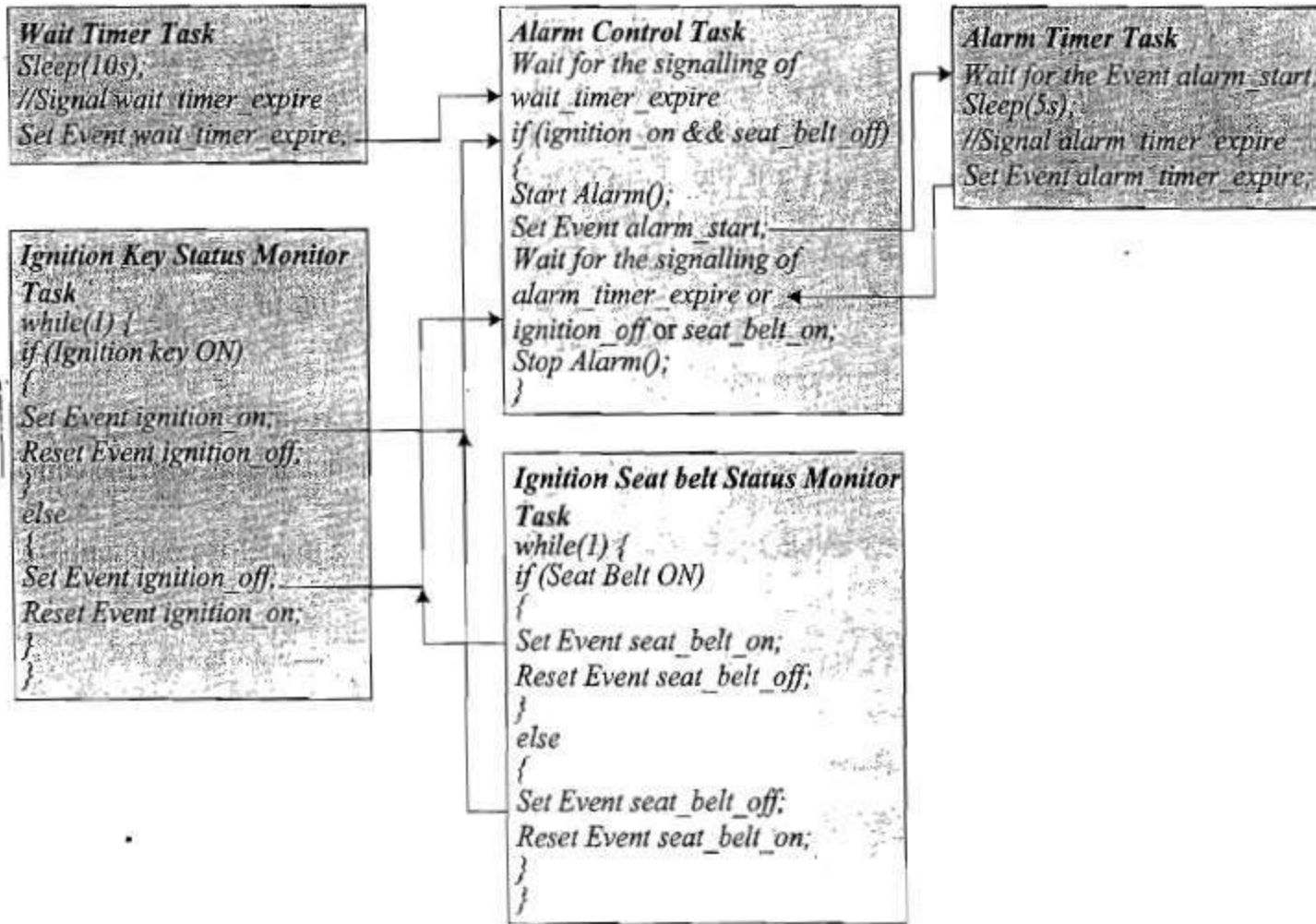
Create task *Wait Timer*

Create task *Ignition Key Status Monitor*

Create task *Seat Belt Status Monitor*

Create task *Alarm Control*

Create task *Alarm Timer*





EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT

53

- Embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements mentioned in the requirements for the particular embedded product
- Firmware is considered as the master brain of the embedded system
- Most of the embedded systems are less adaptive or non-adaptive
- For most of the embedded products the embedded firmware is stored at a permanent memory (ROM)

- Designing embedded firmware requires understanding of the particular embedded product hardware, like:
 - various component interfacing,
 - memory map details
 - I/O port details
 - Configuration and register details of various hardware chips used
 - Some programming language
- Embedded firmware development process starts with the conversion of firmware requirements into a program model
 - Using modeling tools like UML or flow chart based representation
- Once program model is created the next step is the implementation of the tasks and actions by capturing the model using a language

EMBEDDED FIRMWARE DESIGN APPROACHES

- The firmware design approaches for embedded product is purely dependent on:
 - The complexity of the function to be performed
 - The speed of the operation required etc
- The two basic approaches used for embedded firmware design
 - Conventional procedural based design
 - Also known as the **Super Loop Model**
 - Embedded operating system based design

The Super Loop Based Approach

- Adopted for applications that are not time critical and where the response time is not so important
- Very similar to a conventional procedural programming
 - Code is executed task by task
 - The task listed at the top of the program code is executed first and the task just below are executed after completing the first task
- The firmware execution flow for this
 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

```
void main ()
{
    Configurations ();
    Initializations ();
    while (1)
    {
        Task 1 ();
        Task 2 ();
        :
        :
        Task n ();
    }
}
```

- Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation
- Since the tasks are running inside an infinite loop, the only way to ***come out of the loop*** is either a **hardware reset** or an **interrupt assertion**
- Super loop based design doesn't require an operating system
- This type of design is deployed in low cost embedded products
 - Typical example is unit an electronic video game toy containing keypad and display

- The super loop based design is :
 - Simple and straight forward
 - No OS related overheads
- Major drawbacks
 - 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 product stops functioning
 - There are remedial measures for overcoming this
 - Use of hardware and software Watch Dog Timers
 - Lack of real timeliness
 - If the number of tasks to be executed increases the time at which each task is repeated also increases

Embedded Operating System Based Approach

- The operating system based approach contains operating systems
 - General purpose (GPOS) or real time (RTOS)
- General purpose based is very similar to conventional PC based application development
 - Device contains an OS and you will be running user applications on it
- OS based applications also require 'Driver software' for different hardware present on the board to communicate with them
- Real Time OS is employed in products demanding real time response
- RTOS respond in a timely and predictable manner to the events

- Real time OS contains
 - Real Time kernel
 - Responsible for performing pre-emptive multitasking
 - Scheduler for scheduling tasks, multiple threads etc
- Real time OS allows flexible scheduling of system resources like the CPU and memory and offers some way to communicate with the tasks

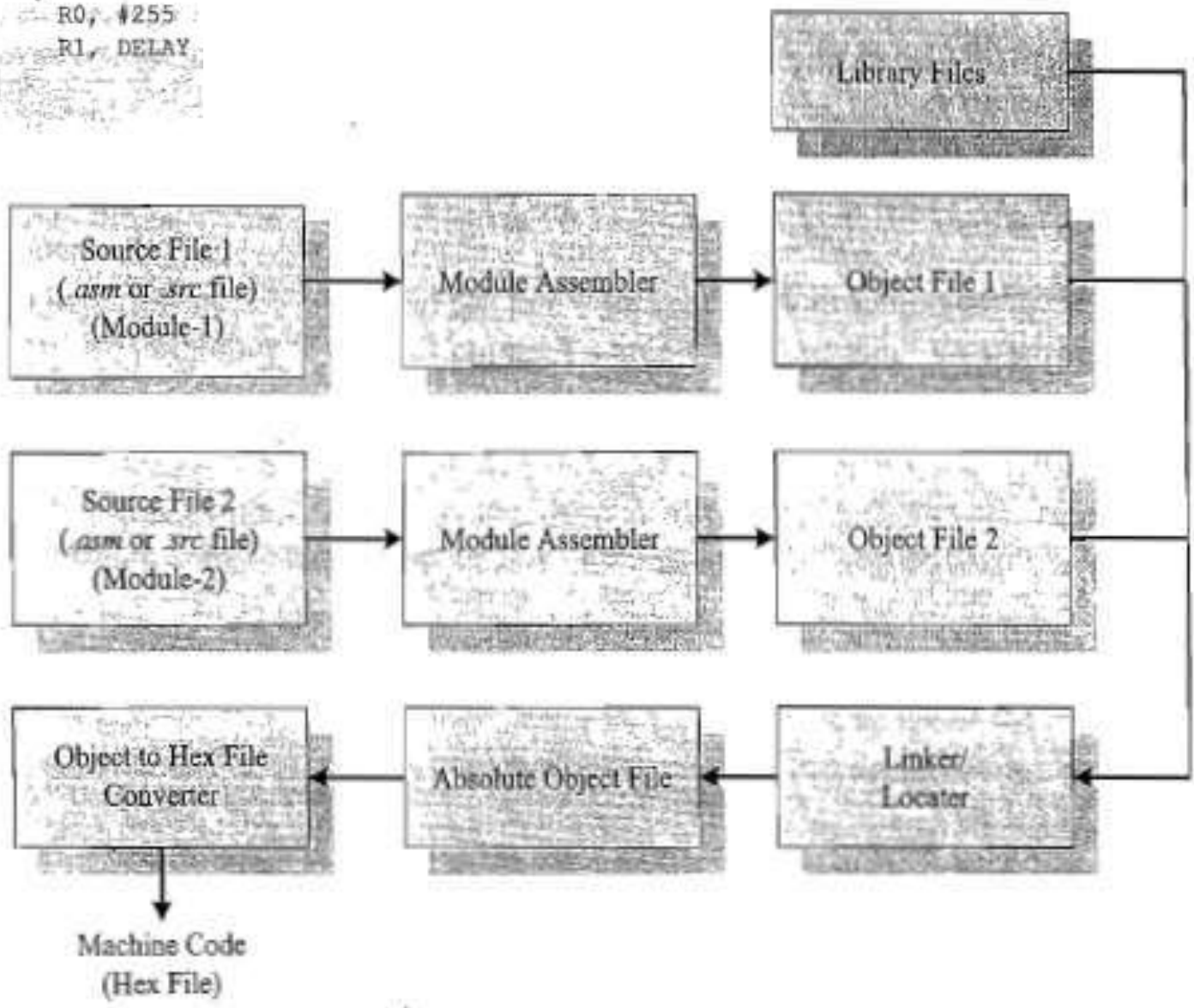
EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

Assembly language based development

- ‘Assembly language’ is human readable notation of machine language
 - Whereas machine language is a processor understandable language
 - Processor only deals with 1’s and 0’s
 - Machine language is made readable by using specific symbols called ‘mnemonics’
 - Assembly language and machine languages are processor/controller dependent and an assembly program written for one processor/controller family will not work with others

LABEL OPCODE OPERAND COMMENTS

```
MOV R0, #255
DJNZ R1, DELAY
RET
```



Advantages of high level language based development

○ Efficient code memory and data memory usage (memory optimization)

- Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations
- This leads to less utilization of code memory and effective utilization of data memory

○ High performance

- Optimized code not only improves the code memory usage but also improves the total system performance
- Through effectively assembling coding, optimum performance can be achieved for a target application

○ Low level hardware access

- Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers and low level interrupt routines etc are making use of direct assembly coding since low level device specific operation support is not commonly available with most of the high-level language cross compilers

○ Code reverse engineering

- It's a process of understanding the technology behind a product by extracting the information from a finished product
- Its performed by hawkers to reveal the technology behind proprietary products
- Though most of them have memory protection, it may be possible to break and read the code memory
- It can be easily 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 organization and register details of the target processor

○ Developer Dependency

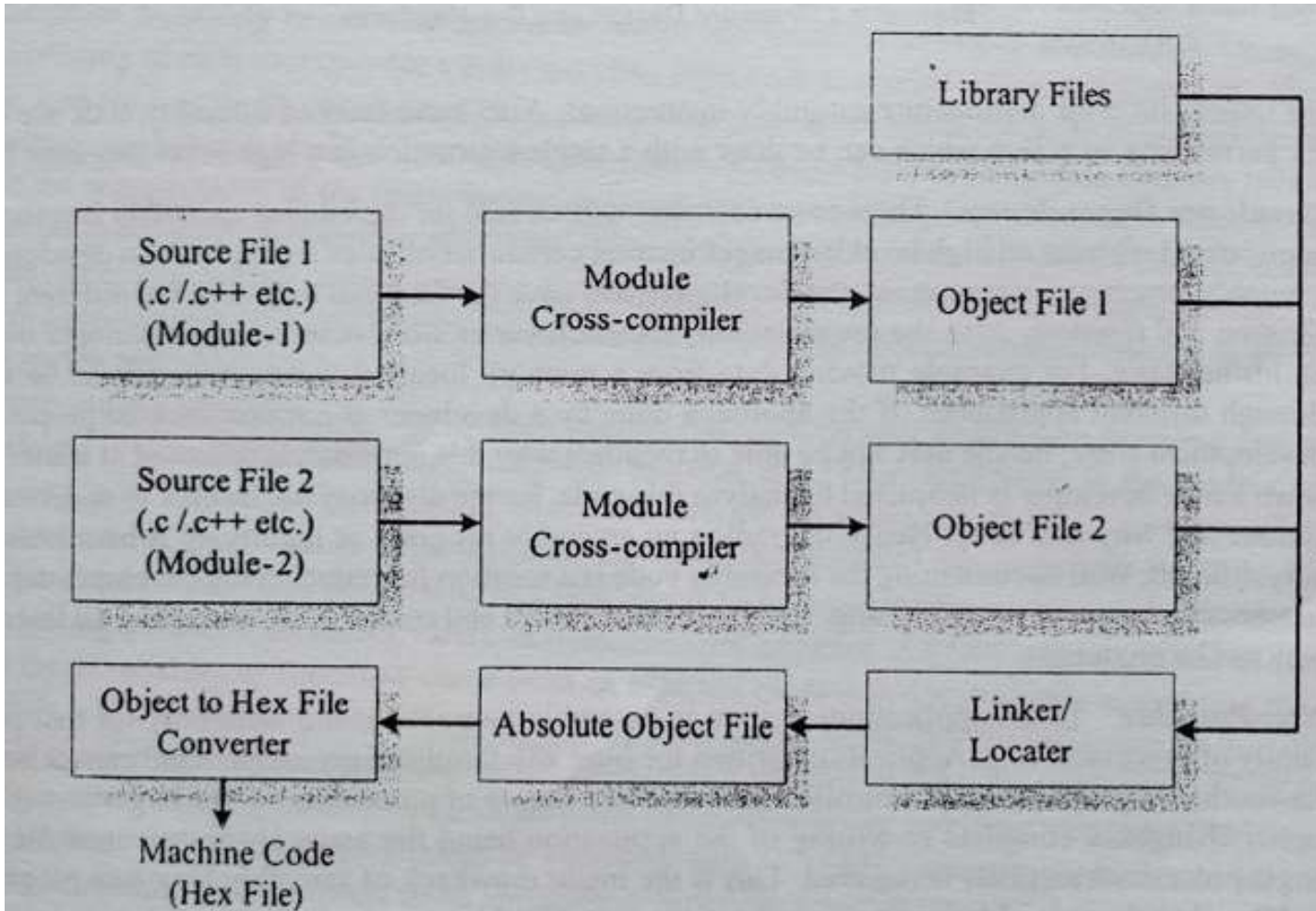
- There is no common written rule for developing assembly language based applications whereas all high level languages instruct certain set of rules for application development.
- In assembly language programming, the developers will have freedom to choose the different memory locations and registers
- Also the programming approach varies from developer to developer depending on his/her taste

○ Non- Portable

- Target application written in assembly instructions are valid only for that particular family of processors and cannot be reused for another target processors/ controllers

HIGH LEVEL LANGUAGE BASED DEVELOPMENT

- Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture.
- Also applications developed in assembly language are non portable.
- Any high level language 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
- Most of the high level languages support modular programming approach and hence you can have multiple source files called modules
- Translation of high level source code to executable object code is done by a cross compiler



Advantages of high level language based development

○ Reduced Development time

- Developer requires less or little knowledge on the internal hardware details and architecture of the target processor/ controller
 - Bare minimal knowledge of the memory organization
 - register details of the target processor in use
 - syntax of the high level language
- The ramp up time required by the developer in understanding the target hardware and target machines assembly instruction is waived off by the cross compiler

○ Developer independency

- The syntax used by most of the high level languages are universal and a program written in high level language can easily be understood by a second person knowing the syntax of the language

○ Portability

- Target applications written in high level languages are converted to target processor/controller understandable format by a cross compiler
- An application written in high level language for a particular target processor can be easily converted to another target processor/controller specific application
 - Involves little or less effort by simply re-compiling/little code modification followed by re-compiling the application for the required target processor/controller

Limitations of high level language based development

- The merits offered by high level language based design take advantage over its limitations
- Some cross compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions
- The investment required for high level language based development tools is high compared to assembly language based firmware development tools
- Hardware access time is critical

MIXING ASSEMBLY AND HIGH LEVEL LANGUAGE

```

RSEG ?PR?_my_assembly_func?TESTCODE
USING 0
?_my_assembly_func:
;---- Variable 'argument?040' assigned to Register 'R6/R7' ----
; SOURCE LINE # 2
;   unsigned int argument)
; {
; SOURCE LINE # 4
; return (argument + 1); // Insert dummy lines to access all args
; and retvals
; SOURCE LINE # 5
;     MOV     A,R7
;     INC     A
;     MOV     R7,A
; }
; SOURCE LINE # 6
?C0001:
;     RET
; END OF _my_assembly_func
;     END
```

PROGRAMMING IN EMBEDDED C

- Whenever the conventional C language and its extension 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 developers have resources in surplus which is not the case with embedded application developer

C

1. Well Structured, well defined and standardized general purpose programming language with extensive bit manipulation support
2. Offers a combination of features of high level language and assembly and helps in hardware access programming
3. Conventional C follows ANSI standard and it incorporates various library files for different OS
4. Platform specific application, known as compiler is used for the conversion of programs written in C to the target processor

Embedded C

1. Can be considered subset of conventional C language
2. Supports all C instructions and incorporates a few target processor specific functions/instructions
3. A software program called cross-compiler is used for the conversion of programs written in embedded C to target processor/ controller specific instructions

Compiler Vs Cross Compiler



RTOS and IDE for Embedded System Design

Operating System Basics

- The OS acts as a bridge between user applications/tasks
- The primary functions of an OS is
 - Make the system convenient to use
 - Organize and manage the system resources efficiently and correctly
- Kernel
 - Core of the OS
 - Responsible for:
 - Managing the system resources and the communication among the hardware and other system services

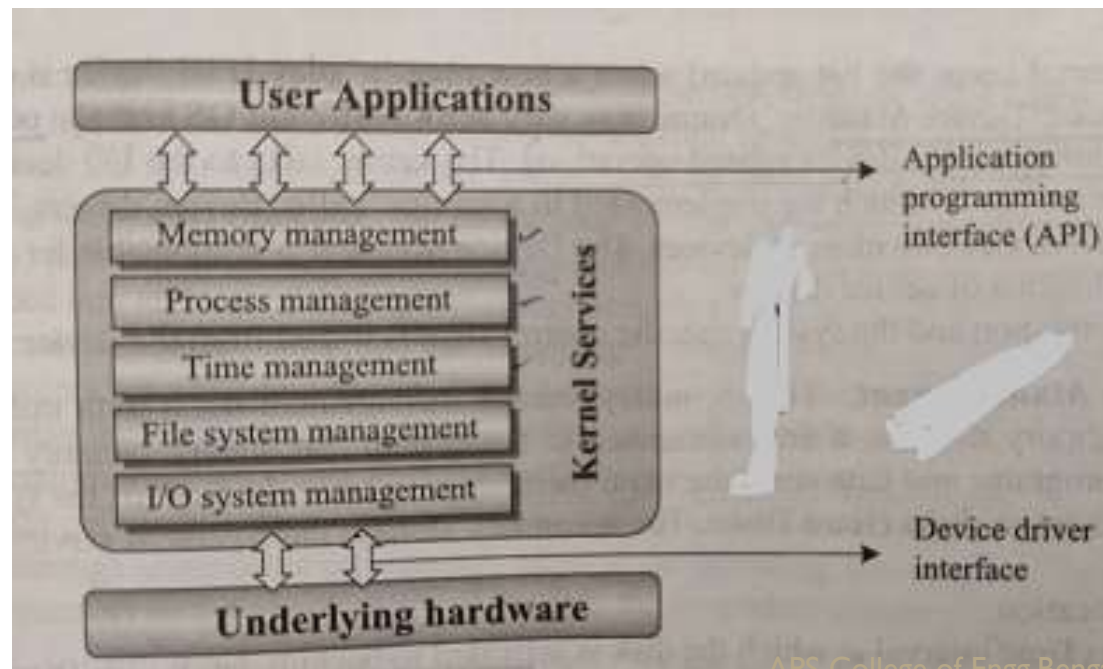
- For a General purpose OS the kernel contains different services for handling the following
 - **Process Management**
 - Setting up memory space
 - Loading the process code
 - Allocating system resources
 - Scheduling and managing execution of process
 - Setting up and managing the process control block
 - Inter process communication and synchronization
 - Process termination/deletion
 - **Primary Memory Management**
 - Refers to RAM
 - The MMU of the kernel is responsible for
 - Keeping track of which part of the memory is being used by which process
 - Allocating and de-allocating memory space on a need basis

• File System Management

- File is a collection of related information
 - E.g program (source code or executable), text files, image files, word documents, audio/video files, etc
- File operation is a useful service provided by the OS
- The file system management service of the kernel is responsible for
 - The creation and deletion and alteration of files
 - Creation, deletion and alteration of directories
 - Saving of files in the secondary storage memory
 - Providing automatic allocation of file space based on the amount of free space available
 - Providing a flexible naming convention of files

- **I/O System (Device) Management**
 - Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system
 - In a well structured OS, the direct accessing of the I/O devices are not allowed
 - Access to them are provided through a set of API (Application Programming Interfaces) exposed by the kernel
 - Kernel maintains a list of all I/O devices of the system

- The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations
- The device manager is responsible for
 - Loading and unloading of device drivers
 - Exchanging information and the system specific control signals to and from the device



• **Secondary Storage management**

- Deals with managing the secondary storage memory devices
- Used as backup medium for programs and data since the main is volatile
- The secondary storage management of kernel deals with
 - Disk storage allocation
 - Disk Scheduling
 - Free Disk space management

• **Protection Systems**

- Most OS support multiple users with different levels of access permissions
- Protection deals with implementing the security policies to restrict the access to both user and system resources

• **Interrupt Handler**

- Kernel provides handler mechanism for all external/internal interrupts generated
- Depending on the type of the OS, a kernel may contain lesser number of components/services or more number of components/services
- Network communication, network management, user interface graphics, timer services, error handler, database management etc are examples for such components/services

• Kernel Space and User Space

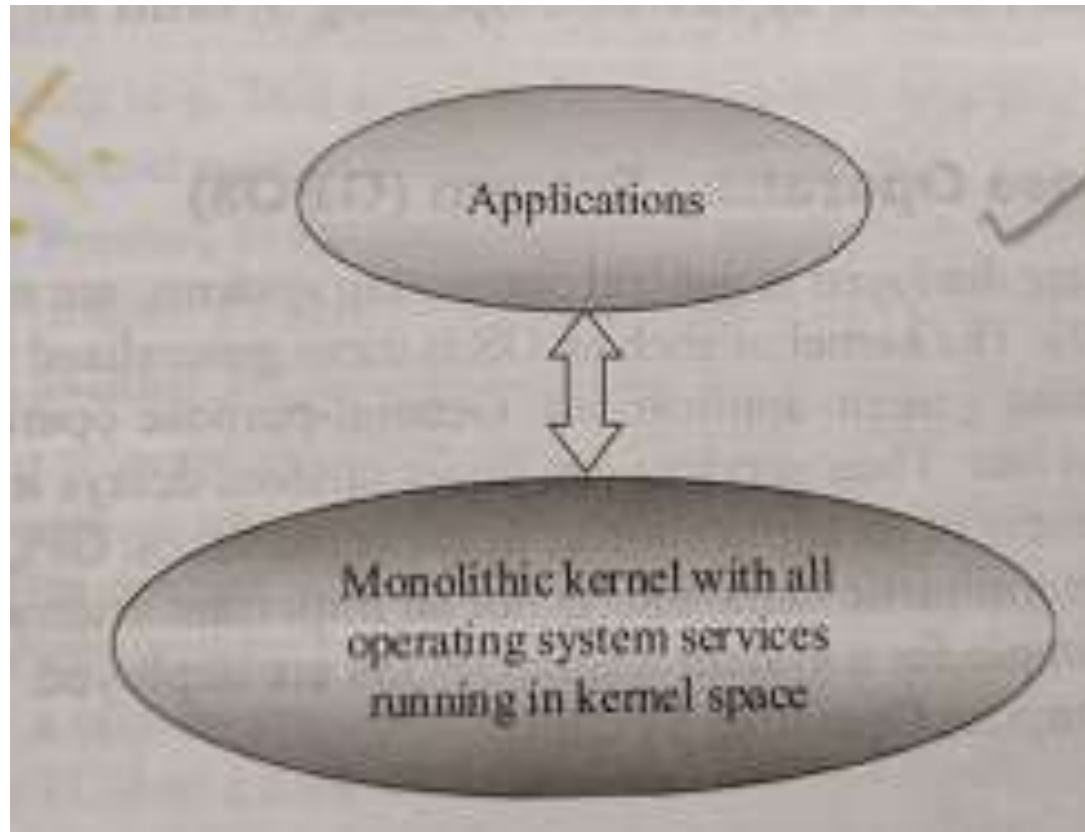
- The application services are classified into two categories, namely
 - User application
 - Kernel application
- The program code corresponding to the kernel application/services are kept in contiguous area of primary memory and is protected from unauthorised access by user programs/applications
- The memory space at which the kernel code is located is known as 'kernel space'
- All user applications are loaded to a specific area of primary memory and this memory area is referred as 'User Space'
 - Memory area where user applications are loaded and executed

- **Monolithic Kernel and Microkernel**

- Kernel forms the heart of an operating system
- Based on the kernel design, they can be classified as 'Monolithic' and 'Micro'

- ***Monolithic Kernel***

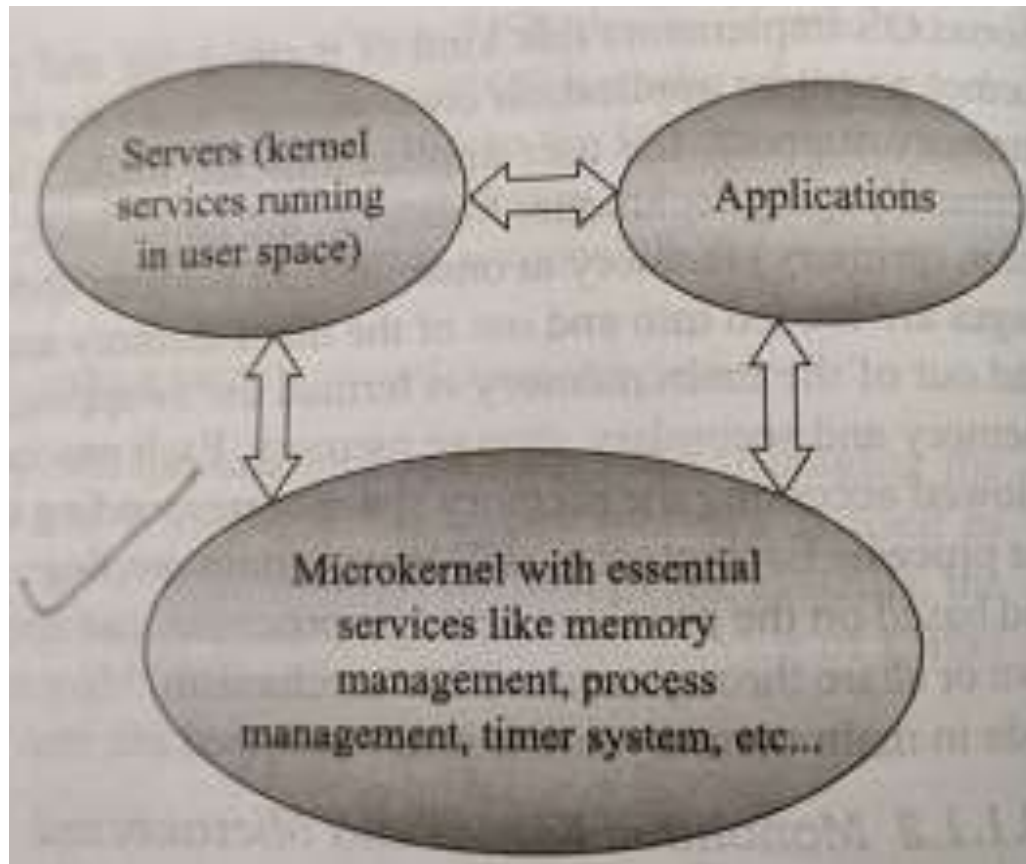
- In this architecture all kernel services run in kernel space
- All kernel modules run within same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- Major drawback is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application
- Ex: LINUX, SOLARIS, MS-DOS



• **Microkernel**

- Incorporates only the essential set of operating system services into 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 abstraction to the kernel
- Memory management, process management, timer systems and interrupt handlers are the essential services which forms the part of microkernel

- **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 services which run as 'servers' are running on a different memory space, the chances of corruption of kernel services are ideally zero
 - **Configurability**
 - Any services, which run as 'server' application can be changed without the need to restart the whole system.
 - This makes the system dynamically configurable



Types of Operating Systems

- OS are classified into different types
- General Purpose Operating System (GPOS)
 - OS which are deployed in general computing systems are referred as GPOS
 - The kernel of such an OS is more generalized and it contains all kinds of services required for executing generic applications
 - Quite non-deterministic in behavior
 - Services can inject random delay and also can cause slow responsiveness of an application

- **Real Time Operating System (RTOS)**
 - Real time implies deterministic timing behavior
 - OS consumes only known and expected amounts of time regardless the number of services
 - RTOS decides which application 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
- **The Real Time kernel**
 - In compliment to the conventional OS kernel, the real time kernel is highly specialized and it contains only the minimal set of services

- The basic functions of a real-time kernel are listed below
 - Task/Process management
 - Task/Process Scheduling
 - Task/Process Synchronization
 - Error/exception handling
 - Memory management
 - Interrupt handling
 - Time Management

• **Task/Process Management**

- Deals with setting up memory space for the tasks
- Loading the tasks code into the memory space
- Allocating system resources
- Setting up a Task control Block (TCB) for the task and task/process termination/deletion
- A TCB is used for holding the information corresponding to a task.

- TCB usually contains the following set of information

Task ID: Task Identification Number

Task State: The current state of the task (e.g. State = 'Ready' for a task which is ready to execute)

Task Type: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.

Task Priority: Task priority (e.g. Task priority = 1 for task with priority = 1)

Task Context Pointer: Context pointer. Pointer for context saving

Task Memory Pointers: Pointers to the code memory, data memory and stack memory for the task

Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc.) used by the task

Task Pointers: Pointers to other TCBs (TCBs for preceding, next and waiting tasks)

Other Parameters Other relevant task parameters

- Task management service utilizes the TCB of a task in the following way
 - Creates a TCB for a task on creating a task
 - Delete/remove the TCB of a task when the task is terminated or deleted
 - Reads the TCB to get the state of a task
 - Update the TCB with updated parameters on need basis
 - Modify the TCB to change the priority of the task dynamically

• **Task/ Process Scheduling**

- Deals with sharing the CPU among various tasks/processes.
- A kernel application called scheduler handles the task scheduling.
- Scheduling is an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide deterministic behavior

• **Task/ Process Synchronization**

- Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks

• **Error/Exception Handling**

- Deals with registering and handling the errors occurred/ exception raised during the execution of tasks
- Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc..
- Errors/exceptions can happen at the kernel level services or at task level
 - Deadlock is an example for kernel level exception
 - Timeout is an example for task level exception

● **Memory Management**

- In general the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated block
- Predictable timing and deterministic behavior are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation
- RTOS makes use of 'block' based memory allocation technique, instead of dynamic memory allocation technique
- RTOS kernel uses blocks of fixed size of dynamic memory and block is allocated for a task on 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

- **Interrupt Handling**

- Deals with handling of various types of interrupts.
- Interrupts provide real-time behavior to systems
- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU
- Interrupts can be either synchronous or Asynchronous.

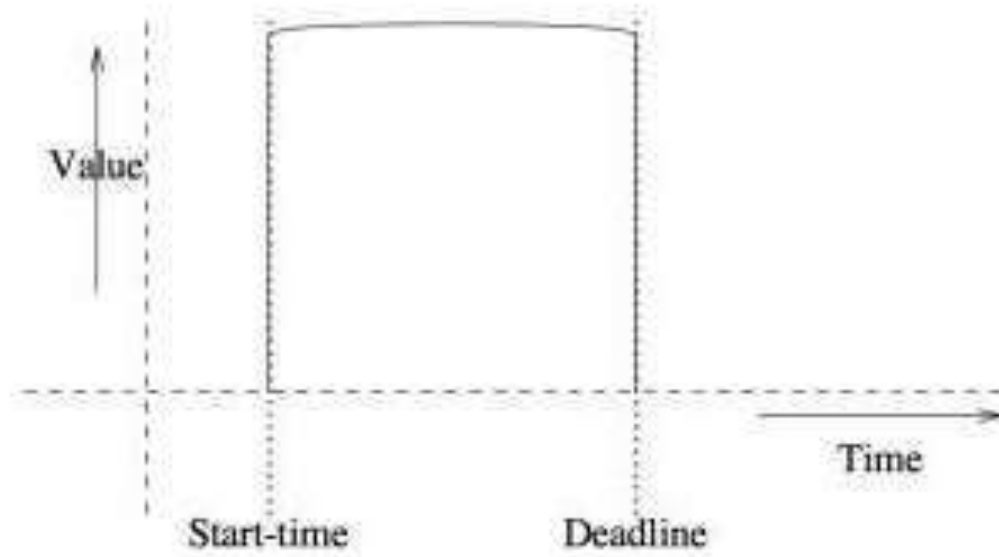
- Interrupts which occurs in sync with the currently executing task is known as Synchronous interrupts.
 - Usually the software interrupts fall under this category
 - Divide by zero, memory segmentation error etc are examples of synchronous interrupts
 - For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task
- Asynchronous interrupts are interrupts which occurs at any point of execution of any task, and are not in sync with currently executing task
- For asynchronous interrupts, the interrupt handler is usually written as separate task and it runs in a different context.

• 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
- The hardware timer is programmed to interrupt the processor/controller at a fixed rate
- This timer interrupt is referred as 'Timer tick'

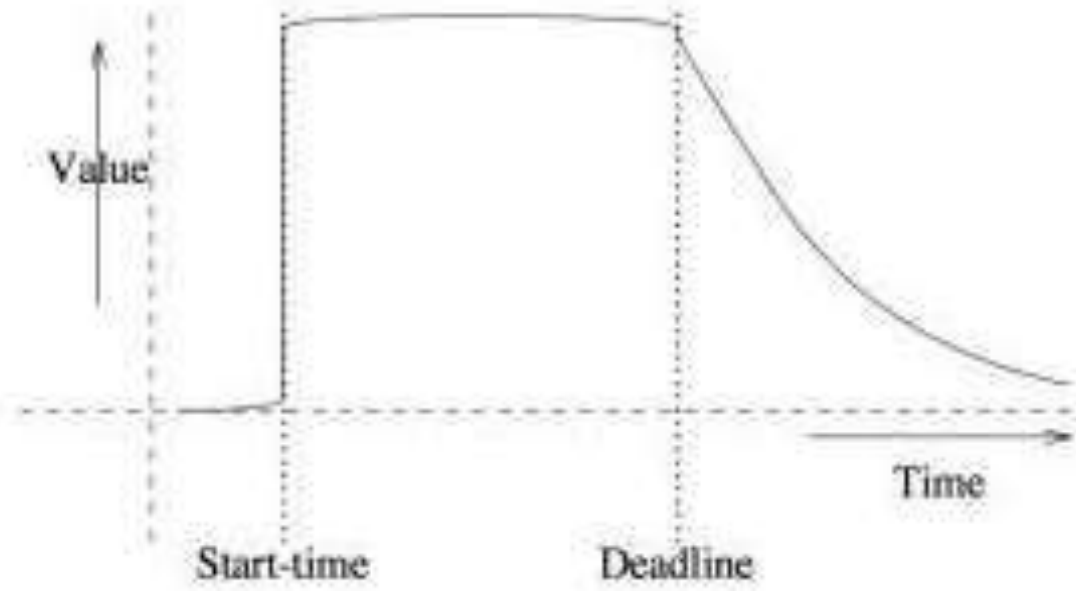
• **Hard Real-Time**

- RTOS that strictly adhere to the timing constraints for a task is referred as 'Hard Real-Time' systems
- A Hard Real-Time system must meet the deadlines for a task without any slippage
- Missing any deadlines may produce catastrophic results for HRT systems
 - Including permanent data loss and irrecoverable damages to the system/users
- HRT systems emphasize the principle 'A late answer is a wrong answer'
 - E.g Air bag control system



• **Soft Real-Time**

- RTOS that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as 'Soft real time' systems
- Missing deadlines for tasks are acceptable, if the frequency of deadline missing is within the compliance limit of the QoS
- SRT emphasizes the principle 'A late answer is an acceptable answer, but it could have been done bit faster'
- ATM is a typical example for SRT





TASKS, PROCESS AND THREADS

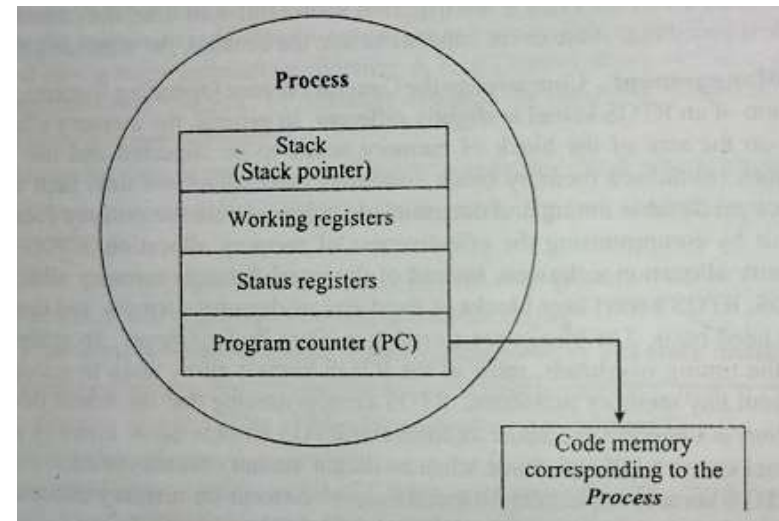
- The term 'Task' refers to something that needs to be done
 - Can be one assigned by our professors/teachers, one related to our personal or family needs
 - We will have an order of priority and schedule/timeline for executing these tasks
- Task is also known as 'Job' in the operating system context
- A program or a part of it in execution is also called a process
- The term 'Task', 'Job' and 'Process' refers to the same entity and used interchangeably

Process

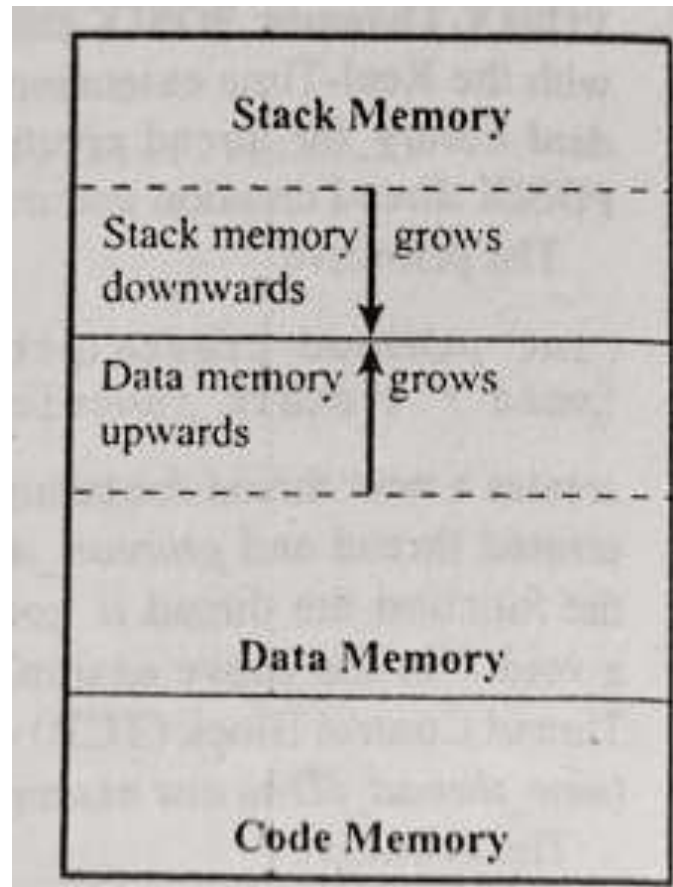
- A 'Process' is a program, or part of it, in execution
- Process is also known as an instance of a program in execution
- Multiple instances of the same program can execute simultaneously.
- A process requires various system resources
 - CPU, memory, I/O devices
- A Process is sequential in execution

The Structure of a Process

- Concept of process leads to concurrent execution of tasks and thereby the efficient utilization of the CPU and other system resources
- Concurrent execution is achieved through the sharing of the CPU among the processes
- A Process which inherits all the properties of the CPU can be considered as a virtual processor

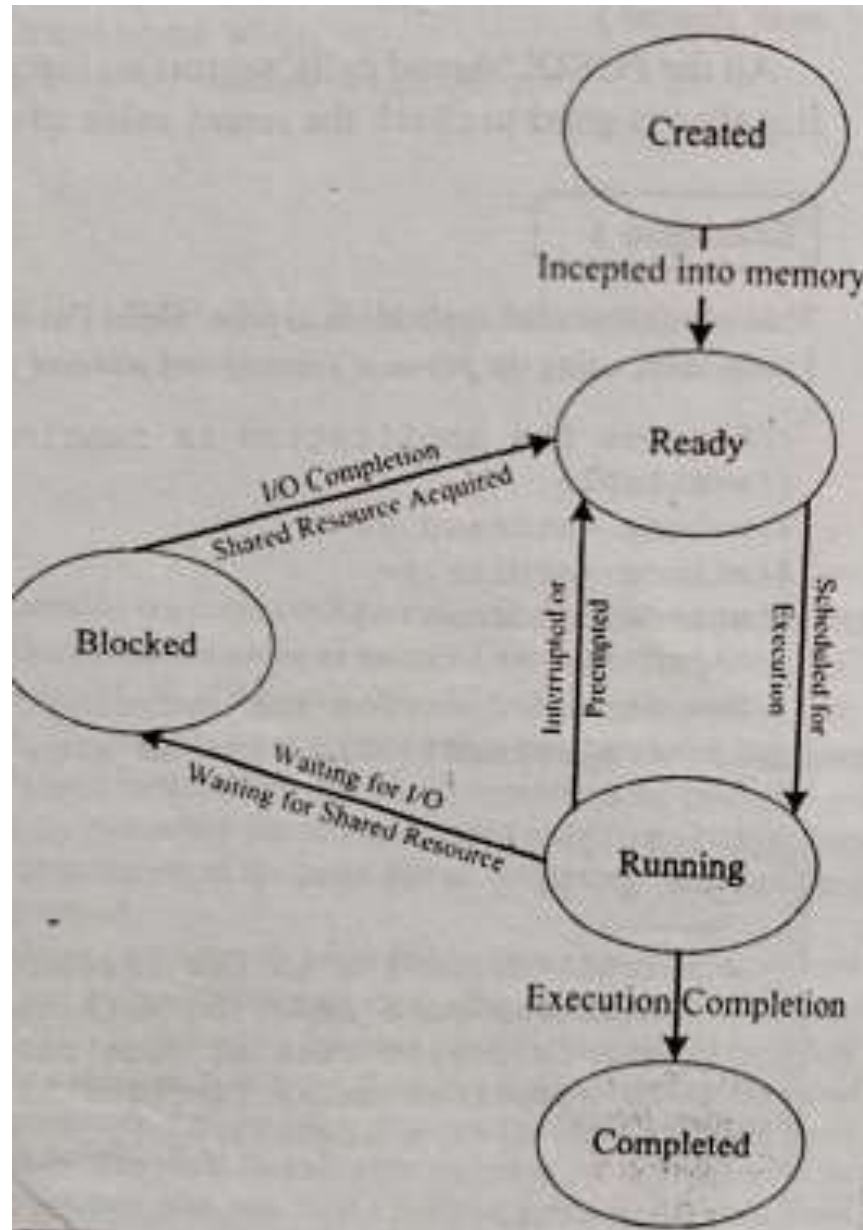


- Memory occupied by the process is segregated into three regions namely, stack, data and code memory



Process States and State Transition

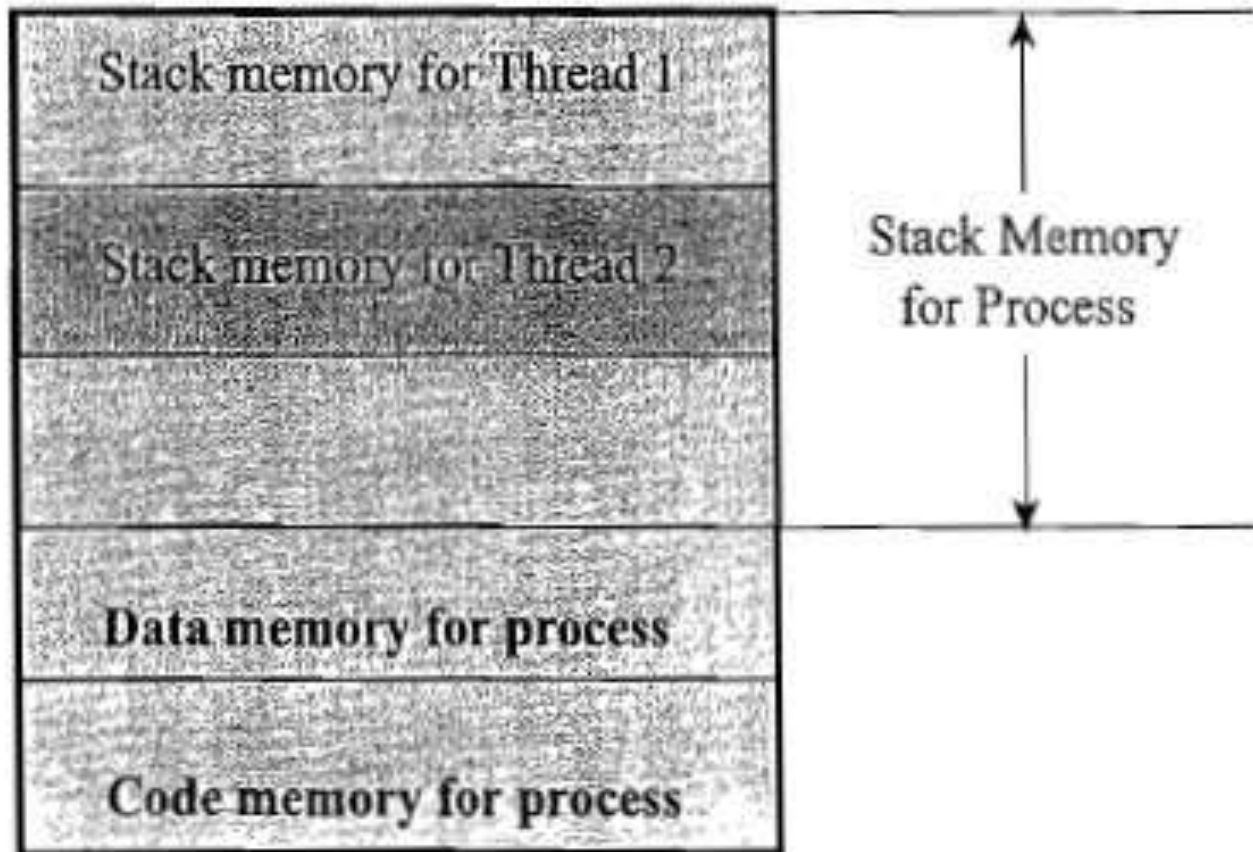
- The process traverses through a series of states during its transition from a newly created state to the terminated state
- The cycle through which a process changes its state from '***newly created***' to '***execution completed***' is known as '***Process life cycle***'



- **Process management deals with:**
 - Creation of a process
 - Setting up a memory space for the process
 - Loading the process's code into the memory space
 - Allocating system resources
 - Setting up a process control block for the process and process termination/ deletion

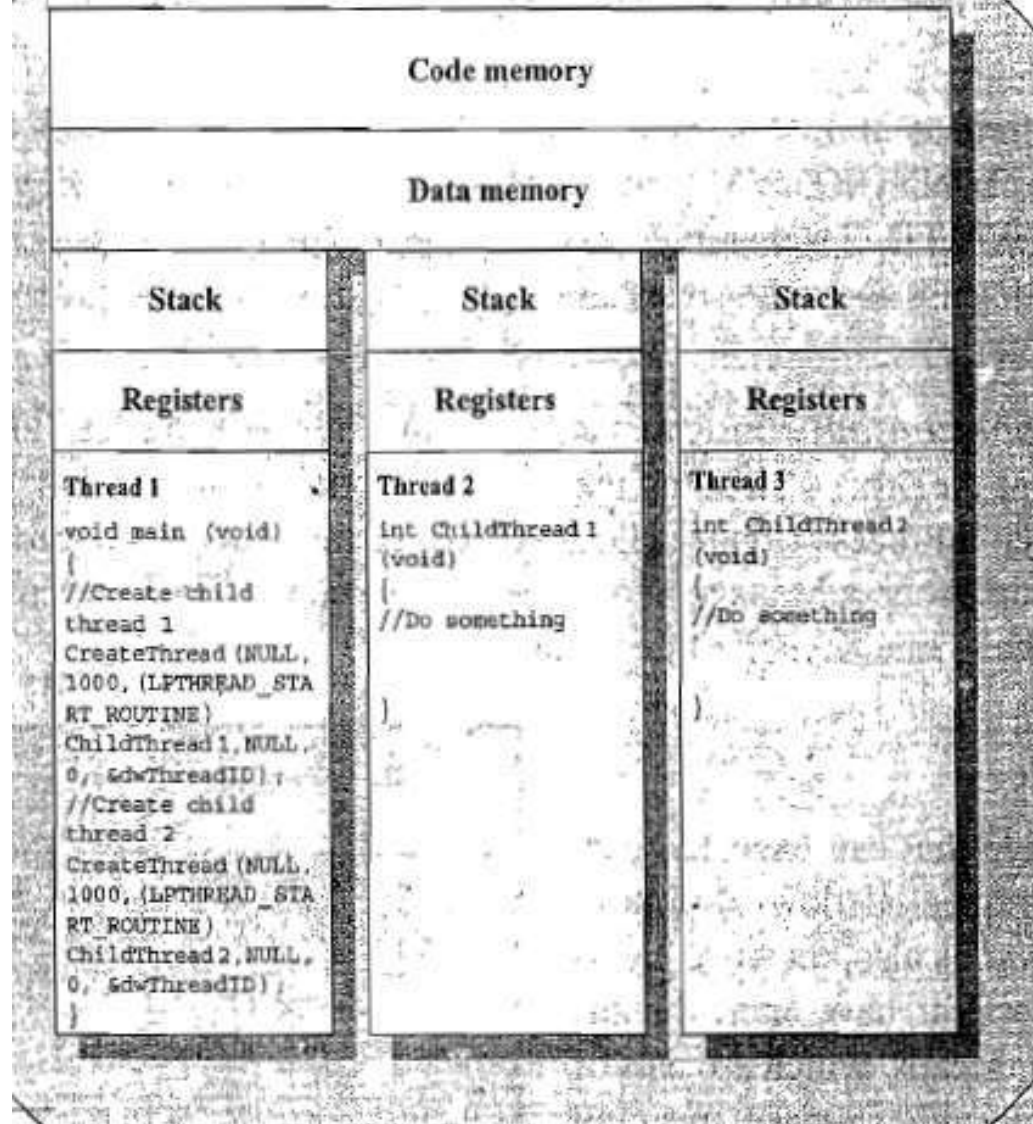
Threads

- 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
 - They share the data memory
 - Code memory
 - Heap memory area
- Threads maintain their own thread status (CPU register values), Program counter and stack




- The concept of multithreading
 - A process/task in embedded application may be a complex or lengthy one and 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 subfunctions of a task are executed in sequence, the CPU utilisation may not be efficient

Task/Process



- 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
- Use of multiple threads to execute a process brings the following advantage
 - **Better memory utilization.**
 - Multiple threads of the same process share the address space for data memory.
 - This also reduces the complexity of inter thread communication since variables can be shared across the threads
 - Since **process is split** into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process
 - **Efficient CPU utilization**, CPU is engaged all the time

- Threads falls into one of the following types
 - **User level thread**
 - 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
 - **Kernel/System Level Thread**
 - Kernel level threads are individual units of execution, which 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 user level threads switching happens only when the currently executing user level thread is voluntarily blocked



- **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

- **One-to-One Model**

- Each user level thread is bonded to a kernel/system level thread

- **Many-to-Many Model**

- Many user level threads are allowed to be mapped to many kernel threads

Thread	Process
Thread is a single unit of execution and is part of process	Process is a program in execution and contains one or more threads
A thread does not have its own data memory and heap memory. It shares 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 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

Preemptive Scheduling

- Preemptive scheduling is employed in systems, which implements preemptive multitasking model
- In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute
 - When and how often each process gets a chance to execute is dependent on the type of preemptive scheduling algorithm used for scheduling the processes

Preemptive SJF Scheduling/ Shortest Remaining Time (SRT)

- The non preemptive SJF scheduling algorithm sorts the ready queue only after completing the execution of the current process or when the process enters the wait state
- 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