



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS & INSTRUMENTATION**

## **UNIT-I**

**EMBEDDED SYSTEM DESIGN— SBMA5201**

# **ARM ARCHITECTURE**

ARM Architecture- ARM Design Philosophy, Registers, Program Status Register, Instruction Pipeline, Interrupts and Vector Table, Architecture Revision, ARM Processor Families.

## **1.History of the ARM Processor**

Developed the first ARM Processor (Acorn RISC Machine) in 1985 at Acorn Computers Limited.

- Established a new company named Advanced RISC Machine Limited and developed ARM6.
- Continuation of the architecture enhancements from the original architecture

## **2.Features of the ARM Processor**

Incorporate features of Berkeley RISC design

- a large register file
- a load/store architecture
- uniform and fixed length instruction field

-simple addressing mode

- Other ARM architecture features

-Arithmetic Logic Unit and barrel shifter

-auto increment and decrement addressing mode

-conditional execution of instructions

- Based on Von Neumann Architecture or Harvard Architecture

### 3.The Evolution of the ARM architecture:

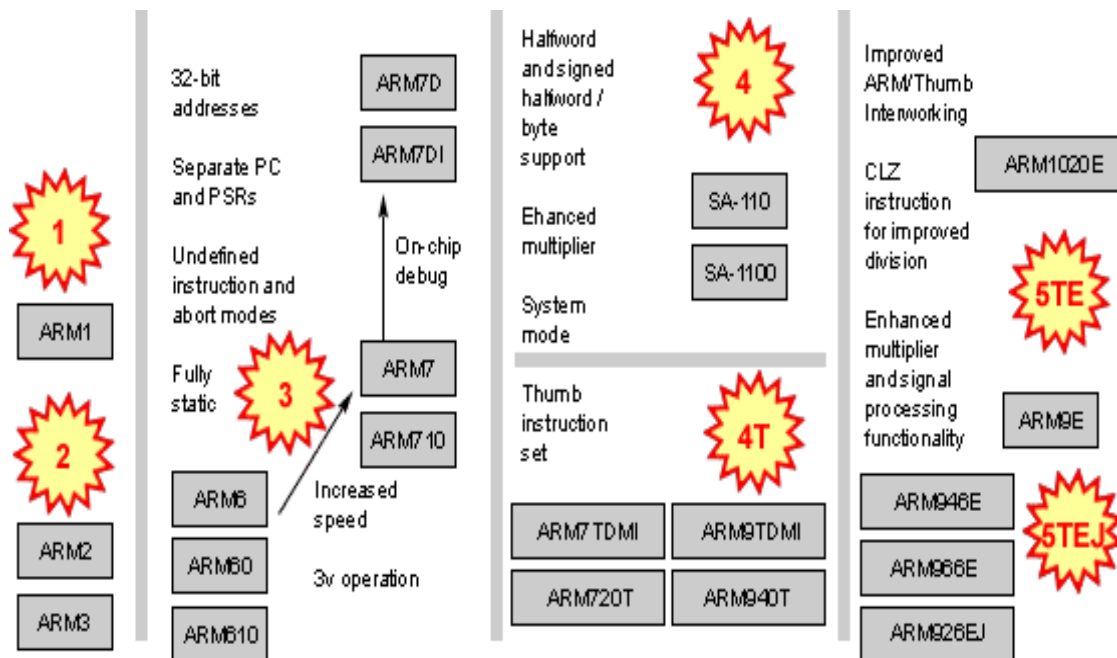


Figure 1.1 ARM Families

Architecture V1 was implemented only in the ARM1 CPU and was not utilized in a commercial product. Architecture V2 was the basis for the first shipped processors. These two architectures were developed by Acorn Computers before ARM became a company in 1990.

After that introduced ARM the Architecture V3, which included many changes over its predecessors. These changes resulted in an extremely small and power-efficient processor suitable for embedded systems. Architecture V4, co-developed by ARM and Digital Electronics Corporation, resulted in the Strong ARM series of processors. These processors are very performance-centric and do not include the on chip debug extensions.

This architecture was further developed to include the Thumb 16-bit instruction set architecture enabling a 32-bit processor to utilize a 16-bit system. Today, ARM only licenses cores based on Architecture V4T or above.

The latest architectures, version 5TE and 5TEJ, embody added instructions for DSP applications and the Jazelle-Java extensions, respectively.

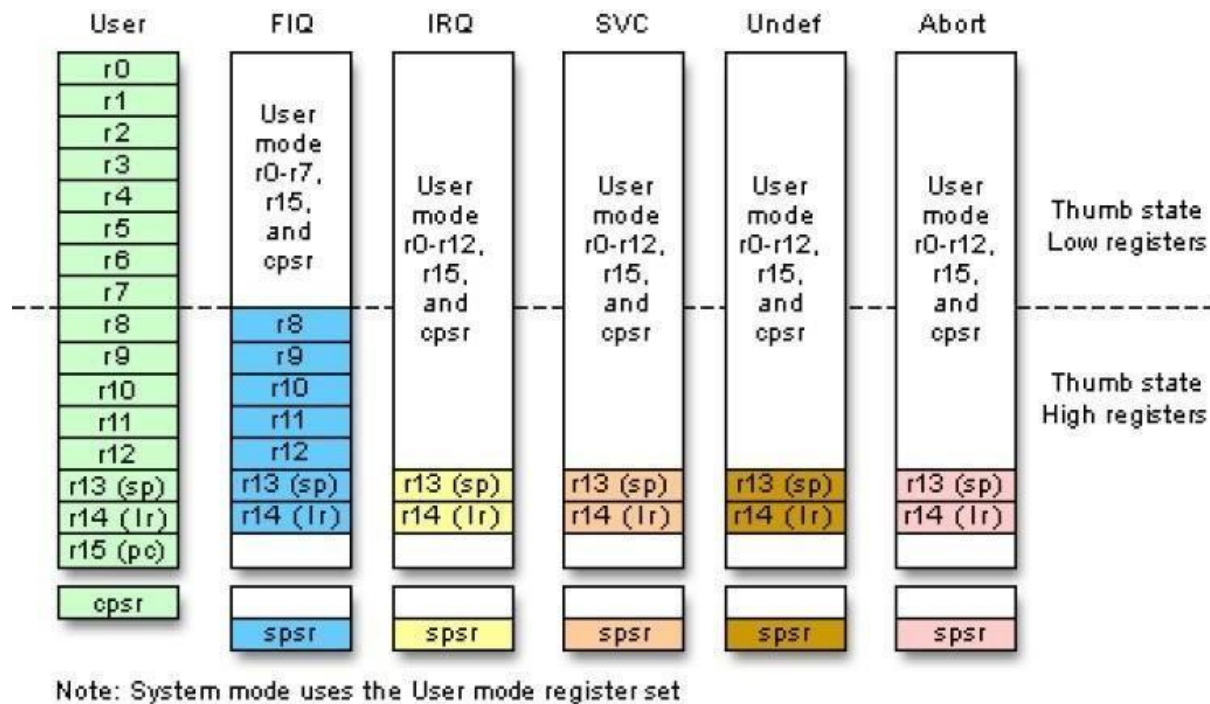
Currently, the ARM9E and 10E family of processors are the only implementations of these architectures. Details on these architectures and cores will be provided later in the course.

## **4.Architecture basics**

ARM cores use a 32-bit, Load-Store RISC architecture. That means that the core cannot directly manipulate the memory. All data manipulation must be done by loading registers with information located in memory, performing the data operation and then storing the value back to memory. There are 37 total registers in the processor. However, that number is split among seven different processor modes. The seven processor modes are used to run user tasks, an operating system, and to efficiently handle exceptions such as interrupts. Some of the registers within each mode are reserved for specific use by the core, while most are available for general use. The reserved registers that are used by the core for specific functions are r13 is commonly used as the stack pointer (SP), r14 as a link register (LR), r15 as a program counter (PC), the Current Program Status Register (CPSR), and the Saved Program Status Register (SPSR).

The SPSR and the CPSR contain the status and control bits specific to the properties the processor core is operating under. These properties define the operating mode, ALU status flags, interrupt disable/enable flags and whether the core is operating in 32-bit ARM or 16-bit Thumb state.

There are 37 total registers divided among seven different processor modes. Figure 09 shows the bank of registers visible in each mode. User mode, the only non-privileged mode, has the least number of total registers visible. It has noSPSR and limited access to the CPSR. FIQ and IRQ are the two interrupt modes of the CPU



**Figure 1.2 Different modes of ARM**

There are 37 total registers divided among seven different processor modes. Figure 02 shows the bank of registers visible in each mode. User mode, the only non-privileged mode, has the least number of total registers visible. It has no SPSR and limited

access to the CPSR. FIQ and IRQ are the two interrupt modes of the CPU. Supervisor mode is the default mode of the processor on start up or reset. Undefined mode traps unknown or illegal instructions when they are passed through the pipeline. Abort mode traps illegal memory accesses as a result of fetching instructions or accessing data.

Finally, system mode, which uses the user mode bank of registers, was introduced to provide an additional privileged mode when dealing with nested interrupts.

Each additional mode offers unique registers that are available for use by exception handling routines. These additional registers are the minimum number of registers required to preserve the state of the processor, save the location in code, and switch between modes.

FIQ mode, however, has an additional five banked registers to provide more flexibility and higher performance when handling critical interrupts.

When the ARM core is in Thumb state, the registers banks are split into low and high register domains. The majority of instructions in Thumb state have a 3-bit register specifier. As a result, these instructions can only access the low registers in Thumb, R0 through R7. The high registers,

R8 through R15, have more restricted use. Only a few instructions have access to these registers.

## **TDMI**

stands for:

- **T**humb, which is a 16-bit instruction set extension to the 32-bit ARM architecture, referred as states of the processor.
- **"D"** and **"I"** together comprise the on-chip debug facilities offered on all ARM cores. These stand for the **D**ebug signals and Embedded **I**CE logic, respectively.
- The **M** signifies the support for 64-bit results and an enhanced multiplier, resulting in higher performance. This multiplier is now standard on all ARMv4 architectures and above.

## **5. Thumb 16-bit Instructions**

With growing code and data size, memory contributes to the system cost. The need to reduce memory cost leads to smaller code size and the use of narrower memory. Therefore ARM developed a modified instruction set to give market-leading code density for compiled standard C language.

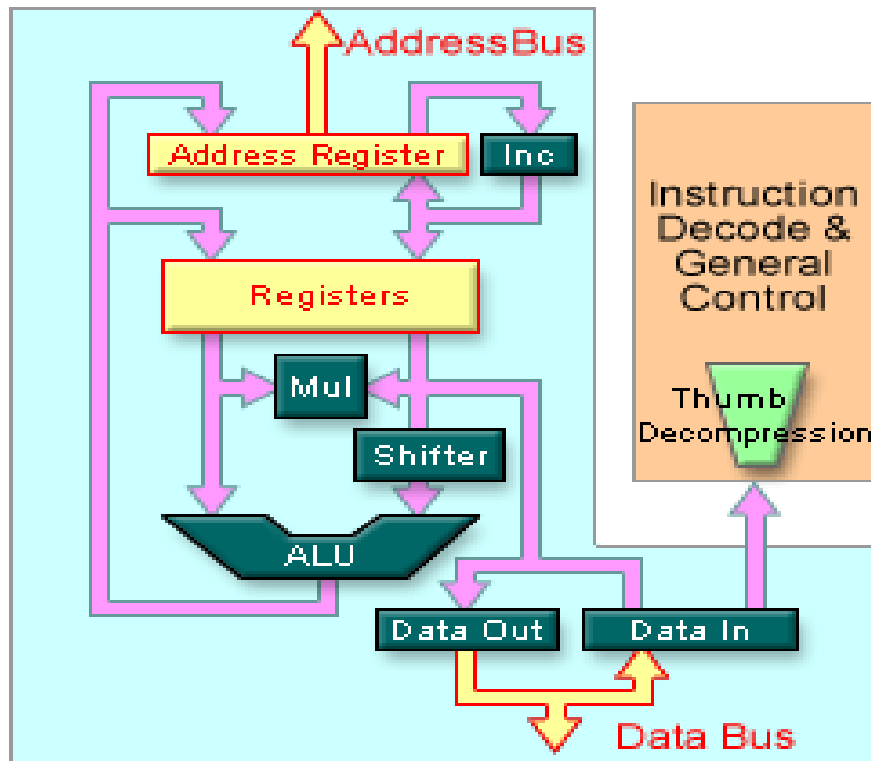
There is also the problem of performance loss due to using a narrow memory path, such as a 16-bit memory path with a 32-bit processor.



The processor must take two memory access cycles to fetch an instruction or read and write data. To address this issue, ARM introduced another set of reduced 16-bit instructions labeled Thumb, based on the standard ARM 32-bit instruction set.

For Thumb to be used, the processor must go through a change of state from ARM to Thumb in order to begin executing 16-bit code. This is because the default state of the core is ARM. Therefore, every application must have code at boot up that is written in ARM. If the application code is to be compiled entirely for Thumb, then the segment of ARM boot code must change the state of the processor. Once this is done, 16-bit instructions are fetched seamlessly into the pipeline without any result.

It is important to note that the architecture remains the same. The instruction set is actually a reduced set of the ARM instruction set and only the instructions are 16-bit; everything else in the core still operates as 32-bit. An application code compiled in Thumb is 30% smaller on average than the same code compiled in ARM and normally 30% faster when using narrow 16-bit memory systems.



**Figure 1.3 Register Bank**

Figure 1.3 shows the register bank in the center of the diagram, plus the required address bus and data bus. The multiplier, in-line barrel shifter, and ALU are also shown. In addition, the diagram illustrates the in-line decompression process of Thumb instructions while in the decode stage of the pipeline. This process creates a 32-bit ARM equivalent instruction from the 16-bit Thumb instruction, decodes the instruction, and passes it on to the execute stage.

## 6.ARM design philosophy

Small processor for lower power consumption (for embedded system)

- High code density for limited memory and Physical size restrictions
- The ability to use slow and low-cost memory
- Reduced die size for reducing manufacture cost and accommodating more peripherals

### 6.1 Registers

ARM has 37 registers all of which are 32-bits long. 1 dedicated program counter □ 1 dedicated current program status register □ 5 dedicated saved program status registers □ 30 general purpose registers □ The current processor mode governs which of several banks is □ accessible. Each mode can access a particular set of r0-r12 registers □ a particular r13 (the stack pointer, sp) and r14 (the link register, lr) □ the program counter, r15 (pc) □ the current program status register, cpsr □ Privileged modes (except System) can also access a particular spsr (saved program status reg

The ARM1136JF-S processor has a total of 37 registers:

- 31 general-purpose 32-bit registers

- six 32-bit status registers. These registers are not all accessible at the same time. The processor state and operating mode determine which registers are available to the programmer

The ARM state register set In ARM state, 16 general registers and one or two status registers are accessible at any time. In privileged modes, mode-specific banked registers become available.

The ARM state register set contains 16 directly-accessible registers, r0-r15. Another register, the Current Program Status Register (CPSR), contains condition code flags, status bits, and current mode bits. Registers r0-r13 are general-purpose registers used to hold either data or address values. Registers r14, r15, and the SPSR have the following special functions

Link Register Register r14 is used as the subroutine Link Register (LR). Register r14 receives the return address when a Branch with Link (BL or BLX) instruction is executed. You can treat r14 as a general-purpose register at all other times. The corresponding banked registers r14\_svc, r14\_irq, r14\_fiq, r14\_abt, and r14\_und are similarly used to hold the return values when interrupts and exceptions arise, or when BL or BLX instructions are executed within interrupt or exception routines.

Program Counter Register r15 holds the PC:

- in ARM state this is word-aligned
- in Thumb state this is halfword-aligned
- in Java state this is byte-aligned. Saved Program Status Register

In privileged modes, another register, the Saved Program Status Register (SPSR), is accessible. This contains the condition code flags, status bits, and current mode bits saved as a result of the exception that caused entry to the current mode.














Banked registers have a mode identifier that indicates which mode they relate to. These mode identifiers are listed in Table **Register mode identifiers**

Mode	Mode identifier
User	usr <sup>a</sup>
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usr <sup>a</sup>
Undefined	und






- a. The usr identifier is usually omitted from register names. It is only used in descriptions where the User or System mode register is specifically accessed from another operating mode.

FIQ mode has seven banked registers mapped to r8–r14 (r8\_fiq– r14\_fiq). As a result many FIQ handlers do not have to save any


registers. The Supervisor, Abort, IRQ, and Undefined modes each have alternative mode-specific registers mapped to r13 and r14, permitting a private stack pointer and link register for each mode

ARM state general registers and program counter					
System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	 r8_fiq	r8	r8	r8	r8
r9	 r9_fiq	r9	r9	r9	r9
r10	 r10_fiq	r10	r10	r10	r10
r11	 r11_fiq	r11	r11	r11	r11
r12	 r12_fiq	r12	r12	r12	r12
r13	 r13_fiq	r13_svc	 r13_abt	 r13_irq	 r13_und
r14	 r14_fiq	r14_svc	 r14_abt	 r14_irq	 r14_und
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

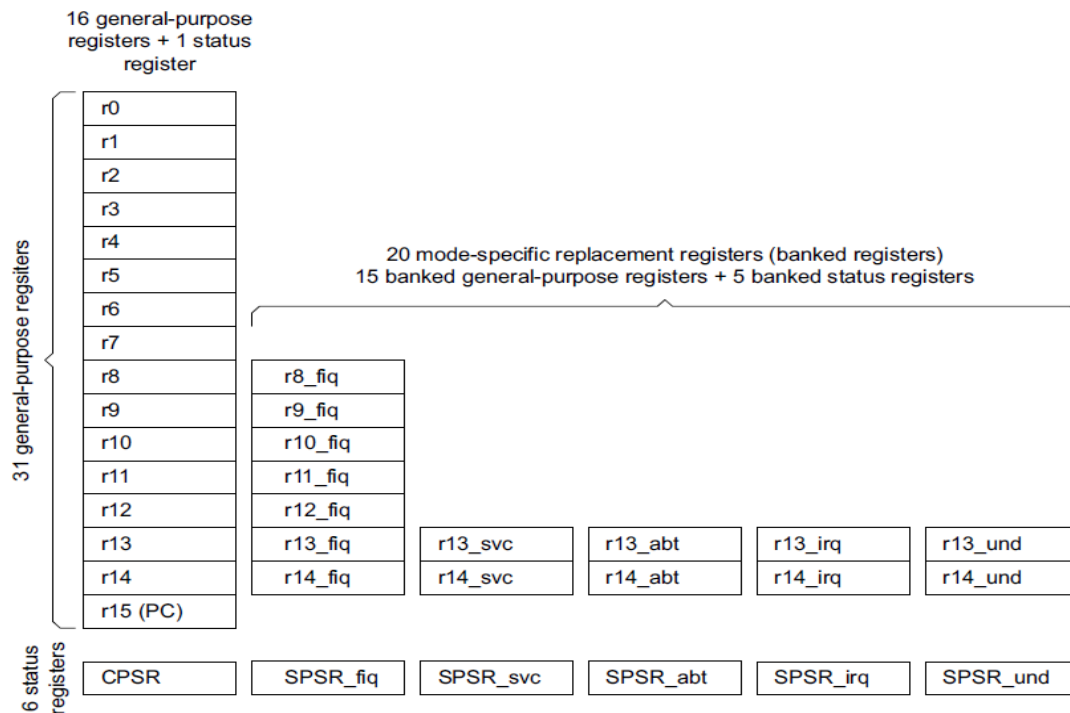
  

ARM state program status registers					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und

 = banked register

**Figure 1.4 register set showing banked registers**



**Figure 1.5 ARM register**











## 7. Thumb state register set

The Thumb state register set is a subset of the ARM state set. The programmer has direct access to:

- Eight general registers, r0–r7
- The PC
- A stack pointer, SP (ARM r13)
- An LR (ARM r14)
- The CPSR.


There are banked SPs, LRs, and SPSRs for each privileged mode.

### Thumb state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
SP	 SP_fiq	 SP_svc	 SP_abt	 SP_irq	 SP_und
LR	 LR_fiq	 LR_svc	 LR_abt	 LR_irq	 LR_und
PC	PC	PC	PC	PC	PC

### Thumb state program status registers

CPSR	 CPSR	 CPSR	 CPSR	 CPSR	 CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

 = banked register

**Figure 1.6 THUMB register**

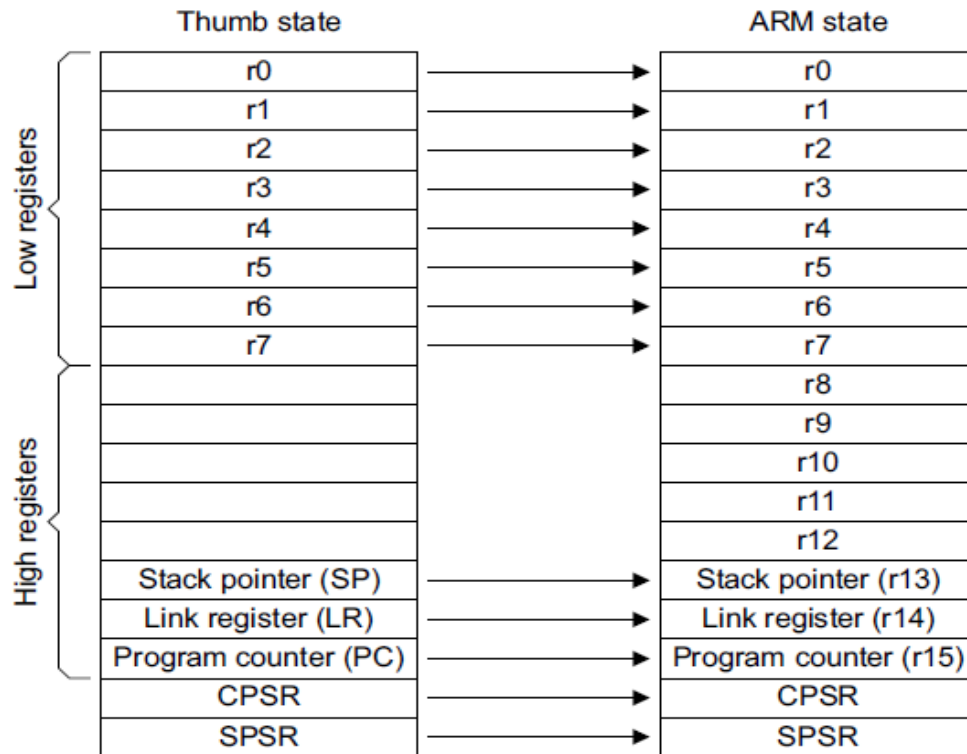
## 7.1 Accessing high registers in Thumb state

In Thumb state, the high registers, r8–r15, are not part of the standard register set. You can use special variants of the MOV instruction to transfer a value from a low register, in the range r0–r7, to a high register, and from a high register to a low register. The CMP instruction enables you to compare high register values with low register values. The ADD instruction enables you to add high register values to low register values.



## 7.2 ARM state and Thumb state registers relationship

Figure 1. 2-1.6 shows the relationships between the Thumb state and ARM state registers



**Figure 2-6 ARM state and Thumb state registers relationship**

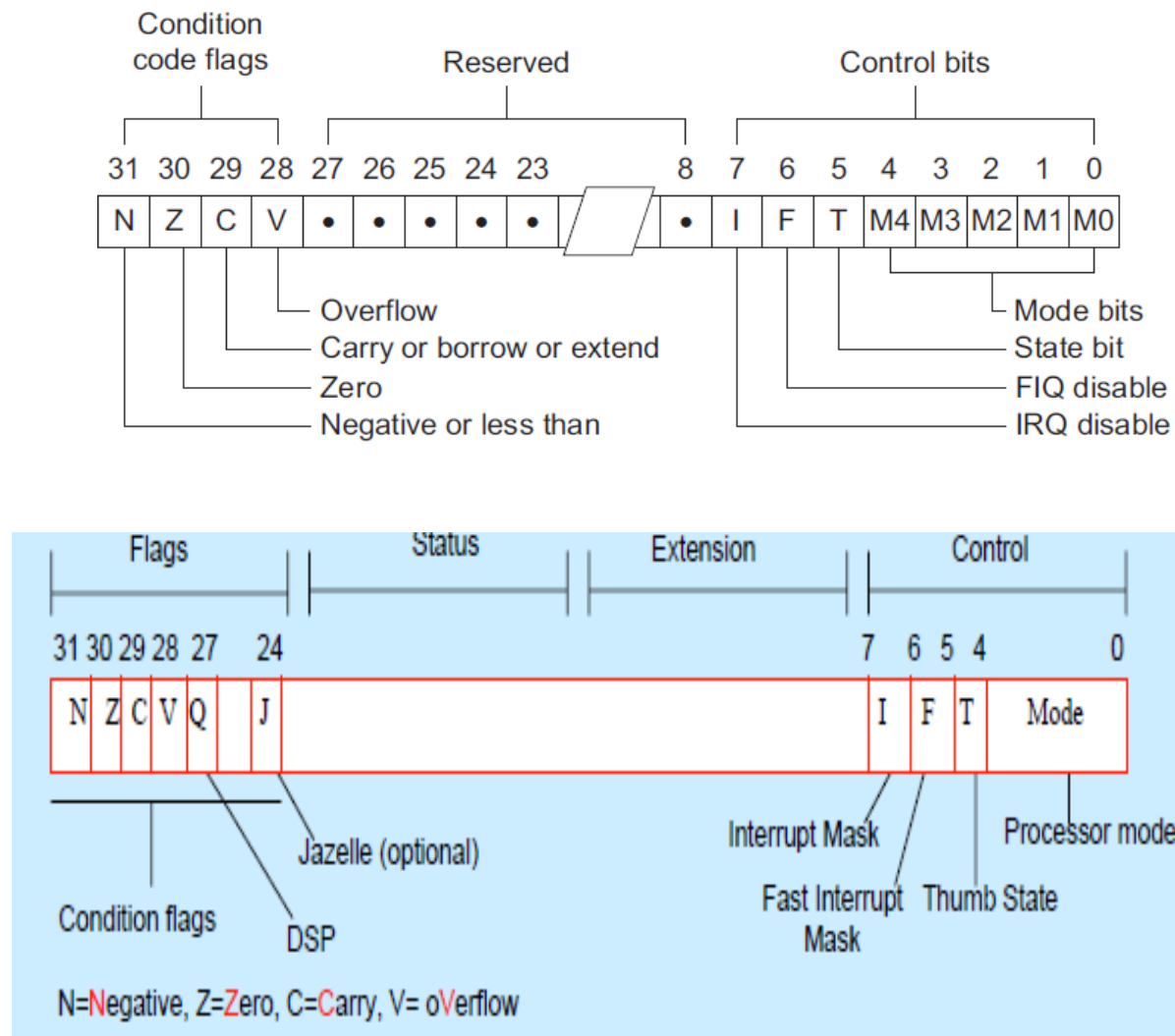
Registers r0–r7 are known as the low registers. Registers r8–r15 are known as the high registers.

## 7.3 The program status registers

program status registers:

- hold the condition code flags
- control the enabling and disabling of interrupts
- set the processor operating mode.

The arrangement of bits is shown in Figure



**Figure 1.7 program status registers**

## **7.4 The condition code flags**

The N, Z, C, and V bits are the condition code flags, You can set these bits by arithmetic and logical operations. The flags can also be set by MSR and LDM instructions.

The ARM7TDMI-S tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally in ARM state. In Thumb state, only the Branch instruction can be executed conditionally

## **7.5 The control bits**

The bottom eight bits of a PSR are known collectively as the control bits. They are the:

- Interrupt disable bits
- T bit
- Mode bits.

The control bits change when an exception occurs. When the processor is operating in a privileged mode, software can manipulate these bits.

## **7.6 Interrupt disable bits**

The I and F bits are the interrupt disable bits:

- when the I bit is set, IRQ interrupts are disabled
- when the F bit is set, FIQ interrupts are disabled.

## **7.7 T bit**

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor is executing in ARM state. The operating state is reflected by the **CPTBIT** external signal.

### **7.8 Mode bits**

The M4, M3, M2, M1, and M0 bits (M[4:0]) are the mode bits. These bits determine the processor operating mode. Not all combinations of the mode bits define a valid processor mode, so take care to use only the bit combinations shown

### **7.9 Reserved bits**

The remaining bits in the PSRs are unused but are reserved. When changing a PSR flag or control bits make sure that these reserved bits are not altered. Also, make sure that your program does not rely on reserved bits containing specific values because future processors might have these bits set to one or zero

The ARM7TDMI-S is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family offers high performance for very low power consumption and gate count. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles. The RISC instruction set, and related decode mechanism are much simpler than those of Complex Instruction Set Computer (CISC) designs. This simplicity gives:

- a high

Instruction throughput

- an excellent real-time interrupt response
- a small, cost-effective, processor macrocell.

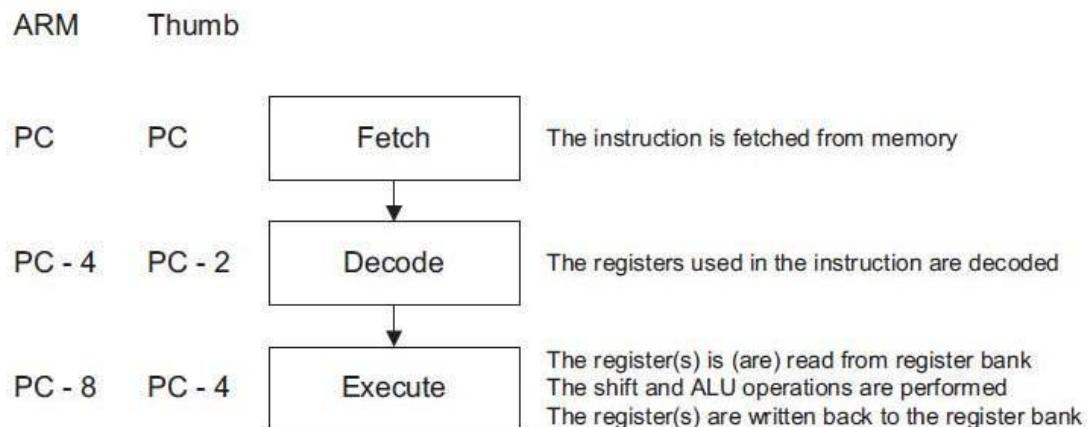
## 8.The instruction pipeline

The ARM7TDMI-S uses a pipeline to increase the speed of the flow of instructions to the processor. This allows several operations to take place simultaneously, and the processing, and memory systems to operate continuously.

A three-stage pipeline is used, so instructions are executed in three stages:

- Fetch
- Decode
- Execute.

The three-stage pipeline is shown in Figure .



**8.1 The Program Counter (PC)** points to the instruction being fetched rather than to the instruction being executed.

During normal operation, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory

## 9. The ARM Processor Families (I)

The ARM7 Family

- ☐ 32-bit RISC Processor. Support three-stage pipeline
- ☐ Uses Von Neumann Architecture.

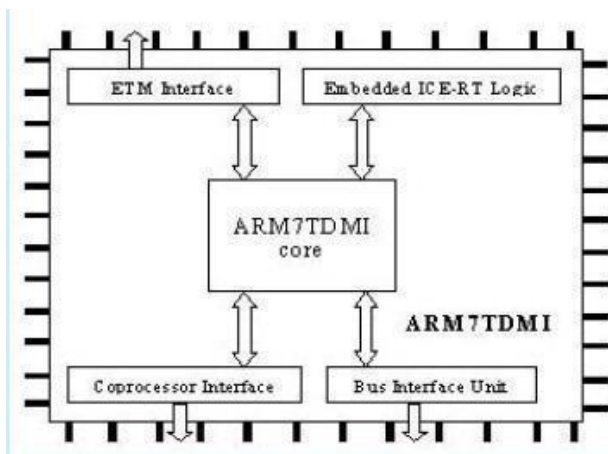


Figure 1.8 ARM7TDMI

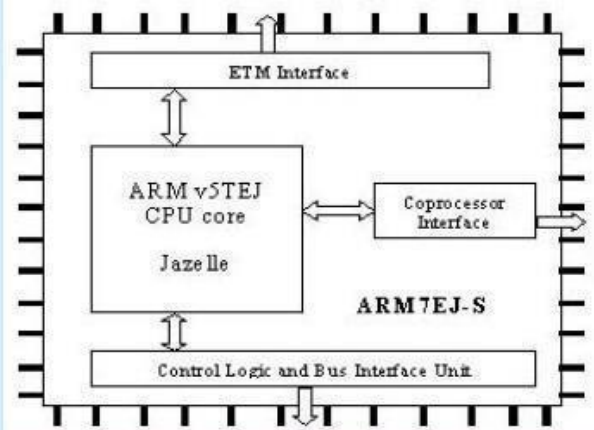


Figure 1.9 ARM7EJ-S

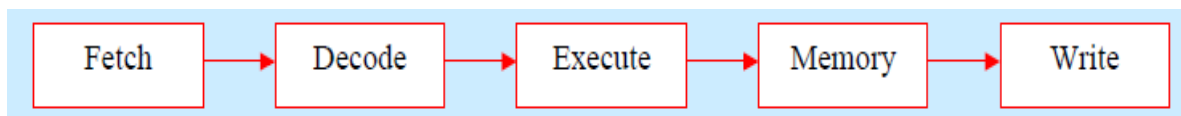
	Cache size(Inst/ Data)	Tightly Coupled Memory	Memory Mgmt	Thumb	DSP	Jazelle
ARM720T	8k unified	-	MMU	Yes	No	No
ARM7EJ-S	-	-	-	Yes	Yes	Yes
ARM7TDMI	-	-	-	Yes	No	No
ARM7TDMI-S	-	-	-	Yes	No	No

Widely used in many applications such as palmtop computers, portable instruments, smart card.

## 9.1 The ARM Processor Families (II)

### The ARM9 Family

- ☐ 32-bit RISC Processor with ARM and Thumb instruction sets Supports five-stage pipeline
- ☐ Uses harvard architecture



Uses Harvard architecture

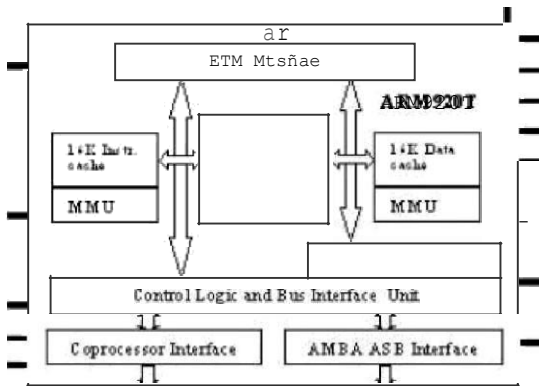


Figure 1.10 ARM920T Processor

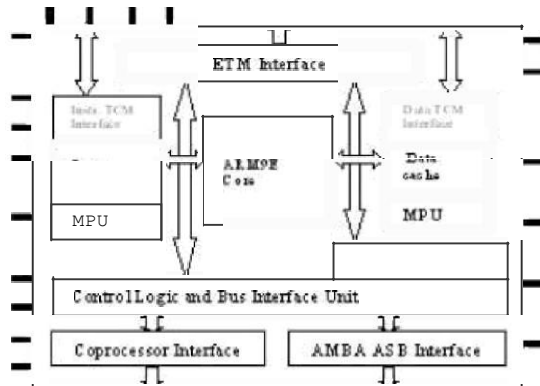


Figure 1.11 ARM946E-S Processor

#### Characteristics of ARM9 Thumb Family

	Cache size(Inst/Data)	Tightly Coupled Memory	Memory Mgmt	Thumb	DSP	Jazelle
ARM920T	64K/64K		MMU	Yes	No	No
ARM922T	8K/8K		MMU	Yes	No	No

#### Characteristics of ARM9E Family

	Cache size(Inst/Data)	Tightly Coupled Memory	Memory Mgmt	Thumb	DSP	Jazelle
ARM926EJ-S	Variable	Yes	MMU	Yes	Yes	Yes
ARM946E-S	Variable	Yes	MPU	Yes	Yes	No
ARM966E-S		Yes		Yes	Yes	No
ARM968E-S	No	Yes	DMA	Yes	Yes	No
ARM968H-S			MPU	Yes	Yes	No



Widely used in mobile phones, PDAs, digital cameras, automotive systems, industrial control systems.

## 9.2 ARM Processor Families (III)

### The ARM10 Family

- ☐ 32-bit RISC processor with ARM, Thumb and DSP instruction sets.
- ☐ Supports six-stage Pipelines Uses
- ☐ Harvard Architecture

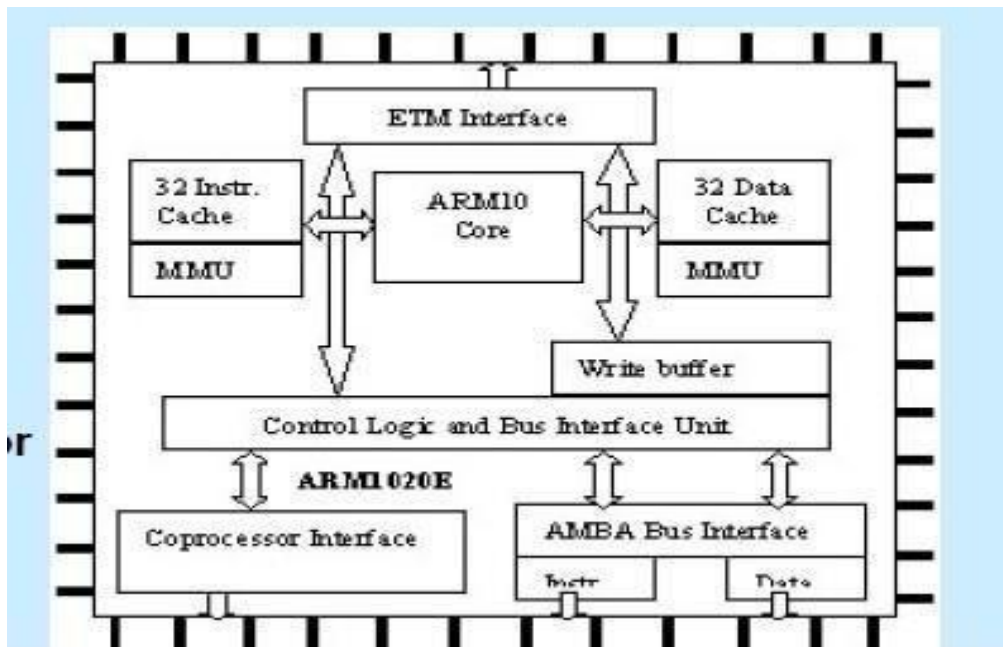


Figure 1.11 ARM1020E Processor

## Characteristics of ARM10 family

	Cache size(Inst /Data)	Tightly Coupled Memory	Memory Mgmt	Thumb	DSP	Jazelle
ARM1020E	32k/32k	-	MMU	Yes	Yes	No
ARM1022E	16k/16k	-	MMU	Yes	Yes	No
ARM1026EJ-S	Variable	Yes	MMU or MPU	Yes	Yes	Yes

Widely used in videophone, PDAs, set-top boxes, game console, digital video cameras, automotive and industrial control systems.

### 9.4 ARM PROCESSOR FAMILIES (IV)

The ARM11 Family

- ☐ 32-bit RISC processor with ARM, Thumb and DSP instruction sets.
- ☐ Uses Harvard Architecture.
- ☐ Supports eight-stage Pipelines except ARM1156T2 uses nine-stage pipeline.
- ☐ Widely used in automotive and industrial control systems, 3D graphics, security critical applications

	Cache size(Inst /Data)	Tightly Coupled Memory	Memory Mgmt	Thumb	DSP	Jazelle
ARM11 MPCore	Variable	Yes	MMU+cache	Yes	Yes	Yes
ARM1136J(F)-S	Variable	Yes	MMU	Yes	Yes	Yes
ARM1156T2(F)-S	Variable	Yes	MPU	Yes	Yes	No
ARM1176JZ(F)-S	Variable	Yes	MMU+TrustZone	Yes	Yes	Yes

## 10. Characteristics of ARM11 family

### 10.1 ARM Pipelines

- Pipeline mechanism to increase execution speed
- The pipeline design of each processor family is different

### 10.2 ARM Processor Modes

Unprivileged mode

User mode Privileged mode Abort mode

Fast Interrupt Request mode Interrupt

Request mode Supervisor mode

System mode

Undefined mode

### 10.3 Exceptions

Exceptions are taken whenever the normal flow of a program must temporarily halt, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the processor preserves the critical parts of the current processor state so that the original program can resume when the handler routine has finished

### 10.4 Exceptions and Interrupts

The ARM processor can work in one of many operating modes. So far we have only considered user mode, which is the "normal" mode of operation.

The processor can also enter "privileged" operating modes which are used to handle exceptions and SWIs

The Current Processor Status Register CPSR has 5 bits [bit4:0] to indicate which mode the processor is in:-

<b>CPSR[4:0]</b>	<b>Mode</b>	<b>Use</b>	<b>Registers</b>
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

## 10.5 How are exceptions generated

By default, the processor is usually in user mode

- ☐ It enters one of the exception modes when unexpected events occur.
- ☐ There are three different types of exceptions (some are called interrupts):-
  - As a direct result of executing an instruction, such as: Software Interrupt
- ☐ Instruction (SWI)
- ☐ Undefined or illegal instruction
- ☐ Memory error during fetching an instruction
  - As a side-effect of an instruction, such as: Memory fault during data read/write from memory
- ☐ Arithmetic error (e.g. divide by zero)
  - As a result of external hardware signals, such as: Reset
- ☐ Fast Interrupt (FIQ)
- ☐ Normal Interrupt (IRQ)
- ☐

## 10.6 Shadow Registers

As the processor enters an exception mode, some new registers are automatically switched in:-

For example, an external event (such as movement of the mouse) occurs that generates a Fast Interrupt (on the FIQ pin), the processor enters FIQ operating mode. It sees the same r0 -

r7 as before, but sees a new set of r8 - r14, and in addition, an extra register called the Saved Processor Status Register (SPSR) stores the value of the CPSR. By swapping to some new registers, it makes it easier for the programmer to preserve the state of the processor.

For example, during FIQ mode, r8 - r14 can be used freely. On returning back to user mode, the original values of r8 - r14 will be automatically restored.

### **10.7 What happens when an exception occurs**

ARM completes current instruction as best it can. It departs from current instruction sequence to handle the exception by performing the following steps:-

1. It changes the operating mode corresponding to the particular exception.
2. It saves the current PC in the r14 corresponding to the new mode. For example, if FIQ occurs, the PC value is stored in r14(FIQ).
3. It saves the old value of CPSR in the Saved Processor Status Register of the new mode.
4. It disables exceptions of lower priority (to be considered later).
5. It forces the PC to a new value corresponding to the exception. This is effectively a forced jump to the Exception Handler or Interrupt Service Routine.

## 10.8 Where is the exception handler routine

Exceptions can be viewed as "forced" subroutine calls. When and if an exception occurs is not predictable (unless it is a SWI exception). A unique address is pre-defined for each exception handler (IRQ, FIQ, etc), and a branch is made to this address. The address to which the processor is forced to branch to is called the exception/interrupt vector.

## 10.9 Exception vector addresses

Each vector (except FIQ) is 4 bytes long (i.e. one instruction) You put a branch instruction at this address: B exception handler FIQ is special in two ways:-

1. You can put the actual FIQ handler (also called Fast Interrupt Service Routine) at 0x0000001C onwards, because FIQ vector occupies the highest address
2. FIQ has many more shadow registers. So you don't have to save as many registers on the stack as other exceptions -faster.

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

## 10.10Exception Return

Once the exception has been handled (by the exception handler),

the user task is resumed.

The handler program (or Interrupt Service Routine) must restore the user state exactly as it was before the exception occurred:

1. Any modified user registers must be restored from the handlers stack
2. The CPSR must be restored from the appropriate SPSR
3. PC must be changed back to the instruction address in the user instruction stream

Steps 1 and 3 are done by user, step 2 by the processor. Restoring registers from the stack would be the same as in the case of subroutine. Restoring PC value is more complicated. The exact way to do it

depends on which exception you are returning from.

### **10.11 Exception Return**

Once the exception has been handled (by the exception handler), the user task is resumed. The handler program (or Interrupt Service Routine) must restore the user state exactly as it was before the exception occurred

1. Any modified user registers must be restored from the handler's stack
2. The CPSR must be restored from the appropriate SPSR



PC must be changed back to the instruction address in the user instruction stream Steps 1 and 3 are done by user, step 2 by the processor

Restoring registers from the stack would be the same as in the case of subroutines Restoring PC value is more complicated. The exact way to do it depends on which exception you are returning from.

Remember that the return address was saved in r14 before entering the exception handler.

To return from a SWI or undefined instruction trap, use: `MOVS pc, r14`

To return from an IRQ, FIQ or prefetch abort, use: `SUBS pc, r14, #4`

To return from a data abort to retry the data access, use: `SUBS pc, r14, #8` If the destination register is the PC, the „S” modifier does NOT mean “set the flags”, but “restore the CPSR”

The differences between these three methods of return is due to the pipeline architecture of the ARM processor. The PC value stored in r14 can be one or two instructions ahead due to the instruction prefetch pipeline.

## **10.12 Exception Priorities**

Since exceptions can arise at the same time, a priority order has to be clearly defined. For the ARM processor this is:

Reset (highest priority)

Data abort (i.e. Memory fault in read/write data) Fast Interrupt

Request (FIQ)

Normal Interrupt Request (IRQ) Prefetch

abort

## **10.13 Software Interrupt (SWI), undefined instruction**

Consider the case of a FIQ and an IRQ occurring at the same time. The processor will process the FIQ handler first and “remember” that there is IRQ pending. On return from FIQ, the process will immediately go to the IRQ handler.

## 11. Interrupts

The processor has two interrupt inputs, for normal interrupts (**nIRQ**) and fast interrupts (**nFIQ**). Each interrupt pin, when asserted and not masked, causes the processor to take the appropriate type of interrupt exception..The CPSR.F and CPSR.I bits control masking of fast and normal interrupts respectively.

A number of features exist to improve the interrupt latency, that is, the time taken between the assertion of the interrupt input and the execution of the interrupt handler. By default, the processor uses the *Low Interrupt Latency* (LIL) behaviors introduced in version 6

and later of the ARM architecture. The processor also has a port for connection of a *Vectored Interrupt Controller* (VIC), and supports *Non-Maskable Fast Interrupts* (NMFI).

The following subsections describe interrupts:

- Interrupt request
- Fast interrupt request
- Non-maskable fast interrupts
- Low interrupt latency
- Interrupt controller.

### 11.1 Interrupt request

The IRQ exception is a normal interrupt caused by a LOW level on the **nIRQ** input. An IRQ has a lower priority than an FIQ, and is masked on entry to an FIQ sequence. You must ensure that the **nIRQ** input is held LOW until the processor acknowledges the interrupt request, either from the VIC interface or the software handler.

Irrespective of whether the exception is taken from ARM state or Thumb state, an IRQ handler returns from the interrupt by executing:

```
SUBS PC, R14_irq, #4
```

You can disable IRQ exceptions within a Privileged mode by setting the CPSR.I bit to b1. See *Program status registers*. IRQ interrupts are automatically disabled when an IRQ occurs, by setting the CPSR.I bit. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable IRQs by clearing the CPSR.I bit.

## 11.2 Fast interrupt request

The *Fast Interrupt Request* (FIQ) reduces the execution time of the exception handler relative to a normal interrupt. FIQ mode has eight private registers to reduce, or even remove the requirement for register saving (minimizing the overhead of context switching).

An FIQ is externally generated by taking the nFIQ input signal LOW. You must ensure that the nFIQ input is held LOW until the processor acknowledges the interrupt request from the software handler.

Irrespective of whether exception entry is from ARM state or Thumb state, an FIQ handler returns from the interrupt by executing:

```
SUBS PC, R14_fiq, #4
```

If Non-Maskable Fast Interrupts (NMFI) are not enabled, you can mask FIQ exceptions by setting the CPSR.F bit to b1. For more information see:

- *Program status registers*
- *Non-maskable fast interrupts.*

FIQ and IRQ interrupts are automatically masked by setting the CPSR.F and CPSR.I bits when an FIQ occurs. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable interrupts.

### **11.3 Non-maskable fast interrupts**

When NMFI behavior is enabled, FIQ interrupts cannot be masked by software. Enabling NMFI behavior ensures that when the FIQ mask, that is, the CPSR.F bit, is cleared by the reset handler, fast interrupts are always taken as quickly as possible, except during handling of a fast interrupt. This makes the fast interrupt suitable for signaling critical events. NMFI behavior is controlled by a configuration input signal CFGNMFI, that is asserted HIGH to enable NMFI operation. There is no software control of NMFI.

Software can detect whether NMFI operation is enabled by reading the NMFI bit of the SCTLR:

## **NMFI == 0**

Software can mask FIQs by setting the CPSR.F bit to b1.

## **NMFI == 1**

Software cannot mask FIQs.

For more information see *c1, System Control Register*. When the NMFI bit in

the SCTLR is b1:

- an instruction writing b0 to the CPSR.F bit clears it to b0
- an instruction writing b1 to the CPSR.F bit leaves it unchanged
- the CPSR.F bit can be set to b1 only by an FIQ or reset exception entry.

### **11.4 Low interrupt latency**

*Low Interrupt Latency (LIL)* is a set of behaviors that reduce the interrupt latency for the processor, and is enabled by default. That is, the FI bit [21] in the SCTLR is Read-as-One.

LIL behavior enables accesses to Normal memory, including multiword accesses and external accesses, to be abandoned part-way through execution so that the processor can react to a pending interrupt faster than would otherwise be the case. When an instruction is abandoned in this way, the processor behaves as

if the instruction was not executed at all. If, after handling the interrupt, the interrupt handler returns to the program in the normal way using instruction SUBS pc, r14, #4, the abandoned instruction is re-executed. This means that some of the memory accesses generated by the instruction are performed twice.

Memory that is marked as Strongly-ordered or Device type is typically sensitive to the number of reads or writes performed. Because of this, instructions that access Strongly-ordered or Device memory are never abandoned when they have started accessing memory. These instructions always complete either all or none of their memory accesses. Therefore, to minimize the interrupt latency, you must avoid the use of multiword load/store instructions to memory locations that are marked as Strongly- ordered or Device.

## **11. Interrupt controller**

The processor includes a VIC port for connection of a *Vectored Interrupt Controller* (VIC). An interrupt controller is a peripheral that handles multiple interrupt sources. Features usually found in an interrupt controller are:

- Multiple interrupt request inputs, one for each interrupt source, and one or more amalgamated interrupt request outputs to the processor
- The ability to mask out particular interrupt requests
- Prioritization of interrupt sources for interrupt nesting.

In a system with an interrupt controller with these features, software is still required to:

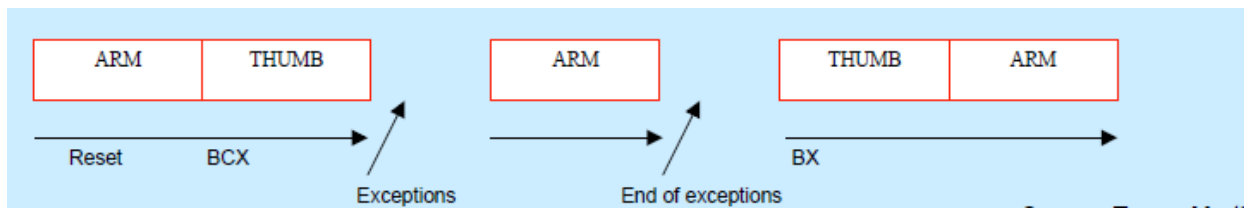
- Determine from the interrupt controller which interrupt source is requesting service
- Determine where the service routine for that interrupt source is loaded
- Mask or clear that interrupt source, before re-enabling processor interrupts to permit another interrupt to be taken.

A VIC does all these in hardware to reduce the interrupt latency. It supplies the starting address of the service routine corresponding to the highest priority asserted interrupt source directly to the processor. When the processor has accepted this address, it masks the interrupt so that the processor can re-enable interrupts without clearing the source. The PL192 VIC is an AMBA compliant, SoC peripheral that is developed, tested, and licensed by ARM.



You can use the VIC port to connect a PL192 VIC to the processor. See the *ARM PrimeCell Vectored Interrupt Controller (PL192) Technical Reference Manual* for more information about the PL192 VIC. You can enable the VIC port by setting the VE bit in the SCTLR. When the VIC port is enabled and an IRQ occurs, the processor performs an handshake over the VIC interface to obtain the address of the handling routine for the IRQ.

### 11.6 Exception when processor in the Thumb mode



#### Vector table

The vector table All ARM systems have a vector table. The vector table does not form part of the initialization sequence, but it must be present for any exception to be serviced. It must be placed at a specific address, usually 0x0.

### **Questions Bank:**

- 1.What are the main features of ARM architecture
2. Draw and explain the ARM family Architecture.
- 3.Compare ARM7, ARM9 and ARM11 series processors stating features
4. Explain the term Banked Register in ARM.
5. What is the significance of special purpose registers R13, R14 and R15.
6. Explain ARM7 Programmers model.
7. Explain terms: CPSR register and Processor modes.
8. What is the function of barrel shifter in ARM data flow model
9. Explain the architecture of ARM
10. Explain the three stage pipelining implemented in ARM processor

### **TEXT / REFERENCE BOOKS**

1. Andrew N.Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guide: Designing & Optimizing System Software, Elsevier, 2004.
2. Jonathan W.Valvano, Embedded Microcomputer Systems: Real Time Interfacing, Cengage Learning, 2011
3. Wayne Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2008.
4. C.M.Krishna, Kang G.Shin, Real time systems, McGraw Hill, 3rd reprint, 2010.
5. Herma K., Real Time Systems: Design for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
6. William Hohl, ARM Assembly Language, Fundamentals and Techniques, Taylor & Francis, 2009.



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY  
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS & INSTRUMENTATION**

## **UNIT-II**

**EMBEDDED SYSTEM DESIGN— SBMA5201**

## 1. Arm Instruction Set

Data Processing Instructions, Addressing Modes, Branch, Load, Store Instructions, PSR Instructions, Conditional Instructions. Thumb Instruction Set: Register Usage, Other Branch Instructions, Data Processing Instructions, Single Register and Multi Register Load -Store Instructions, Stack, Software Interrupt Instructions

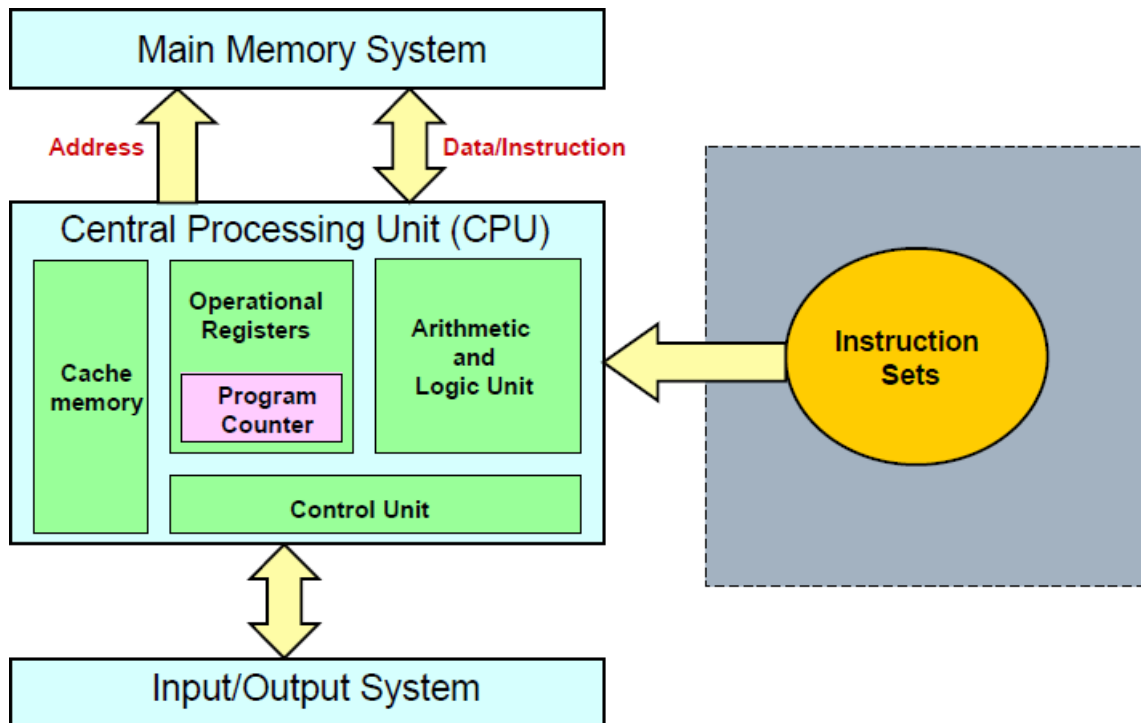


Figure 2.1 Arm Architecture

### **1.1.ARM has 37 registers all of which are 32-bits long.**

- 1 dedicated program counter□
- 1 dedicated current program status register
- 5 dedicated saved program status registers 30 general purpose registers

The current processor mode governs which of several banks is accessible. Each mode can access a particular set of r0-r12 registers a particular r13 (the stack pointer, sp) and r14 (the link register, lr)□ the program counter, r15 (pc)□ the current program status register, cpsr

### **1.2 Privileged modes (except System) can also access**

a particular spsr (saved program status register)

- ARM processor was designed by Advanced RISC Machine (ARM) Limited Company
- ARM processors are major used for low-power and low cost applications
- Mobile phones
- Communication modems
- Automotive engine management systems
- Hand-held digital systems

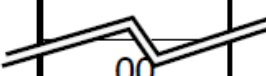
Different versions of ARM processors share the same basic machine instruction sets

## 2.ARM programmer model

- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general purpose registers (R0-R14), program counter (PC) and CPSR.
- Instruction set defines the operations that can change the state.

### Memory system Byte ordering

- Memory is a linear array of bytes addressed from 0 to  $2^{32}-1$
- Word, half-word, byte
- Little-endian

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
	
0xFFFFFFFFD	00
0xFFFFFFFEE	00
0xFFFFFFFFF	00

- **Big Endian**

- Least significant byte has highest address

Word address 0x00000000

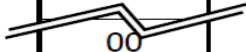
Value: 00102030

- **Little Endian**

- Least significant byte has lowest address

Word address 0x00000000


Value: 30201000

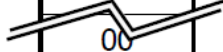
0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
	
0xFFFFFFFFD	00
0xFFFFFFFFE	00
0xFFFFFFFFF	00

## Byte ordering

## ARM programmer model

R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
R12	R13	R14	PC

31	30	29	28	27	26			8	7	6	5	4	3	2	1	0
N	Z	C	V	Q				I	F	T	M	M	M	M	M	M
											4	3	2	1	0	

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
	
0xFFFFFFFFD	00
0xFFFFFFFFE	00
0xFFFFFFFFF	00

ARM instructions are all 32-bit long (except for Thumb mode). There are  $2^{32}$  possible machine instructions. Fortunately, they are structured.

[illegible]

### 3.Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple registers at once
- Possible to combine shift and ALU operations in a single instruction

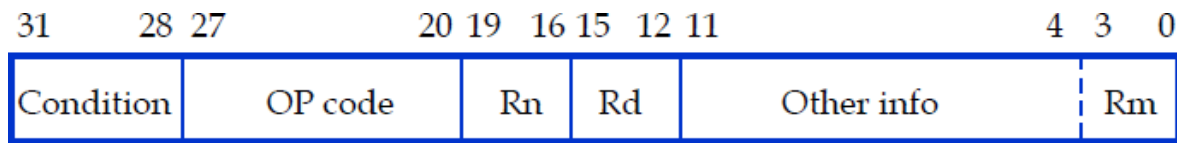
## 4. Registers and Memory Access



- In the ARM architecture
- Memory is byte addressable
- 32-bit addresses
- 32-bit processor registers
  - Two operand lengths are used in moving data between the memory and the processor registers
    - Bytes (8 bits) and words (32 bits)
  - Word addresses must be aligned, i.e., they must be multiple of 4
- Both little-endian and big-endian memory addressing are supported
  - When a byte is loaded from memory into a processor register or stored from a register into the memory
- It always located in the low-order byte position of the register

## **5.ARM Instruction Format**

- Each instruction is encoded into a 32-bit word
- Access to memory is provided only by Load and Store instructions
- The basic encoding format for the instructions, such as Load, Store, Move, Arithmetic, and Logic instructions, is shown below



An instruction specifies a conditional execution code (Condition), the OP code, two or three registers (Rn, Rd, and Rm), and some other information

### Conditional Execution of Instructions

A distinctive and somewhat unusual feature of ARM processors is that all instructions are conditionally executed

- Depending on a condition specified in the instruction
  - The instruction is executed only if the current state of the processor condition code flag satisfies the condition specified in bits b31-b28 of the instruction
- Thus the instructions whose condition is not met the processor condition code flag are not executed
  - One of the conditions is used to indicate that the instruction is always executed

## 6. Instruction Set of Arm Processor

- ARM Instruction Set:
  - standard 32-bit Instruction set
- Thumb Instruction Set: 16-bit instruction set
- Jazelle Instruction Set: 8-bit instruction set

ARM Instruction Set supports six different types of instructions Data Processing

Instructions

Branch Instructions Load/Store Instructions

Software Interrupt Instruction

Program Status Register Instructions Coprocessor Instructions

## **6.1 Data Processing Instructions**

The data processing instructions operate on data held in general purpose registers. Of the two source operands, one is always a register.

The other has two basic forms:

- An immediate value
- A register value optionally shifted.

If the operand is a shifted register the shift amount might have an immediate value or the value of another register.

Four types of shift can be specified. Most data processing instructions can perform a shift followed by a logical or arithmetic operation.

Multiply instructions come in two classes:

- normal - 32-bit result
- long - 32-bit result variants.

Both types of multiply instruction can optionally perform an accumulate operation.

Used to manipulate data in general-purpose registers, employ a 3-address format, support barrel shifter

## 6.2 Arithmetic Instructions

ADD, ADC, SUB, SBC, RSB, RSC

Move Instructions MOV,

MVN

Bit-Wise Logical Instructions AND, EOR,

ORR, BIC

Comparison Instructions TST, TEQ,

CMP, CMN

Multiply Instructions: MUL,

MLA

### 6.3 Addressing modes Load /store instructions

have three primary addressing modes

- offset
- pre-indexed
- post-indexed.

offset :They are formed by adding or subtracting an immediate or register-based offset to or from a base register

Register-based offsets can also be scaled with shift operations.

- Preindex
  - Memory address is formed as for offset addressing
  - Memory address also written back to base register
  - So base register value incremented or decremented by offset value
- Postindex
  - Memory address is base register value
  - Offset added or subtracted Result written back to base register

- Base register acts as index register for pre index and post index addressing

Pre-indexed and post-indexed addressing modes update the base register with the base plus offset calculation.

- Offset either immediate value in instruction or another register
- If register scaled register addressing available
  - Offset register value scaled by shift operator
  - Instruction specifies shift size

As the PC is a general purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4GB memory space.

## **Branch Instructions**

Change the flow of sequential execution of instructions and force to modify the program counter

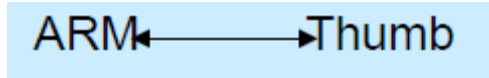
Branch (B) jumps in a range of +/-32 MB.

Branch with link(BL) suitable for subroutine call by storing the address of next instructions after BL into the link register and

restore the program counter from the link register while returning from subroutine.

Branch Exchange and Branch Exchange Link  
processor state from Thumb to ARM and vice versa

Link for switching the



## 6.4 Load/Store Instructions

The second class of instruction is load and store instructions. These instructions come in two main types

- load or store the value of a single register or register pair
- load and store multiple register values.

Load and store single register instructions can transfer a 32-bit word, a 16-bit half word and an eight-bit byte between memory and a register.

Byte and half word loads may be automatically zero extended or sign extended as they are loaded.

A preload „hint“ instruction is available to help minimize memory system latency.

Swap instructions perform an atomic load and store as synchronization primitive

Transfer data between memory and registers

## 6.5 Single Register Transfer Instructions

- Used to move a single data item in and out of register (signed, unsigned, 16-bit half words and 32-bit word)
- ☐ Supports register indirect, base-plus-offset and stack addressing mode LDR, STR, LDRB, STRB, LDRH, STRH, LDRSB

## 6.6 Multiple Register Transfer Instructions

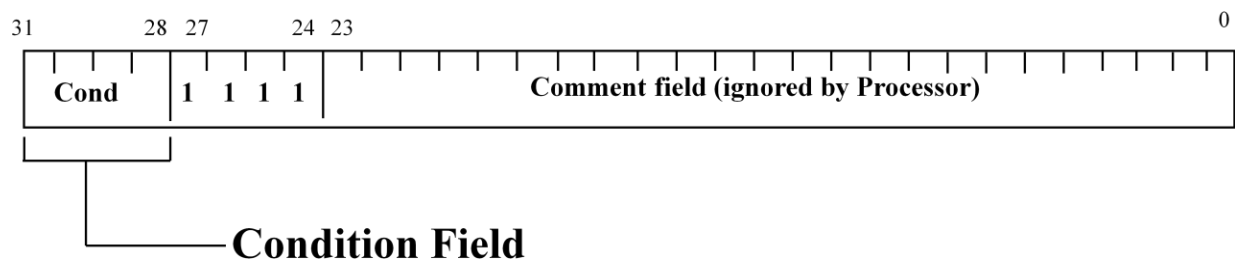
Any subset or all the 16 registers loaded from or stored to memory but increase interrupt latency.

Addressing modes-IA, IB, DA, DB

Stack operations-FA, FD, EA, ED LDM, STM

Swap Instructions :swap the content of memory with the content of registers.SWP, SWPB

## Software Interrupt Instruction



\* In effect, a SWI is a user-defined instruction.



- \* It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- \* The handler can then examine the comment field of the instruction to decide what operation has been requested.
- \* By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.

### Software Interrupt Instruction

1 1 0 1 1 1 1 1	8 – Bit Immediate
-----------------	-------------------

- Address of next instruction is saved in r14\_svc
- CPSR is saved in r14\_svc
- Disables IRQ, Clears T bit, Enters Supervisor mode
- PC is forced to 0x08
- 8 bit immediate is zero extended to fill the 24-bit field in the ARM instruction.

Limits SWIs to first 256 of 16 million ARM SWIs

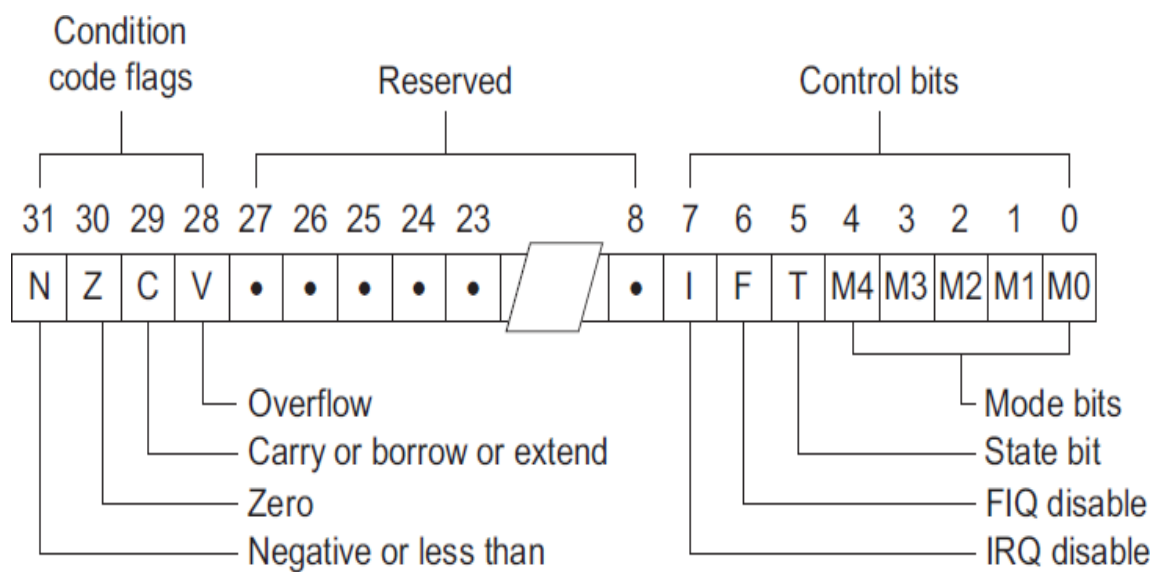
## 6.7 Program Status Register Instructions

- Used to transfer the content of program status registers to/from a general-purpose register
- MRS (copy program status register to a general purpose register), MSR(move a general-purpose register to a program status register)

## 6. Program status registers

The ARM7TDMI-S contains a CPSR and five SPSRs for exception handlers to use. The program status registers:

- Hold the condition code flags
- Control the enabling and disabling of interrupts
- Set the processor operating mode.



The arrangement of bits is shown in Figure

## 7.1 condition code flags

The N, Z, C, and V bits are the condition code flags, You can set these bits by arithmetic and logical operations. The flags can also be set by MSR and LDM instructions. The ARM7TDMI-S tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally in ARM state. In Thumb state, only the Branch instruction can be executed conditionally

## 7.2 control bits

The bottom eight bits of a PSR are known collectively as the *control bits*. They are the:

- *Interrupt disable bits*
- *T bit*
- *Mode bits.*

The control bits change when an exception occurs. When the processor is operating in a privileged mode, software can manipulate these bits.

## 7.3 Interrupt disable bits

The I and F bits are the interrupt disable bits:

- when the I bit is set, IRQ interrupts are disabled
- when the F bit is set, FIQ interrupts are disabled.

### **T bit**

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor executing in ARM state. The operating state is reflected by the **CPTBIT** external signal.

### **Mode bits**

The M4, M3, M2, M1, and M0 bits (M[4:0]) are the mode bits. These bits determine the processor operating mode as listed in Table 2-2. Not all combinations of the mode bits define a valid processor mode, so take care to use only the bit combinations shown

### **Reserved bits**

The remaining bits in the PSRs are unused but are reserved. When changing a PSR flag or control bits make sure that these reserved bits are not altered. Also, make sure that your program does not rely on reserved bits containing specific values because future processors might have these bits set to one or zero

### **Coprocessor Instructions**

Used to extend the instruction set, to control on-chips functions (caches and memory management) and for additional computations.

- CDP(data processing), MRC/MCR (register transfer), LDC/STC (memory transfer).

## **7.4 Program status registers**

The processor contains one CPSR and five SPSRs for exception handlers to use.

The program status registers:

- hold information about the most recently performed ALU operation
- control the enabling and disabling of interrupts
- set the processor operating mode

## **8.Addressing modes**

Load and store instructions have three primary addressing modes

- offset
- pre-indexed
- post-indexed.

They are formed by adding or subtracting an immediate or register-based offset to or from a base register. Register-based offsets can also be scaled with shift operations. Pre-indexed and post-indexed addressing modes update the base register with the

base plus offset calculation. As the PC is a general purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4GB memory space

## 8.1 Conditional instructions

ARM and Thumb instructions can execute conditionally on the condition flags set by a previous instruction.

The conditional instruction can occur either:

- ☐ Immediately after the instruction that updated the flags.
- ☐ After any number of intervening instructions that have not updated the flags.

The instructions that you can make conditional depends on whether the processor is in ARM state or Thumb state.

To make an instruction conditional, you must add a condition code suffix to the instruction mnemonic. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- ☐ Does not execute.
- ☐ Does not write any value to its destination register.
- ☐ Does not affect any of the flags.
- ☐ Does not generate any exception.

Conditional execution in ARM state Almost all ARM instructions can be executed conditionally on the value of the ALU status flags in the APSR. You can either add a condition code suffix to the instruction or you can conditionally skip over the instruction using a conditional branch instruction. Using conditional branches instructions to control the flow of execution can be better when a series of instructions depend on the same condition.

## 8.2 Conditional instructions to control execution

ere follows a list of available conditional codes: EQ : Equal

If the Z flag is set after a comparison.

NE : Not Equal

If the Z flag is clear after a comparison.

VS : Overflow Set

If the V flag is set after an arithmetical operation, the result of which will not fit into a 32bit destination register.

VC : Overflow Clear

If the V flag is clear, the reverse of VS. HI : Higher

Than (unsigned)

If after a comparison the C flag is set **AND** the Z flag is clear.

LS : Lower Than or Same (unsigned)

If after a comparison the C flag is clear **OR** the Z flag is set.

PL : Plus

If the N flag is clear after an arithmetical operation. For the purposes of defining 'plus', zero is positive because it isn't negative...

MI : Minus

If the N flag is set after an arithmetical operation.

CS : Carry Set

Set if the C flag is set after an arithmetical operation OR a shift operation, the result of which cannot be represented in 32bits. You can think of the C flag as the 33rd bit of the result.

CC : Carry Clear

The reverse of CS.

GE : Greater Than or Equal (signed) If after a comparison...

the N flag is set **AND** the V flag is set or...

the N flag is clear **AND** the V flag is clear.

GT : Greater Than (signed) If after a

comparison...

the N flag is set **AND** the V flag is set or...

the N flag is clear **AND** the V flag is clear



*and...*

the Z flag is clear.

LE : Less Than or Equal To (signed) If after a comparison...

the N flag is set **AND** the V flag is clear or...

the N flag is clear **AND** the V flag is set

*and...*

the Z flag is set.

LT : Less Than (signed)

If after a comparison...

the N flag is set **AND** the V flag is clear or...

the N flag is clear **AND** the V flag is set.

AL : Always

The default condition, so does not need to be explicitly stated.

NV : Never Not particularly useful, it states that the instruction should never be executed.

A kind of Poor Mans' NOP.

NV was included for completeness (as the reverse of AL), but you should not use it in your own code.

There is a final conditional code which works in the reverse way. S, when applied to an instruction, causes the status flags to

be updated. This does *not* happen automatically - except for those instructions whose purpose is to set the status. For example:

```
ADD      R0, R0, R1
```

```
ADDS     R0, R0, R1
```

```
ADDEQS R0, R0, R1
```

The first example shows us a basic addition (adding the value of R1 to R0) which does not affect the status registers.

The second example shows us the same addition, only this time it will cause the status registers to be updated.

The last example shows us the addition again, updating the status registers. The difference here is that it is a conditional instruction. It will only be executed if the result of a previous operation was EQual (if the Z flag is set).

Here is an example of conditional execution at work. You want to compare register zero with the contents of something stored in register ten. If not equal to R10, then call a software interrupt, increment and branch back to do it again. Otherwise clear R10 and return to a calling piece of code (whose address is stored in R14)

## An example of conditional execution

```
.loop                                ; Mark the loop start position

CMP      R0, R10                    ; Compare R0 with R10
SWINE    &40017                     ; Not equal: Call SWI &40017
ADDNE    R0, R0, #1                 ;          Add 1 to R0
BNE      loop                       ;          Branch to 'loop'
MOV      R10, #0                    ; Equal   : Set R10 to zero
LDMFD    R13!, {R0-R12,PC}          ;          Return to caller Notes:
```

## 9.The 32 bit PSR

Processors after the ARM 3 provide a 32 bit addressing space by moving the PSR out of R15 and giving R15 a full 32 bits in which to store the address of the current position.

Currently, RISC OS works in 26 bit mode, except for a few special cases which is unlikely to be encountered.

The 32 bit mode is important because 26 bits (as in the old PSR) restricts the maximum amount of addressable memory per-application to 28Mb. That is why you can't drag the Next slot beyond 28Mb irrespective of how much memory you have installed.

The allocation of the bits within the CPSR (and the SPSR registers to which it is saved) is:

31	30	29	28	---	7	6	-	4	3	2	1	0	
N	Z	C	V		I	F		M4	M3	M2	M1	M0	
						0	0	0	0	0			User26 mode
						0	0	0	0	1			FIQ26 mode
						0	0	0	1	0			IRQ26 mode
						0	0	0	1	1			SVC26 mode
						1	0	0	0	0			User mode
						1	0	0	0	1			FIQ mode
						1	0	0	1	0			IRQ mode
						1	0	0	1	1			SVC mode
						1	0	1	1	1			ABT mode
						1	1	0	1	1			UND mode

Typically, the processor will be operating in User26, FIQ26, IRQ26 or SVC26 mode. It is possible to enter a 32 bit mode, but extreme care must be taken. RISC OS won't expect it, and will sulk greatly if it finds itself in it!

(except RISC OS 5 which works totally in 32bit mode - and you cannot enter 26bit as the processor doesn't have that anymore...)

You cannot use MOV<sub>S</sub> PC, R14 or LDMFD R13!, {*registers*, PC}^ in 32 bit code.

Neither can you use ORRS PC, R14,

#1<<28 to set the V flag.

All of this is now possible using MRS and MSR.

Copy a register into the PSR

```
MSR    CPSR, R0                ; Copy R0 into CPSR
MSR    SPSR, R0                ; Copy R0 into SPSR
MSR    CPSR_flg, R0            ; Copy flag bits of R0 into CPSR

MSR    CPSR_flg, #1<<28        ; Copy flag bits (immediate) into
CPSR
```

Copy the PSR into a register

```
MRS    R0, CPSR                ; Copy CPSR into R0
MRS    R0, SPSR                ; Copy SPSR into R0
```

You have two PSRs - CPSR which is the Current Program Status Register and SPSR which is the Saved Program Status Register. Each privileged mode has its own PSR, so the total available selection of PSR is:

- CPSR\_all - current
- SPSR\_svc - saved, SVC(32) mode
- SPSR\_irq - saved, IRQ(32) mode
- SPSR\_abt - saved, ABT(32) mode
- SPSR\_und - saved, UND(32) mode
- SPSR\_fiq - saved, FIQ(32) mode

It appears as if you cannot explicitly specify to save the current PSR in, say, `SPSR_fiq`. Instead, you should change to FIQ mode and then save to SPSR. In other words, you can only alter the SPSR of the mode you are in.

Using the `_flag` suffix allows you to alter the flag bits without affecting the control bits.

In user(32) mode, the control bits of CPSR are protected, you can only alter the condition flags. In other modes, the entire CPSR is available. You should not specify R15 as a source or destination register. And finally, you must not attempt to access the SPSR in user(32) mode as it doesn't exist!

To set the V flag:

```
MSR      CPSR_flag, #&10000000
```

This sets the V flag and doesn't affect the control bits.

Here, for your delectation, is a way to set the V flag on *any* ARM processor:

```
CMP      R0, #1<<31
```

```
CMNVC    R0, #1<<31
```

Clever, huh?

To change mode:

```

MRS      R0, CPSR_all          ; Copy the PSR BIC R0,
R0, #&1F; Clear the mode bits
ORR      R0, R0, #new_mode     ; Set bits for new mode
MSR      CPSR_all, R0          ; write PSR back, changing mode

```

## STACK

The ARM architecture offers extensive support for memory stack by allowing programmers to chose one of four stack format/orientation.

- Empty or Full:
- Empty: Stack Pointer points to the next free space on stack
- Full: Stack Pointer points to the last item on the stack
- Ascending or Descending:
- Ascending: Grows from low memory to high memory
- Descending: Grows from high memory to low memory
- I386, Sparc and PowerPC all use a “Full, Descending” stack format.

We need to store the processor state when making nested calls.



The multiple data transfer instructions provide a mechanism for storing state on the *stack* (pointed to by R13).

The STM and LDM instructions" modes have aliases for accessing stacks:

- FD = Full Descending
  - STMFD/LDMFD = STMDB/LDMIA
- ED = Empty Descending
  - STMED/LDMED = STMDA/LDMIB
- FA = Full Ascending
  - STMFA/LDMFA = STMIB/LMDA
- EA = Empty Ascending
  - STMEA/LDMEA = STMIA/LDMDB Anything

but a full descending stack is rare!

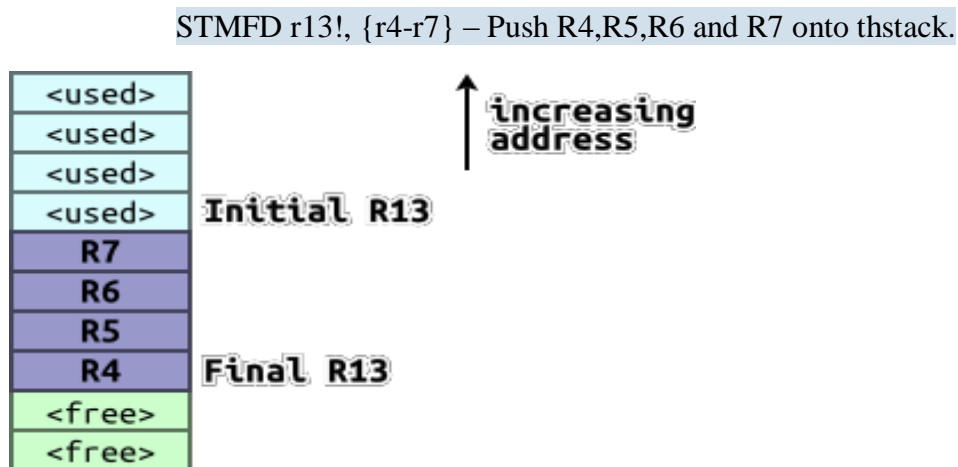


Figure : 2.2 Register File

LDMFD r13!, {r4-r7} – Pop R4,R5,R6 and R7 from the stack

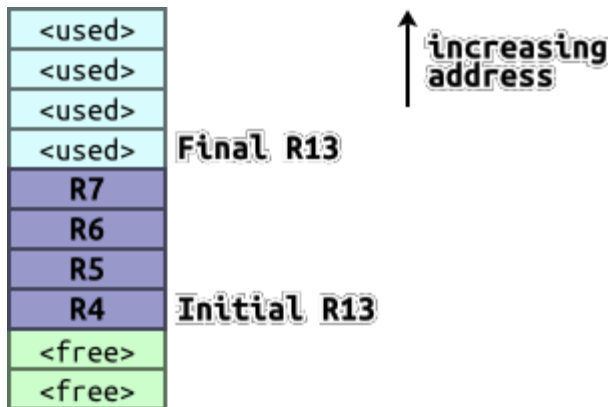


Figure 2.3 Register file

## 9. Thumb Instruction Set

- ARM architecture versions v4T and above define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set.
- A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ARM instructions is operating in *ARM state*.
- A processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM instructions. You must ensure that the processor never

receives instructions of the wrong instruction set for the current state.

- Each instruction set includes instructions to change processor state.
- *Note: ARM processors always start executing code in ARM state.*
- Thumb is a 16-bit instruction set
  - Optimized for code density from C code
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has two execution states – ARM and Thumb
  - Switch between them using **BX** instruction
- Thumb has characteristic features:
  - Most Thumb instructions are executed unconditionally
  - Many Thumb data processing instructions use a 2-address format
  - Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.
- The processor in Thumb mode uses same eight general-purpose integer registers that are available in ARM

mode. Some Thumb instructions also access the PC (ARM register 15), the Link Register (ARM register 14) and Stack Pointer (ARM register 13).

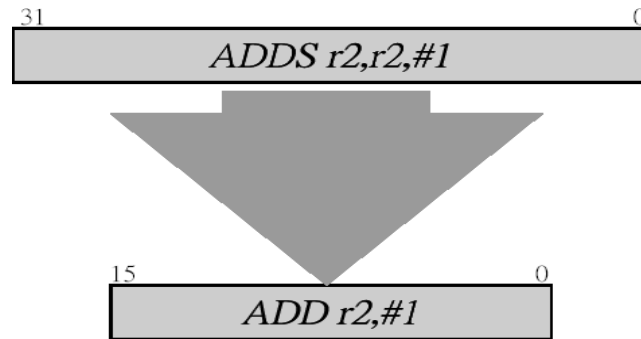
- When R15 is read, bit[0] is zero and bits[31:1] contain the PC. When R15 is written, bit[0] is IGNORED and bits[31:1] are written to the PC.
- Thumb does not provide direct access to the CPSR or any SPSR.
- Thumb execution is flagged by the T bit (bit[5]) in the CPSR. T==0 32-bit instructions are fetched (ARM instruction)  
T==1 16-bit instructions are fetched (Thumb instruction)

## 9.1 Thumb applications

In a typical embedded system:

- use ARM code in 32-bit on-chip memory for small speed-critical routines
- use Thumb code in 16-bit off-chip memory for large non-critical control routines

Note: Switching between ARM and Thumb States of Execution Using BX Instruction



**Figure 2.5 Thumb Instructions**

For Most Instruction Generated by the Compiler

- Condition Execution is not used.
- Source and Destination Registers are identical
- Only low registers used
- Constants are limited size
- Inline barrel shifter not used

### Data Types

Byte (8-bit): placed on any byte boundary.

Half-word (16-bit): aligned to two-byte boundaries. Word (32-bit): aligned to four- byte boundaries

### Thumb Programmers Model

- Registers r0 to r7 are accessible (Lo)
- Few instructions require r8 to r15 to be specified

- r13 is used as the stack pointer
- r14 is used as the link register
- r15 is used as the program counter

## 9.2 THUMB Register Organization

### Thumb General registers and Program Counter

User / System	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
SP	SP_FIQ	SP_SVC	SP_ABT	SP_IRQ	SP_UND
LR	LR_FIQ	LR_SVC	LR_ABT	LR_IRQ	LR_UND
PC	PC_FIQ	PC_SVC	PC_ABT	PC_IRQ	PC_UND

### Thumb Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_FIQ	SPSR_SVC	SPSR_ABT	SPSR_IRQ	SPSR_UND

## **ARM-Thumb differences**

- Unconditional Execution of instruction
- 2-address format for data processing
- Less regular instruction formats.

### **Thumb exception**

- With exception processor is returned to ARM mode.
- While returning previous mode is restored as SPSR is transferred to CPSR

### **Thumb Branching**

- Short conditional branches
- Medium range unconditional branches
- Long range Subroutine calls
- Branch to change to ARM Mode
- **Thumb-ARM Decompression**
- Translation from 16-bit Thumb instruction to 32-bit ARM instruction
- Condition bits changed to „always“
- Lookup to translate major and minor opcodes

- Zero extending 3-bit register specifiers to give 4-bit specifiers
- Zero extending immediate values
- Implicit „S“ (affecting condition codes) should be explicitly specified.
- Thumb 2-address format must be mapped to ARM 3-address format

### **Properties**

- Thumb code requires 70% of space of ARM code
- Thumb code uses 40% more instructions than the ARM code
- With 32-bit memory ARM code is 40% faster
- With 16-bit memory Thumb code is 45% faster than ARM code

Thumb code uses 30% less external memory power than ARM code

### **Question Bank:**

1. Explain the different addressing modes of the Arm processor
2. What are the features of thumb instructions
3. Write an assembly program of HEX TO ASCII conversion for ARM
4. Write an assembly program to divide a 32-bit number by an 8-bit number
5. Draw the architectural block diagram of ARM and explain data flow of each unit.
6. Explain the working of "Barrel shifter" with an example instruction and diagram.
7. Explain the function of following instructions one by one:
  - i) SUB r0, r1, #5
  - ii) ADD r2, r3, r3, LSL, #2
  - iii) LOR r0, [r1]
  - iv) SWP r3, r2, [r1]
  - v) ADDEQ r5, r5, r6



8. How ZIGBEE can be interfaced with an ARM processor. Draw and explain an interfacing diagram.
9. Explain the need for a fast interrupt service and a normal interrupt service in ARM programmer model with proper diagram.
10. What do you mean by addressing mode? Describe any four.

### **TEXT / REFERENCE BOOKS**

1. Andrew N. Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guide: Designing & Optimizing System Software, Elsevier, 2004.
2. Jonathan W. Valvano, Embedded Microcomputer Systems: Real Time Interfacing, Cengage Learning, 2011
3. Wayne Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2008.
4. C.M. Krishna, Kang G. Shin, Real time systems, McGraw Hill, 3rd reprint, 2010.
5. Herma K., Real Time Systems: Design for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
6. William Hohl, ARM Assembly Language, Fundamentals and Techniques, Taylor & Francis, 2009



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY  
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS & INSTRUMENTATION**

## **UNIT-III**

**EMBEDDED SYSTEM DESIGN— SBMA5201**

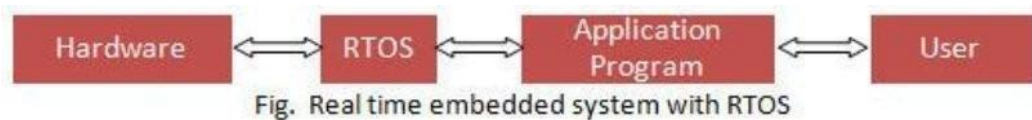
# REAL TIME OPERATING SYSTEM

Real time operating systems (RTOS) – real time kernel – OS tasks – task states – task scheduling – interrupt processing – clocking communication and synchronization – control blocks – memory requirements and control – kernel services

## 1. RTOS

A Real-Time Operating System (RTOS) comprises of two components, viz., “Real- Time” and “Operating System”. An Operating system (OS) is nothing but a collection of system calls or functions which provides an interface between hardware and application programs. It manages the hardware resources of a computer and hosting applications that run on the computer. An OS typically provides multitasking, synchronization, Interrupt and Event Handling, Input/ Output, Inter-task Communication, Timers and Clocks and Memory Management. Core of the OS is the Kernel which is typically a small, highly optimized set of libraries.

Real-time systems are those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.



**Figure 3.1** *real time embedded system with RTOS*

RTOS is key to many embedded systems and provides a platform to build applications. All embedded systems are not designed with RTOS. Embedded systems with relatively simple/small hardware/code might not require an RTOS. Embedded systems with moderate-to-large software applications require some form of scheduling, and hence RTOS.

## 2.Difference: RTOS v/s General Purpose OS

**Determinism** - The key difference between general-computing operating systems and real-time operating systems is the “deterministic ” timing behavior in the real-time operating systems. "Deterministic" timing means that OS consume only known and expected amounts of time. RTOS have their worst case latency defined. Latency is not of a concern for General Purpose OS.

**Task Scheduling** - General purpose operating systems are optimized to run a variety of applications and processes simultaneously, thereby ensuring that all tasks receive at least some processing time. As a consequence, low-priority tasks may have their priority boosted above other higher priority tasks, which the designer may not want. However, RTOS uses priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently. All system calls are deterministic, implying time bounded operation for all operations and ISRs. This is important for embedded systems where delay could cause a safety hazard. The scheduling in RTOS is time based. In case of General purpose OS, like Windows/Linux, scheduling is process based.

- **Preemptive kernel** - In RTOS, all kernel operations are preemptible
- **Priority Inversion** - RTOS have mechanisms to prevent priority inversion
- **Usage** - RTOS are typically used for embedded applications, while General Purpose OS are used for Desktop PCs or other generally purpose PCs.

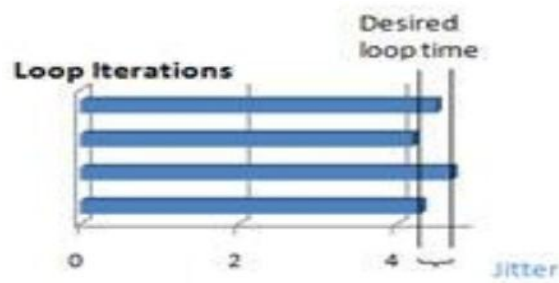
### 3. RTOS CLASSIFICATION

RTOS specifies a known maximum time for each of the operations that it performs. Based upon the degree of tolerance in meeting deadlines, RTOS are classified into following categories

- **Hard real-time:** Degree of tolerance for missed deadlines is negligible. A missed deadline can result in catastrophic failure of the system
- **Firm real-time:** Missing a deadline might result in an unacceptable quality reduction but may not lead to failure of the complete system
- **Soft real-time:** Deadlines may be missed occasionally, but system

doesn't fail and also, system quality is acceptable

For a life saving device, automatic parachute opening device for skydivers, delay can be fatal. Parachute opening device deploys the parachute at a specific altitude based on various conditions. If it fails to respond in specified time, parachute may not get deployed at all leading to casualty. Similar situation exists during inflation of air bags, used in cars, at the time of accident. If airbags don't get inflated at appropriate time, it may be fatal for a driver. So such systems must be hard real time systems, whereas for TV live broadcast, delay can be acceptable. In such cases, soft real time systems can be used



***Fig 3. 2 Jitter***

Jitter: Jitter is an indirect information obtained from several latency measures, consisting of a random variation between each latency value. In a RTOS, the jitter impact could be notorious, as it is analyzed by Proctor when trying to control step motors. For example, the pulses duration controls the motor rotation, but the jitter induce the torque to vary, causing step losses in the motor [Proctor and Shackelford 2001]. To compute jitter, the time difference between two

consecutive interrupt latency measures is calculated. Finally, the greatest encountered difference is selected as the worst jitter of this system;

### **3.1 Important terminologies used in context of real time systems**

**Determinism:** An application is referred to as deterministic if its timing can be guaranteed within a certain margin of error.

**Jitter:** Timing error of a task over subsequent iterations of a program or loop is referred to as jitter. RTOS are optimized to minimize jitter

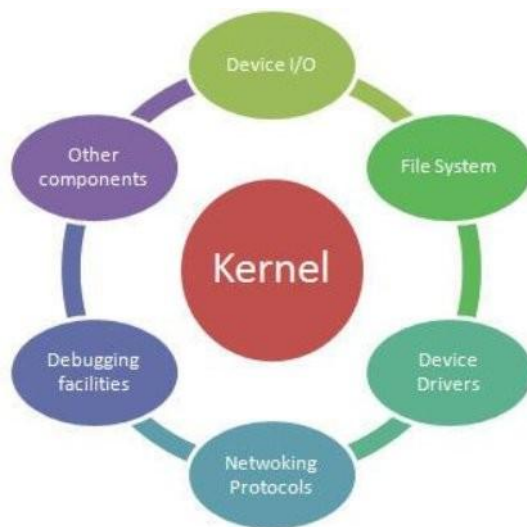
## **4. RTOS Features**

- i. Multithreading and preemptability - The scheduler should be able to preempt any task in the system and allocate the resource to the thread that needs it most even at peak load.
- ii. Thread Priority - All tasks are assigned priority level to facilitate pre-emption. The highest priority task that is ready to run will be the task that will be running.
- iii. Inter Task Communication & Synchronization - Multiple tasks pass information among each other in a timely fashion and ensuring data integrity
- iv. Priority Inheritance - RTOS should have large number of priority levels & should prevent priority inversion using priority inheritance.
- v. Short Latencies - The latencies are short and predefined.
  - Task switching latency: The time needed to save the context of a currently executing task and switching to another task.
  - Interrupt latency: The time elapsed between execution of the last instruction of the interrupted task and the first instruction in the interrupt handler.
  - Interrupt dispatch latency: The time from the last instruction in the interrupt handler to the next task scheduled to run.

## 5.RTOS Architecture

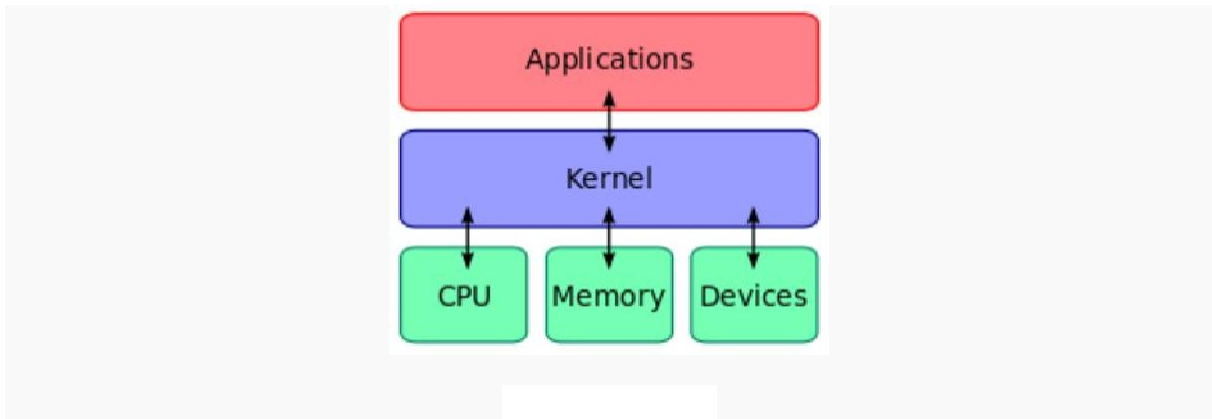
For simpler applications, RTOS is usually a kernel but as complexity increases, various modules like networking protocol stacks debugging facilities, device I/Os are includes in addition to the kernel The general architecture of RTOS is shown in the fig.3.3

### 5.1KERNEL



**Figure 3.3 RTOS Architecture**

Kernel" – the part of an operating system that provides the most basic services to application software running on a processor. The "kernel" of a real-time operating system ("RTOS") provides an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run



A kernel connects the application software to the hardware of a computer. With the aid of the firmware and device drivers, the kernel provides the most basic level of control over all of the computer's hardware devices. It manages memory access for programs.



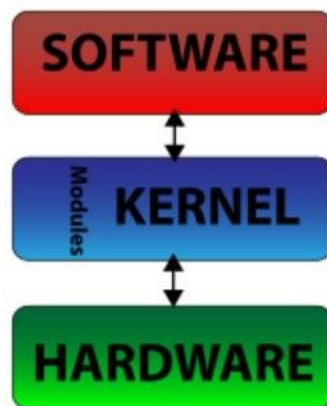
in the RAM, it determines which programs get access to which hardware resources, it sets up or resets the CPU's operating states for optimal operation at all times, and it organizes the data for long-term non-volatile storage with file systems on such media as disks, tapes, flash memory, etc RTOS kernel acts as an abstraction layer between the hardware and the applications.

There are three broad categories of kernels

- **Monolithic kernel**

A monolithic kernel executes all the operating system instructions in the same address space in order to improve the performance of the system.

Monolithic kernels are part of Unix-like operating systems like Linux, FreeBSD etc. A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) and provides powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel.



**Figure 3.5 Diagram of a monolithic kernel**

In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access.. The main disadvantages of monolithic kernels are the dependencies between

system components — a bug in a device driver might crash the entire system — and the fact that large kernels can become very difficult to maintain.

. In the monolithic kernel, some advantages hinge on these points:

- Since there is less software involved it is faster.
- As it is one single piece of software it should be smaller both in source and compiled forms.
- Less code generally means fewer bugs which can translate to fewer security problems
- This design has several flaws and limitations:

Coding in kernel can be challenging, in part because one cannot use common libraries and because one needs to use a source-level debugger. Rebooting the computer is often required. This is not just a problem of convenience to the developers. When debugging is harder, and as difficulties become stronger, it becomes more likely that code will be "buggier".

Bugs in one part of the kernel have strong side effects; since every function in the kernel has all the privileges, a bug in one function can corrupt data structure of another, totally unrelated part of the kernel, or of any running program.

Kernels often become very large and difficult to maintain.

Even if the modules servicing these operations are separate from the whole, the code integration is tight and difficult to do correctly.

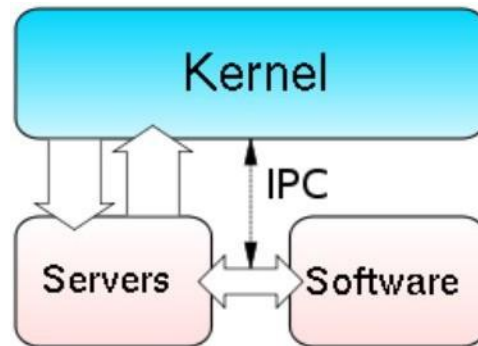
Since the modules run in the same address space, a bug can bring down the entire system.

Monolithic kernels are not portable; therefore, they must be rewritten for each new architecture that the operating system is to be used on.

## **5.2 Microkernel**

A microkernel runs most of the operating system's background processes in user space, to make the operating system more modular and, therefore, easier to maintain. It runs only basic process communication (messaging) and I/O control. It normally provides only the minimal services such as managing memory protection, Inter process communication and the process management. The other functions such as running the hardware processes are not

handled directly by microkernels. Thus, micro kernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as the kernel is unaffected even if the servers failed (i.e. File System). Microkernels are part of the operating systems like AIX, BeOS, Mach, Mac OS X, MINIX, and QNX. Etc



**Figure3.6 Microkernel**

The very essence of the microkernel architecture illustrates some of its advantages:

- Maintenance is generally easier.
- Patches can be tested in a separate instance, and then swapped in to take over a production instance.
- Rapid development time and new software can be tested without having to reboot the kernel.
- More persistence in general, if one instance goes hay-wire, it is often possible to substitute it with an operational mirror

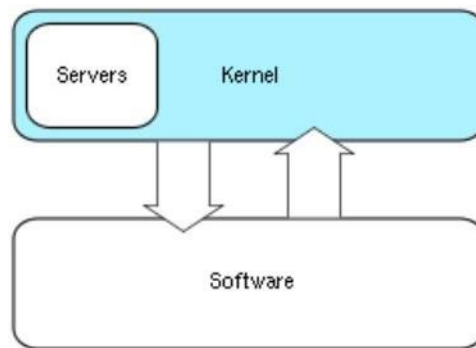
**Disadvantages in the microkernel exist however. Some are:**

- Larger running memory footprint
- More software for interfacing is required, there is a potential for performance loss.
- Messaging bugs can be harder to fix due to the longer trip they have to take versus the one off copy in a monolithic kernel.

- Process management in general can be very complicated

### 5.3 Hybrid Kernel

Hybrid kernels are extensions of microkernels with some properties of monolithic kernels. Hybrid kernels are similar to microkernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space. These are part of the operating systems such as Microsoft Windows NT, 2000 and XP. DragonFly BSD, etc



**Figure 3.7 Hybrid kernel**

Hybrid kernels are micro kernels that have some "non-essential" code in kernel-space in order for the code to run more quickly than it would were it to be in user-space. Hybrid kernels are a compromise between the monolithic and microkernel designs. This implies running some services (such as the network stack or the file system) in kernel space to reduce the performance overhead of a traditional microkernel, but still running kernel code (such as device drivers) as servers in user space

A few advantages to the modular (or) Hybrid kernel are:

- Faster development time for drivers that can operate from within modules. No reboot required for testing (provided the kernel is not destabilized).
- On demand capability versus spending time recompiling a whole kernel for things like new drivers or subsystems.
- Faster integration of third party technology (related to development but pertinent unto itself nonetheless)
- Some of the disadvantages of the modular approach are:
- With more interfaces to pass through, the possibility of increased bugs exists (which implies more security holes).
- Maintaining modules can be confusing for some administrators when dealing with problems like symbol differences

## **5.4 Nano Kernels**

A nanokernel delegates virtually all services — including even the most basic ones like interrupt controllers or the timer — to device drivers to make the kernel memory requirement even smaller than a traditional microkernel

## **5.6 Exokernel**

Exokernels provides efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems and leaving the management to the application.

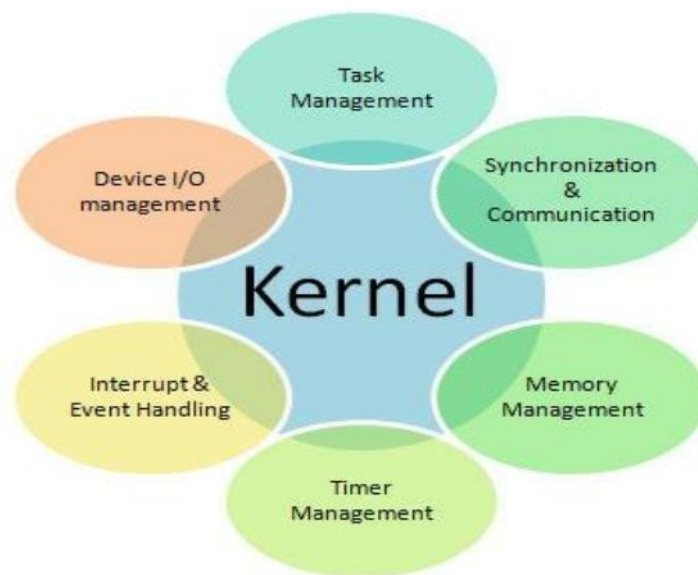
Exokernels are a still-experimental approach to operating system design. They differ from the other types of kernels in that their functionality is limited to the protection and multiplexing of the raw hardware, providing no hardware abstractions on top of which to develop applications. This separation of hardware protection from hardware management

enables application developers to determine how to make the most efficient use of the available hardware for each specific program.

Exokernels in themselves are extremely small. However, they are accompanied by library operating systems providing application developers with the functionalities of a conventional operating system. A major advantage of exokernel-based systems is that they can incorporate multiple library operating systems, each exporting a different API, for example one for high level UI development and one for real-time control

### 5.7 Kernel Services

Six types of common services are shown in the following figure below and explained in subsequent sections



**Figure 3.8 kernel service**

**RTOS** is therefore an operating system that supports real-time applications by providing logically correct result within the deadline required. Basic Structure is similar to

regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks.

Though real-time operating systems may or may not increase the speed of execution, they can provide much more precise and predictable timing characteristics than general-purpose OS.

### **5.8 Real Time Kernel**

The heart of a real-time OS (and the heart of every OS, for that matter) is the **kernel**. A kernel is the central core of an operating system, and it takes care of all the OS jobs:

1. Booting
2. Task Scheduling
3. Standard Function Libraries

In an embedded system, frequently the kernel will boot the system, initialize the ports and the global data items. Then, it will start the scheduler and instantiate any hardware timers that need to be started. After all that, the Kernel basically gets dumped out of memory (except for the library functions, if any), and the scheduler will start running the child tasks.

*A real-time kernel is software that manages the time and resources of a*

*microprocessor, microcontroller or Digital Signal Processor (DSP), and provides indispensable services to your applications.*

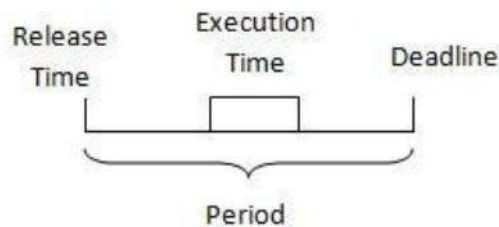
*A Real Time Operating System (RTOS) generally contains a real-time kernel and other higher-level services such as file management,*

## **6.TASK MANAGEMENT**

### **Task Object**

In RTOS, The application is decomposed into small, schedulable, and sequential program units known as “Task”, a basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time. Release time refers to the

point in time from which the task can be executed. Deadline is the point in time by which the task must complete. Execution time denotes the time the task takes to execute.



**Figure 3.9 basic unit of execution**

Each task may exist in following states

### **OS tasks Task States**

Task States: – Running – Ready (possibly: suspended, pended) – Blocked (possibly: waiting, dormant, delayed) – Scheduler – schedules/shuffles tasks between Running and Ready states – Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied) – When a task is unblocked with a

**Running:** This is the task which has control of the CPU. It will normally be the task which has the highest current priority of the tasks which are ready to run.

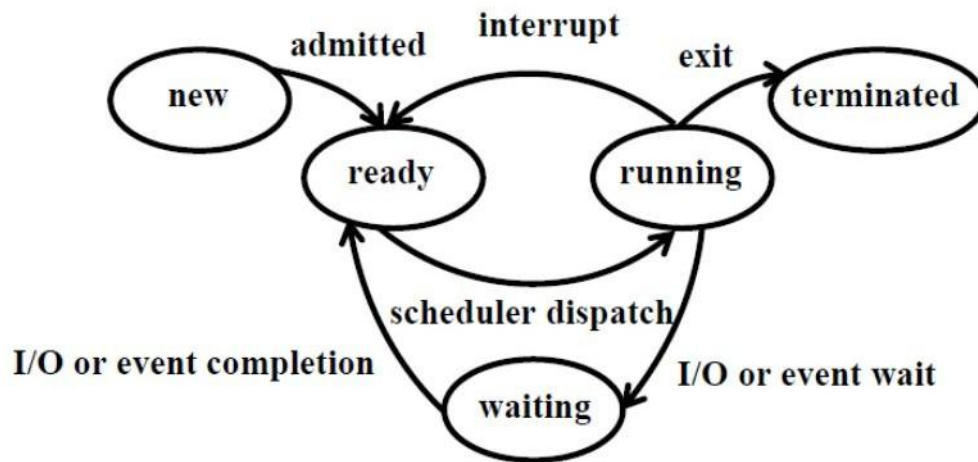
**Ready:** There may be several tasks in this state. The attributes of the task and the resources required to run it must be available for it to be placed in the 'ready' state.

**Waiting:** The execution of tasks placed in this state has been suspended because the task requires some resources which is not available or because the task is waiting for some signal from the plant, e.g., input from the analog-to-digital converter, or the task is waiting for the elapse of time.

**New:** The operating system is aware of the existence of this task, but the task has not been allocated a priority and a context and has not been included into the list of schedulable tasks



Terminated: The operating system has not as yet been made aware of the existence of this task, although it may be resident in the memory of the computer.



**Figure 3. 10 task state**

When a task is “spawned”, either by the operating system, or another task, it is to be created, which involves loading it into the memory, creating and updating certain OS data structures such as the Task Control Block, necessary for running the task within the multi-tasking environment.

During such times the task is in the new state. Once these are over, it enters the ready state where it waits. At this time it is within the view of the scheduler and is considered for execution according to the scheduling policy.

A task is made to enter the running state from the ready state by the operating system dispatcher when the scheduler determines the task to be the one to be run according to its scheduling policy. While the task is running, it may execute a normal or abnormal exit according to the program logic, in which case it enters the terminated state and then removed from the view of the OS. Software or hardware interrupts may also occur while the task is running.

In such a case, depending on the priority of the interrupt, the current task may be transferred to the ready state and wait for its next time allocation by the scheduler

Finally, a task may need to wait at times during its course of execution, either due to requirements of synchronization with other tasks or for completion of some service such as I/O that it has requested for.

During such a time it is in the waiting state. Once the synchronization requirement is fulfilled, or the requested service is completed, it is returned to the ready state to again wait its turn to be scheduled.

During the execution of an application program, individual tasks are continuously changing from one state to another. However, only one task is in the running mode (i.e. given CPU control) at any point of the execution. In the process where CPU control is change from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved, the process referred to as context switching. A task object is defined by the following set of component

## **7. TASK CONTROL BLOCK (TCB)**

Task uses TCBs to remember its context. TCBs are data structures residing in RAM, accessible only by RTOS

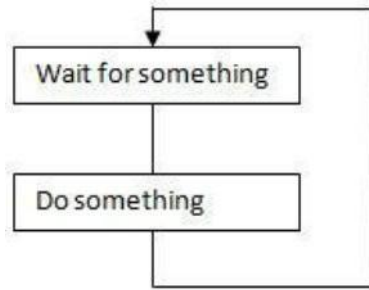
Task_ID
Task_State
Task_Priority
Task_Stack_Pointer
Task_Prog
_Counter

- Task Stack: These reside in RAM, accessible by stack pointer.
- Task Routine: Program code residing in ROM

## Scheduler

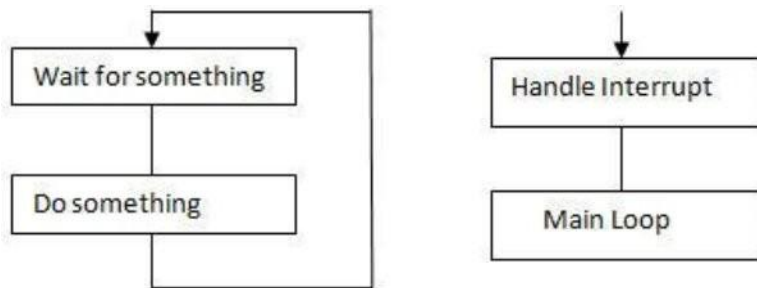
The scheduler keeps record of the state of each task and selects from among them that are ready to execute and allocates the CPU to one of them. Various scheduling algorithms are used in RTOS

- Polled Loop: Sequentially determines if specific task requires time.



**Figure 3.11 polled loop**

Polled System with interrupts. In addition to polling, it takes care of critical tasks



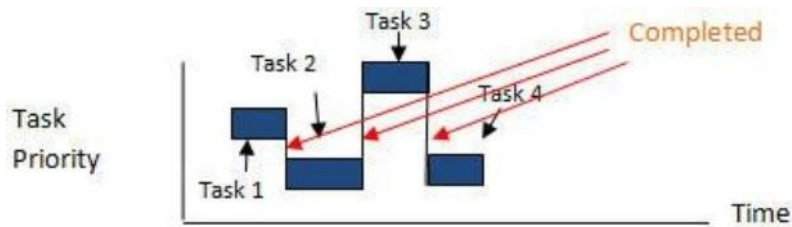
**Figure 3.12 polled with interrupts**

**Round Robin :** Sequences from task to task, each task getting a slice of time

**Hybrid System:** Sensitive to sensitive interrupts, with Round Robin system working in background.

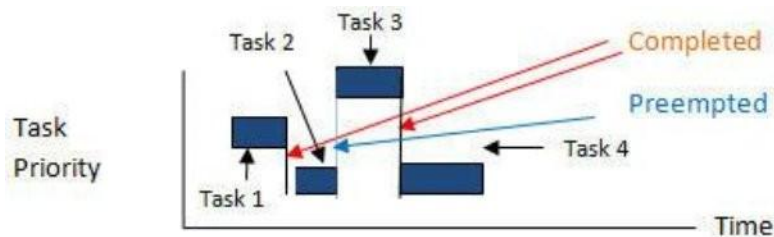
**Interrupt Driven:** System continuously wait for the interrupts

Non pre-emptive scheduling or Cooperative Multitasking: Highest priority task executes for some time, then relinquishes control, re-enters ready state.



**Fig 3.13 Non pre-emptive scheduling**

Preemptive scheduling Priority multitasking: Current task is immediately suspended Control is given to the task of the highest priority at all time



**Figure 3.14 Non pre-emptive scheduling**

### Dispatcher

The dispatcher gives control of the CPU to the task selected by the scheduler by performing context switching and changes the flow of execution.

## **8.TASK SCHEDULING**

### **Scheduling**

The data structure of the ready list in the scheduler is designed so as to minimize the worst-case length of time spent in the scheduler's critical section • The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task. In a well-designed RTOS, readying a newtask will take 3-20 instructions per ready queue entry, and restoration of the highest priority ready task will take 5-30 instructions.

### **Inter task Comm. & resource sharing**

It is "unsafe" for two tasks to access the same specific data or hardware resource simultaneously.

- 3 Ways to resolve this:
- Temporarily masking/disabling interrupts
- Binary Semaphores
- Message passing

### **Memory Allocation**

- Speed of allocation
- Memory can become fragmented

### **Interrupt Handling**

- Interrupts usually block the highest priority tasks

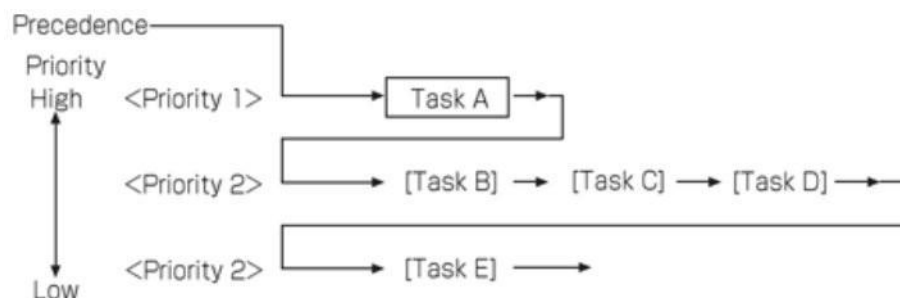
## Task Scheduling Rules

The T-Kernel adopts a preemptive priority-based scheduling method based on priority levels assigned to each task. Tasks having the same priority are scheduled on a FCFS (First Come First Served) basis. Specifically, task precedence is used as the task scheduling rule, and precedence among tasks is determined as follows based on the priority of each task. If there are multiple tasks that can be run, the one with the highest precedence goes to RUNNING state and the others go to READY state. In determining precedence among tasks, of those tasks having different priority levels, that with the highest priority has the highest precedence. Among tasks having the same priority, the one that entered a run state (RUNNING state or READY state) first has the highest precedence. It is possible, however, to use a system call to change the precedence among tasks having the same priority. When the task with the highest precedence changes from one task to another, a dispatch occurs immediately and the task in RUNNING state is switched. If no dispatch occurs (during execution of a handler, during dispatch disabled state, etc.), however, the switching of the task in RUNNING state is held off until the next dispatch occurs. According to the scheduling rules adopted in the T-Kernel, so long as there is a higher precedence task in a run state, a task with lower precedence will simply not run. That is, unless the highest-precedence task goes to WAITING state or for other reason cannot run, other tasks are not run. This is a fundamental difference from TSS (Time Sharing System) scheduling in which multiple tasks are treated equally. It is possible, however, to issue a system call changing the precedence among tasks having the same priority. An application can use such a system call to realize round-robin scheduling, which is a typical kind of TSS scheduling. Examples in figures below illustrate how the task that first goes to a run state (RUNNING state or READY state) gains precedence among tasks having the same priority figure shows the precedence among tasks after Task A of priority 1, Task E of priority 3, and Tasks B, C and D of priority 2 are started in that order. The task with the highest precedence, Task A, goes to RUNNING state. When Task A exits, Task B. with the next-highest precedence goes to RUNNING state (Figure 3). When Task A is again started, Task B is preempted and reverts to READY state; but since Task B went to a run state earlier than Task C and Task D, it still has the highest precedence among tasks with the same priority. In other words, the task precedence reverts to that in Figure 2.

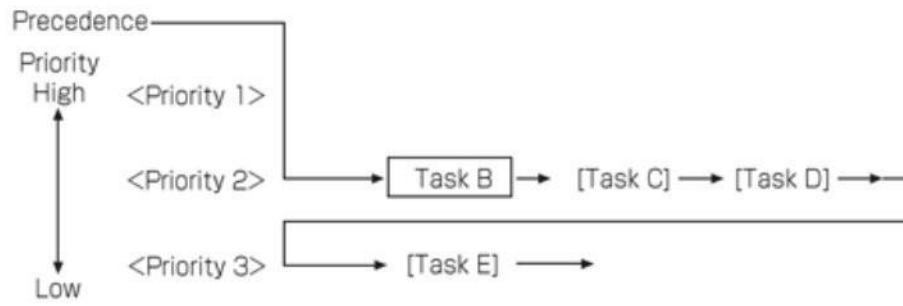
Next, consider what happens when Task B goes to WAITING state in the conditions in Figure 3. Since task precedence is defined among tasks that can be run, the precedence among tasks becomes as shown in 3.15. Thereafter when the Task B waiting state is released, Task B goes to run state after Task C and Task D, and thus assumes the lowest precedence among tasks of the same priority

Summarizing the above, immediately after a task that goes from READY state to RUNNING state reverts to READY state, it has the highest precedence among tasks of the same priority; but after a task goes from RUNNING state to WAITING state and then the wait is released, its precedence is the lowest among tasks of the same priority.

Note that after a task goes from SUSPENDED state to a run state, it has the lowest precedence among tasks of the same priority. In a virtual memory system, if a task is made to wait for paging by putting the task in SUSPENDED state, in such a system the task precedence changes as a result of a paging wait.

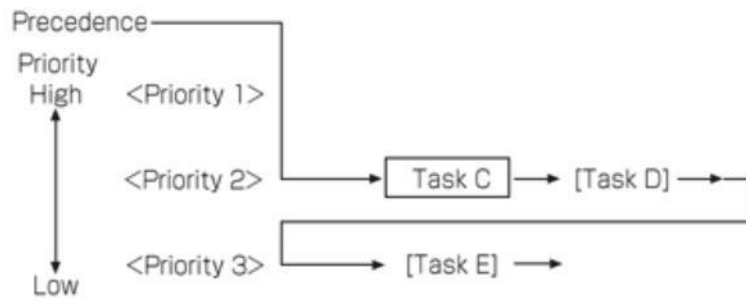


**Figure 3.15 Precedence in Initial State**

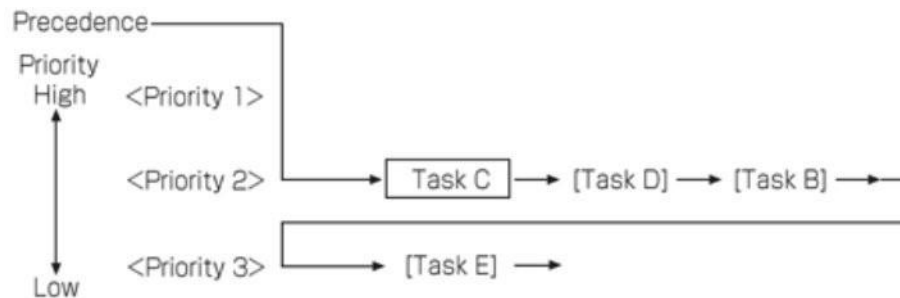


**Figure 3.16 Precedence after Task B Goes To RUNNING State**





**Figure 3.17. Precedence after Task B Goes To WAITING State**



**Figure 3.18 Precedence After Task B WAITING State Is Released**

### **Synchronisation and communication**

Task Synchronisation & intertask communication serves to pass information amongst tasks.

#### **Task Synchronisation**

Synchronization is essential for tasks to share mutually exclusive resources (devices, buffers, etc) and/or allow multiple concurrent tasks to be executed (e.g. Task A needs a result from task B, so task A can only run till task B produces it).

Task synchronization is achieved using two types of mechanisms:

### Event Objects

Event objects are used when task synchronization is required without resource sharing. They allow one or more tasks to keep waiting for a specified event to occur. Event object can exist either in triggered or non-triggered state. Triggered state indicates resumption of the task.

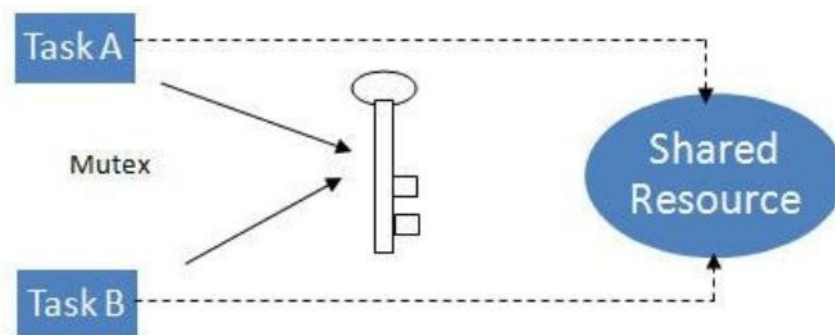
### Semaphores.

A semaphore has an associated resource count and a wait queue. The resource count indicates availability of resource. The wait queue manages the tasks waiting for resources from the semaphore. A semaphore functions like a key that define whether a task has the access to the resource. A task gets an access to the resource when it acquires the semaphore.

There are three types of semaphore:

- Binary Semaphores
- Counting Semaphores
- **Mutually Exclusion(Mutex)** Semaphores

Semaphore functionality (Mutex) represented pictorially in the following figure



**Figure 3.19. semaphore**

## **Intertask communication**

Intertask communication involves sharing of data among tasks through sharing of memory space, transmission of data, etc. Intertask communications is executed using following mechanisms

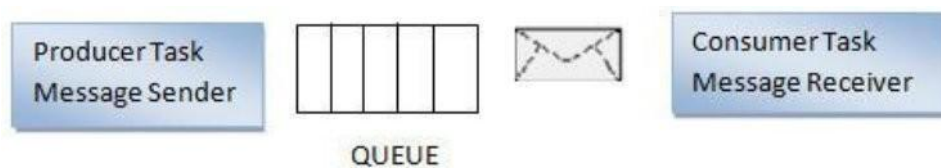
**Message queues** - A message queue is an object used for intertask communication through which task send or receive messages placed in a shared memory. The queue may follow

- 1) First In First Out (FIFO),
- 2) Last in First Out(LIFO) or

### 3) Priority (PRI) sequence.

Usually, a message queue comprises of an associated queue control block (QCB), name, unique ID, memory buffers, queue length, maximum message length and one or more task waiting lists. A message queue with a length of 1 is commonly known as a mailbox.

**Pipes** - A pipe is an object that provide simple communication channel used for unstructured data exchange among tasks. A pipe does not store multiple messages but stream of bytes. Also, data flow from a pipe cannot be prioritize



**Figure 3.20 inter task communication**

**Remote procedure call (RPC)** - It permits distributed computing where task can invoke the execution of another task on a remote computer.

**Intertask communication and resource sharing** Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. "Unsafe" means the results are inconsistent or unpredictable. There are three common approaches to resolve this problem:

#### **Temporarily masking/disabling interrupts**

General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wishes. Some modern CPUs don't allow user mode code to disable interrupts as such control is considered a key operating system resource. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater syscall efficiency and also to permit the application to have greater control of the operating environment without requiring

OS intervention. On single-processor systems, if the application runs in kernel mode and can mask interrupts, this method is the solution with the lowest overhead to prevent simultaneous access to a shared resource. While interrupts are masked and the current task does not make a blocking OS call, then the current task has *exclusive* use of the CPU since no other task or interrupt can take control, so the critical section is protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few instructions and contains no loops. This method is ideal for protecting hardware bit-mapped registers when the bits are controlled by different tasks.

### **Binary semaphores**

When the shared resource must be reserved without blocking all other tasks (such as waiting for Flash memory to be written), it is better to use mechanisms also available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS's dispatcher code on exit, so they typically take hundreds of CPU instructions to execute, while masking interrupts may take as few as one instruction on some processors.

A binary semaphore is either **locked** or unlocked. When it is locked, tasks must wait for the semaphore to unlock. A binary semaphore is therefore equivalent to a mutex. Typically a task will set a timeout on its wait for a semaphore. There are several well-known problems with semaphore based designs such as priority inversion and deadlocks.

In priority inversion a high priority task waits because a low priority task has a semaphore, but the lower priority task is not given CPU time to finish its work. A typical solution is to have the task that owns a semaphore run at, or 'inherit,' the priority of the highest waiting task. But this simple approach fails when there are multiple levels of

waiting: task *A* waits for a binary semaphore locked by task *B*, which waits for a binary semaphore locked by task *C*. Handling multiple levels of inheritance without introducing instability in cycles is complex and problematic.

In a deadlock, two or more tasks lock semaphores without timeouts and then wait forever for the other task's semaphore, creating a cyclic dependency. The simplest deadlock scenario occurs when two tasks alternately lock two semaphores, but in the opposite order. Deadlock is prevented by careful design or by having floored semaphores, which pass control of a semaphore to the higher priority task on defined conditions.

### **Message passing**

The other approach to resource sharing is for tasks to send messages in an organized message passing scheme. In this paradigm, the resource is managed directly by only one task. When another task wants to interrogate or manipulate the resource, it sends a message to the managing task. Although their real-time behavior is less crisp than semaphore systems, simple message-based systems avoid most protocol deadlock hazards, and are generally better-behaved than semaphore systems. However, problems like those of semaphores are possible. Priority inversion can occur when a task is working on a low-priority message and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its incoming message queue. Protocol deadlocks can occur when two or more tasks wait for each other to send response messages

### **Memory Management**

Two types of memory managements are provided in RTOS – Stack and Heap. Stack management is used during context switching for TCBs. Memory other than memory used for program code, program data and system stack is called heap memory and it is used for dynamic allocation of data space for tasks. Management of this memory is called heap management. Memory allocation is more critical in a real-time operating system than in other operating systems.

First, for stability there cannot be memory leaks (memory that is allocated, then unused but never freed). The device should work indefinitely, without ever a need for a reboot.

For this reason, dynamic memory allocation is frowned upon. Whenever possible, allocation

of all required memory is specified statically at compile time.

Another reason to avoid dynamic memory allocation is memory fragmentation. With frequent allocation and releasing of small chunks of memory, a situation may occur when the memory is divided into several sections, in which case the RTOS cannot allocate a large continuous block of memory, although there is enough free memory. Secondly, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block, which is unacceptable in an RTOS since memory allocation has to occur within a certain amount of time. Because mechanical disks have much longer and more unpredictable response times, swapping to disk files is not used for the same reasons as RAM allocation discussed above.

The simple fixed-size-blocks algorithm works quite well for simple embedded systems because of its low overhead

### **Memory management**

Among other things, a multiprogramming operating system kernel must be responsible for managing all system memory which is currently in use by programs. This ensures that a program does not interfere with memory already in use by another program. Since programs time share, each program must have independent access to memory.

Cooperative memory management, used by many early operating systems, assumes that all programs make voluntary use of the kernel's memory manager, and do not exceed their allocated memory. This system of memory management is almost never seen any more, since programs often contain bugs which can cause them to exceed their allocated memory. If a program fails, it may cause memory used by one or more other programs to be affected or overwritten. Malicious programs or viruses may purposefully alter another program's memory, or may affect the operation of the operating system itself. With cooperative memory management, it takes only one misbehaved program to crash the system.

Memory protection enables the kernel to limit a process' access to the computer's memory. Various methods of memory protection exist, including memory

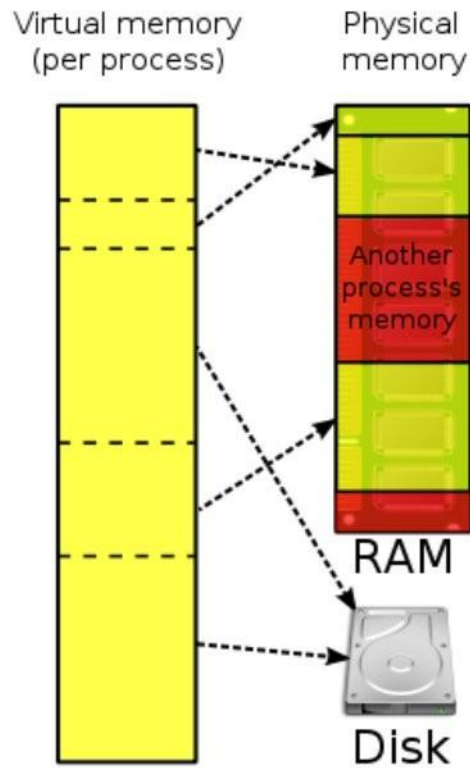
segmentation and paging. All methods require some level of hardware support (such as the 80286 MMU), which doesn't exist in all computers. In both segmentation and paging, certain protected mode registers specify to the CPU what memory address it should allow a running program to access. Attempts to access other addresses trigger an interrupt which cause the CPU to re-enter supervisor mode, placing the kernel in charge. This is called a segmentation violation or Seg-V for short, and since it is both difficult to assign a meaningful result to such an operation, and because it is usually a sign of a misbehaving program, the kernel generally resorts to terminating the offending program, and reports the error. Windows versions 3.1 through ME had some level of memory protection, but programs could easily circumvent the need to use it. A general protection fault would be produced, indicating a segmentation violation had occurred; however, the system would often crash anyway.

## **Virtual memory**

Many operating systems can "trick" programs into using memory scattered around the hard disk and RAM as if it is one continuous chunk of memory, called virtual memory. The use of virtual memory addressing (such as paging or segmentation) means that the kernel can choose what memory each program may use at any given time, allowing the operating system to use the same memory locations for multiple tasks. If a program tries to access memory that isn't in its current range of accessible memory, but nonetheless has been allocated to it, the kernel is interrupted in the same way as it would if the program were to exceed its allocated memory. (See section on memory management.) Under UNIX this kind of interrupt is referred to as a page fault. When the kernel detects a page fault it generally adjusts the virtual memory range of the program which triggered it, granting it access to the memory requested. This gives the kernel discretionary power over where a particular application's memory is stored, or even whether or not it has actually been allocated yet. In modern operating systems, memory which is accessed less frequently can be temporarily stored on disk or other media to make that space available for use by other programs. This is called swapping, as an area of memory can be used by multiple programs, and what that memory area contains can be swapped or exchanged on demand.



"Virtual memory" provides the programmer or the user with the perception that there is a much larger amount of RAM in the computer than is really there



**Figure 3.21 virtual memory**

### **Timer Management**

Tasks need to be performed after scheduled durations. To keep track of the delays, timers- relative and absolute- are provided in RTOS.

## **Interrupt and event handling**

RTOS provides various functions for interrupt and event handling, viz., Defining interrupt handler, creation and deletion of ISR, referencing the state of an ISR, enabling and disabling of an interrupt, etc. It also restricts interrupts from occurring when modifying a data structure, minimize interrupt latencies due to disabling of interrupts when RTOS is performing critical operations, minimizes interrupt response times.

Interrupts are central to operating systems, as they provide an efficient way for the operating system to interact with and react to its environment. The alternative – having the operating system "watch" the various sources of input for events (polling) that require action – can be found in older systems with very small stacks (50 or 60 bytes) but is unusual in modern systems with large stacks. Interrupt-based programming is directly supported by most modern CPUs. Interrupts provide a computer with a way of automatically saving local register contexts, and running specific code in response to events. Even very basic computers support hardware interrupts, and allow the programmer to specify code which may be run when that event takes place.

When an interrupt is received, the computer's hardware automatically suspends whatever program is currently running, saves its status, and runs computer code previously associated with the interrupt; this is analogous to placing a bookmark in a book in response to a phone call. In modern operating systems, interrupts are handled by the operating system's kernel. Interrupts may come from either the computer's hardware or the running program.

When a hardware device triggers an interrupt, the operating system's kernel decides how to deal with this event, generally by running some processing code. The amount of code being run depends on the priority of the interrupt (for example: a person usually responds to a smoke detector alarm before answering the phone). The processing of hardware interrupts is a task that is usually delegated to software called a device driver, which may be part of the operating system's kernel, part of another program, or both. Device drivers may then relay information to a running program by various means.

A program may also trigger an interrupt to the operating system. If a program wishes to access hardware, for example, it may interrupt the operating system's kernel, which causes control to be passed back to the kernel. The kernel then processes the request. If a program wishes additional resources (or wishes to shed resources) such as memory, it triggers an interrupt to get the kernel's attention

## 9.HANDLING THE INTERRUPT

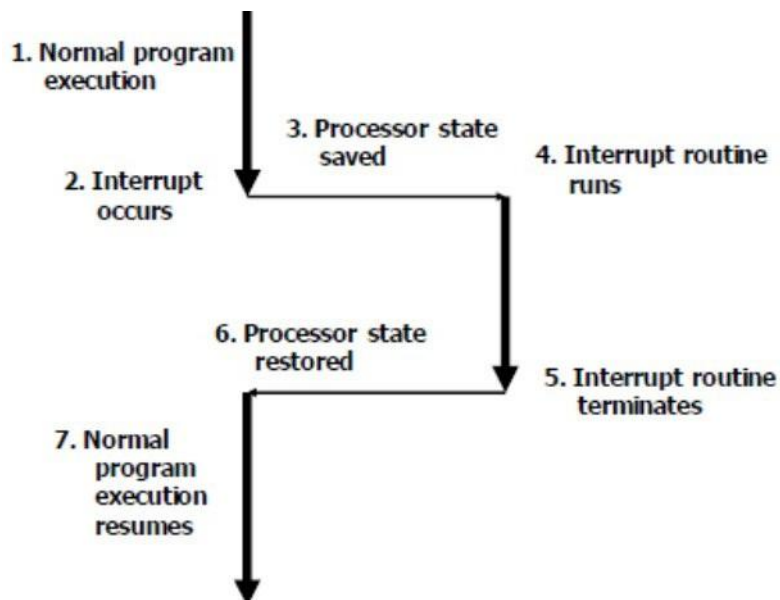


Figure 2.22 interrupt handling

Most interrupt routines: Copy peripheral data into a buffer Indicate to other code that data has arrived

Acknowledge the interrupt (tell hardware)

Longer reaction to interrupt performed outside interrupt routine

E.g., causes a process to start or resume running

## 9.1 Interrupt Handling

One of the central tasks of real-time software is the processing of interrupts. As soon as several tasks run in a program, it is virtually impossible to achieve good response times by polling (continuous enquiry of an event). Continuous polling would prevent tasks with lower priorities from running and thus waste precious CPU time.

Therefore, it should always be attempted to replace polling by interrupts. This leads to the best response times and optimal use of the hardware available.

RTKernel-32 uses the timer interrupt to activate tasks waiting for a certain point in time. Module RTCom provide interrupt support for serial ports. This section discusses how to implement a handler for any interrupt source.

An interrupt handler may be thought of as a task running with a priority higher than all other tasks. However, there are some important considerations to keep in mind.

Interrupt handlers usually have little stack space. Therefore, interrupt handlers should be very economical in their stack usage (e.g., refrain from using functions like `sprintf` by all means).

While an interrupt handler is active, no other interrupts with lower priorities can be processed. Therefore, it is important to minimize the execution times of interrupt handlers, because otherwise the interrupt response time for other interrupts might suffer. The handler should avoid any processing not immediately required and delegate it to a task.

Interrupt handlers under RTKernel-32 are never directly addressed by the hardware; instead, they are called by the low-level handlers or dispatchers of the kernel (see

section Module RTKernel-32, Interrupt Handling). While the handler is being executed, the scheduler is disabled; thus, the handler need not consider being disrupted by a task switch (it can, however, be interrupted by interrupts of higher priority). Since the scheduler is disabled, interrupt handlers must not force blocking task switches.

An interrupt handler may:

- declare and use local variables,
- call other functions,
- call functions RTKSignal or RTKWaitCond to activate other tasks,
- call the **conditional** mailbox and message passing operations (RTKPutCond, RTKGetCond, RTKSendCond, RTKReceiveCond) to exchange data with tasks.
- An interrupt handler must not:
  - use the coprocessor or emulator without saving/restoring its state,
  - use more than 512 bytes of stack (the less, the better),
  - cause blocking task switches.

## 9.2 Types of interrupts

Asynchronous (or hardware interrupt) by hardware event the interrupt handler as a separated task in a different context. Synchronous by software instruction a divide by zero, a memory segmentation fault, etc. The interrupt handler runs in the context of the interrupting task

## **Interrupt latency**

- The time delay between the arrival of interrupt and the start of corresponding ISR.
- Modern processors with multiple levels of caches and instruction pipelines that need to be reset before ISR can start might result in longer latency.
- The ISR of a lower-priority interrupt may be blocked by the ISR of a high-priority

### **9.3 Interrupt handlers and the scheduler**

Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware if possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns) and notify a task that work needs to be done. This can be done by unblocking a driver task through releasing a semaphore, setting a flag or sending a message. A scheduler often provides the ability to unblock a task from interrupt handler context.

An OS maintains catalogues of objects it manages such as threads, mutexes, memory, and so on. Updates to this catalogue must be strictly controlled. For this reason it can be problematic when an interrupt handler calls an OS function while the application is in the act of also doing so. The OS function called from an interrupt handler could find the object database to be in an inconsistent state because of the application's update. There are two major approaches to deal with this problem: the unified architecture and the segmented architecture. RTOSs implementing the unified architecture solve the problem by simply disabling interrupts while the internal catalogue is updated. The downside of this is that interrupt latency increases, potentially losing interrupts. The segmented architecture does not make direct OS calls but delegates the OS related work to a separate handler. This handler runs at a higher priority than any thread but lower than the interrupt handlers. The advantage of this architecture is that it adds very few cycles to interrupt latency. As a result, OSes which implement the segmented architecture are more predictable and can deal with higher interrupt rates compared to the unified architecture.

## **10.Device I/O Management**

RTOS generally provides large number of APIs to support diverse hardware device drivers.

A device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. It is a specialized hardware-dependent computer program which

is also operating system specific that enables another program, typically an operating system or applications software package or computer program running under the operating system kernel, to interact transparently with a hardware device, and usually provides the requisite interrupt handling necessary for any necessary asynchronous time-dependent hardware interfacing needs.

The key design goal of device drivers is abstraction. Every model of hardware (even within the same class of device) is different. Newer models also are released by manufacturers that provide more reliable or better performance and these newer models are often controlled differently. Computers and their operating systems cannot be expected to know how to control every device, both now and in the future. To solve this problem, operating systems essentially dictate how every type of device should be controlled. The function of the device driver is then to translate these operating system mandated function calls into device specific calls. In theory a new device, which is controlled in a new manner, should function correctly if a suitable driver is available. This new driver ensures that the device appears to operate as usual from the operating system's point of view.

Under versions of Windows before Vista and versions of Linux before 2.6, all driver execution was co-operative, meaning that if a driver entered an infinite loop it would freeze the system. More recent revisions of these operating systems incorporate kernel preemption, where the kernel interrupts the driver to give it tasks, and then separates itself from the process until it receives a response from the device driver, or gives it more tasks to do.

## **11.NETWORKING**

Currently most operating systems support a variety of networking protocols, hardware, and applications for using them. This means that computers running dissimilar operating systems can participate in a common network for sharing resources such as computing, files, printers, and scanners using either wired or wireless connections. Networks can essentially allow a computer's operating system to access the resources of a remote computer to support the same functions as it could if those resources were connected directly to the local computer. This includes everything from simple communication, to



using networked file systems or even sharing another computer's graphics or sound hardware. Some network services allow the resources of a computer to be accessed transparently, such as SSH which allows networked users direct access to a computer's command line interface.

Client/server networking allows a program on a computer, called a client, to connect via a network to another computer, called a server. Servers offer (or host) various services to other network computers and users. These services are usually provided through ports or numbered access points beyond the server's network address. Each port number is usually associated with a maximum of one running program, which is responsible for handling requests to that port. A daemon, being a user program, can in turn access the local hardware resources of that computer by passing requests to the operating system kernel.

Many operating systems support one or more vendor-specific or open networking protocols as well, for example, SNA on IBM systems, DECnet on systems from Digital Equipment Corporation, and Microsoft-specific protocols (SMB) on Windows. Specific protocols for specific tasks may also be supported such as NFS for file access. Protocols like ESound, or esd can be easily extended over the network to provide sound from local applications, on a remote system's sound hardware.

## 12.POPULAR RTOS

There are number of commercially available RTOS, each with some distinct features and targeted for a specific set of applications. Following table lists some of the widely used commercially available RTOS.

RTOS	Applications/Features
Windows CE	Used for Small footprint, mobile and connected devices Supported by ARM,MIPS, SH4 & x86 architectures
LynxOS	Complex, hard real-time applications POSIX-compatible, multiprocess, multithreaded OS. Supported by x86, ARM, PowerPC architectures

VxWorks	Most widely adopted RTOS in the embedded industry. Used in famous NASA rover robots Spirit and Opportunity
Micrium $\mu$ C/OS-II	Ported to more than a hundred architectures including x86, mainly used in microcontrollers with low resources.  certified by rigorous standards, such as RTCADO-178B
QNX	Most traditional RTOS in the market.  Microkernel architecture; completely compatible with the POSIX Certified by FAADO-278 and MIL-STD-1553 standards.
RT Linux	Hard realtime kernel  Good real time performance, but no certification
Jbed	Developed for embedded systems and Internet applications under the Java platform.  Allows an entire application including the device drivers to be written using Java
Symbian	Designed for Smartphones Supported by ARM, x86 architecture
VRTX	Suitable for traditional board based embedded systems and SoC architectures  Supported by ARM, MIPS, PowerPC & other RISC architectures

## RTOS Kernel Services

- The services provided by the embedded RTOS kernel can be divided into the following areas:
- Multitasking
  - Tasks
  - Timers

- Time-triggered tasks
- Synchronisation
  - Semaphores
  - Mutexes
  - Flags
- Intertask Communication
  - Mailboxes
  - Signals
- Error Detection
- Interrupt Services
- Memory Allocation
- Event Logging
- Statistics

**Multitasking** is a way of letting several different execution units, or tasks, share a single processor so that all tasks can be said to run in parallel. In reality the RTOS scheduler chooses which task it should run according to the scheduling policy, but because tasks can be swapped in and out of the processor at a high rate, the illusion of parallelism occurs.

**Timers** are a simpler form of tasks that are invoked periodically by the RTOS, as defined by the application.

**Time-triggered tasks** are tasks that are activated according to cyclic schedules defined by the application. Time-triggered tasks have priority over normal tasks. The schedule defines a set of time-triggered tasks, along with activation points and deadlines for each task. The RTOS monitors time-triggered tasks for missed deadlines.

**Semaphores**, mutexes and flags provide synchronisation between tasks, while mailboxes and signals also provide a way of exchanging data.

The rt-kernel embedded RTOS performs error detection when running in the RTOS kernel.

Unlike many traditional embedded RTOS, rt-kernel services do not return error codes. Instead, a common error handling routine is invoked whenever the RTOS kernel detects an error condition. This simplifies application code as the tedious and error-prone checking of return values is eliminated. Interrupt services are for the most part provided by the architecture layers of each processor. The RTOS provides a unified mechanism for enabling, disabling and attaching to interrupts, thereby making it possible to reuse drivers regardless of the system they were originally written for. The RTOS supports dynamic memory allocation. It can also be configured for static memory allocation. An optional event logging mechanism is provided. All application interactions with the RTOS kernel can be logged. A host tool is used to present the collected data. This is normally used during debugging when it can be a great help to visualise the behaviour of the real-time system. The RTOS can optionally continually monitor the processor load to provide some run-time *statistics*. There is also functionality for measuring the processor load over a given block of code.

## **Memory Allocation Strategies**

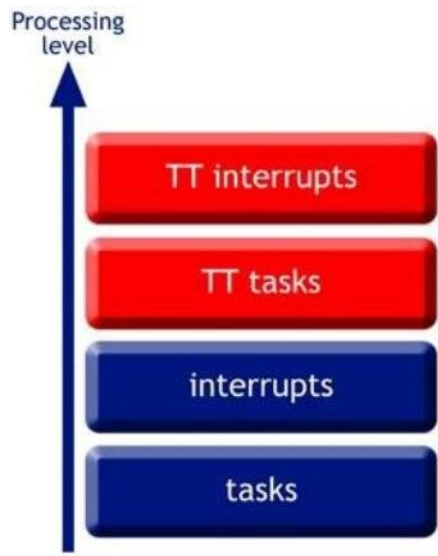
The RTOS provides the functionality needed for creating a dynamic real-time system. Tasks and all kernel objects can be created and destroyed at run-time. This provides a lot of flexibility for many types of applications. A server could for instance create a new task to handle each new session. When the session finishes, the task and all its resources can be returned to the system. The drawback to a completely dynamic system is the memory fragmentation problem. The memory is said to be fragmented when a request to allocate memory fails because there is no contiguous memory area large enough to satisfy the request, even though there is enough free memory in the memory heap. The dynamic memory allocation algorithm used by the rt-kernel RTOS is not particularly prone to memory fragmentation, but for some applications it can nevertheless be an issue. For this type of application the RTOS also provides a static memory heap.

An application using the static memory heap must create all resources (tasks, semaphores, etc) when the system starts up. No resources should be dynamically created when the system is running. Likewise, no resources should be destroyed. The static memory heap

will not reuse any memory that is returned to it. Fragmentation will not be an issue because once the system is running, no extra memory will be allocated from the heap. The heap can be further subdivided into pools suitable for allocation of fixed size messages. This allows dynamic allocation of data for signals and mailboxes even in an otherwise completely static systems. A default pool for signals is allocated when the system boots

## **Time-triggered Tasks**

Time-triggered tasks are defined statically according to a set of schedules provided by the user. It is not possible to create time-triggered tasks dynamically at run-time. However, it is possible to change schedules during run-time. Time-triggered tasks have priority over normal tasks and interrupts. They are started at the times given by the schedule. The schedule also defines a stopping point, or a deadline, by which time the task must have finished executing. The RTOS will call an error routine if a task misses its deadline. The schedule also defines a set of interrupts that are managed by the time-triggered subsystem. These time-triggered interrupts have priority over time-triggered tasks. Each time-triggered interrupt is allowed to fire only once after it has been activated, to stop a misfiring interrupt from damaging the performance of the system. It is then disabled until the next cycle or until it is explicitly re-enabled later in the schedule. The figure below shows the different processing levels of the RTOS when the time-triggered subsystem is being used.



**Figure 2.23 RTOS processing levels**

Conceptually, time-triggered tasks execute at priority level 31. Normal tasks should not use this priority level when time-triggered tasks are also being used.

### **13.Timers**

A timer is a function that is called by the RTOS kernel at intervals defined by the application. The timer function can be called periodically or one time only. The timer can be thought of as a simplified task that may be invoked periodically. The RTOS will call the timer function from interrupt context. This means that timers are subject to the same restrictions as interrupt service routines. In particular, the timer function must not call any function that may block, such as **sem\_wait()**, **mtx\_lock()**, and others. See **Interrupt Services** for further details on interrupt services. Timers offer a simple, low-overhead mechanism for periodic execution compared to tasks. However, a timer is more restricted than a task. Timers and tasks can work in unison; a timer can for instance perform some initial work and then signal a task to perform the remainder of the work from task context instead.

**Semaphores:** Semaphores are used for synchronisation in real-time systems. A semaphore is essentially a counter with atomic updates. The value of the counter determines if the semaphore is available. In order to proceed a task using the semaphore must first read, then write the counter. The RTOS guarantees that access to the counter is atomic. A semaphore can be used to guard access to shared resources. The semaphore is initialised to the number of resources it protects. This type of semaphore is called a counting semaphore. A task trying to take the semaphore will be blocked if the value of the counter is less than 1, indicating that there are no free resources, otherwise it will decrease the counter and proceed. When it has finished with the resource it signals the semaphore, and in doing so it increases the value of the counter and unblocks the first task that may have been blocked on the semaphore. A semaphore that can only have the values 1 and 0 is a binary semaphore and can be used to implement mutual exclusion, however mutexes are optimised for that type of operation and should be used instead.

**Mutexes:** Mutexes are binary semaphores optimised for mutual exclusion. They are typically used to guard a critical region in an application against simultaneous execution by multiple tasks.

**Flags:** Flags are used to synchronise a task to external events. Unlike semaphores, it is possible to wait for many events at the same time. A flag object is usually made up of 32 individual flags and the application can choose to wait for any combination of the flags to occur. (The number of flags in a flag object is equal to the number of bits in an integer for the current architecture. The size of an int is 32 bits on most architectures).

**Mailboxes:** Mailboxes are RTOS kernel objects that can hold messages to be delivered between tasks. Mailboxes have a finite size. The size is configured when the mailbox is created. A task that tries to post a message to a mailbox that is full will be blocked. A task that tries to fetch from a mailbox that is empty will also be blocked. A mailbox can hold any type of message. The message is just a pointer to a data structure. All tasks that access the mailbox

must agree on the representation of the data. The RTOS transfers the value of the pointer between the posting and fetching tasks. The message itself is not copied. The posting task must not use the message after posting it to the mailbox. The fetching task should free the message if it was dynamically allocated

**Signals:** Signals are messages that can be sent directly from task to task. Unlike mailboxes there is no need to provide a RTOS kernel object to hold undelivered messages. Signals can represent any kind of data structure. Each type of signal is associated with a number. The number is chosen by the application when the signal is created. The number should be unique, so that no two types of signals share the same number. When a task receives a signal it can decide what course of action to take based on the number identifying the signal type. Signals can be filtered. A task can choose to receive only certain types of signals. Signals that are sent to the task while the filter is being used will be kept in a queue, and can be received later. This mechanism can for instance be used in a subroutine to only deal with the types of signals that are of interest for the subroutine. Signals that were delivered while in the subroutine can be received by the main task when execution returns from the subroutine. The type of the signal must be defined so that the first member of the signal data structure is the signal number. This is the only information about the signal that is of interest to the RTOS. The number will be used to match against the filter if one has



been applied by the receiving task. The signal number 0 is used to terminate filter lists and is therefore reserved.

## **Error Detection**

Unlike many traditional embedded RTOS, rt-kernel services do not return error codes. Errors that are detected by the RTOS kernel are with few exceptions fatal errors. There is very little the application can do to handle the error gracefully. In a production system, the only possible course of action is often to reset the system and start over. When the RTOS detects an error, it calls a common error handler. The default error handler will halt the system in a busy loop. This is normally used during development, when a debugger is used to load and run code. If an error occurs, the debugger will be in the busy loop, the error code can be inspected, and the debugger backtrace function can be used to find out exactly where in the application the RTOS detected the error. Alternatively, the application can install its own error handler. The error handler can attempt to handle the error. For instance, if an out of memory error was detected, the application could attempt to free memory if it is known that some memory area can be safely deallocated. If there is no safe way to handle the error, the application should reset the system. The approach taken by the rt-kernel RTOS also has the beneficial side effect that the application does not have to check return values from RTOS services. This is a tedious and error-prone procedure that can lead to errors going undetected.

## **Interrupt Services**

Interrupt Service Routines (ISRs), are subroutines that are called by the RTOS kernel to handle interrupts. Each Interrupt Request Line (IRQ) can be mapped to an ISR that handles the interrupt. The rt-kernel embedded RTOS supports nested interrupts, i.e. interrupts of higher priority can preempt lower priority interrupts. A dedicated interrupt stack is used to store the state of nested interrupts. The application should call **int\_connect()** to install the interrupt service routine. The RTOS will store the address of the ISR in an internal table. When the interrupt occurs, the RTOS kernel will first swap out the currently running task, then call the ISR. Lower level interrupts are disabled while the ISR is running. To maintain a low interrupt latency for the system, it

is important that all interrupts are handled as quickly as possible. A common design pattern for complex peripherals is to let the ISR clear the interrupt source, then notify a task that handles the higher level processing of the interrupt.

## **Memory Allocation**

The rt-kernel embedded RTOS supports dynamic memory allocation from a heap. The standard C memory allocation functions `malloc()` and `free()` are supported. The `malloc` functions executes with interrupts locked and are therefore thread-safe. The heap is created when the system boots and will fill the available RAM.

## **Memory Pools**

The RTOS memory pools support allocation of fixed size messages. They are primarily intended for allocating signal and mailbox payloads, but can be used to allocate any object. The `sig_create()` call allocates signals from the default signal pool, which is created when the system boots. A memory pool can be used to allocate messages of up to 8 user-definable sizes. The buffer that is returned will be of the smallest available size that will hold the requested number of bytes.

## **Event Logging**

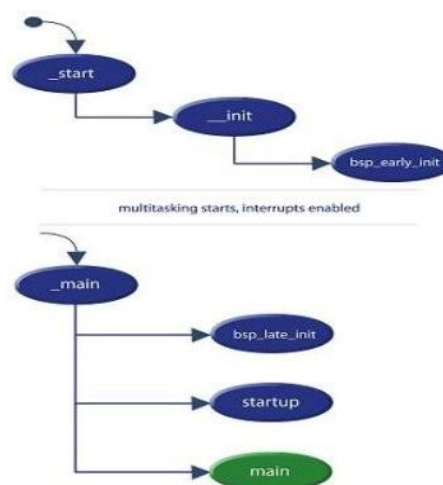
All RTOS kernel events can be logged. An event is in this context defined as any interaction with the RTOS kernel. Examples of events are tasks being swapped in and out, calling RTOS kernel services, interrupts occurring, etc. User-defined events can also be logged. Event logging is useful as a complement to traditional debugging tools. The event log offers insight into how the system behaves over a period of time, which traditional debuggers can not do. It should be noted that there is an overhead associated with collecting the log data. This may cause systems running under tight margins to behave differently when the events are being logged. Event logging can be completely disabled by setting the size of the event log buffer to zero. Event logging is always disabled initially, and must be enabled by calling `log_enable()`. Note that currently the rt-kernel RTOS is only shipped with event logging compiled in. This increases the size of the the RTOS libraries. Contact rt-labs if you have no need for event logging and prefer the space savings instead.

## Board Support Packages

The Board Support Package is responsible for configuring the board and initialising the RTOS. The BSP will normally contain an assembly file that sets up the board so that the RTOS can run, a timer driver, and a driver for the interrupt controller.

## Boot Sequence

The boot sequence is the time from power-on until the RTOS starts executing the first task



### Figure 3.24 boot sequence

The following events take place.

## Boot sequence

The target's reset vector should be mapped so that it starts to execute the function

\_start in the assembler file crt0.S. This function is responsible for setting up the embedded target to a point where it can execute C code. At a minimum, this consists of:

- Disabling interrupts
- Configuring a stack for the RTOS kernel

- Copying data section to RAM (if running from ROM)
- Clearing the BSS
- Jump to the RTOS kernel init function

### **Question Bank**

Write a short note about

- a) Time services b) Scheduling Mechanisms
2. a) Explain the overview of Threads and Tasks.
  - b) Draw the structure of Micro kernel and explain in brief.
3. a) Discuss in brief about the Interrupt services.
  - b) Mention the Importance of Memory management
4. Discuss the Communication and Synchronization issues.
5. a) Describe the Threads and Tasks functionality
  - b) Name some of the Scheduling mechanisms with an example.
6. Discuss how kernel plays an important role in the Operating systems
7. Write a short note about
  - a) Message Queue b) Message Priority Inheritance
8. Describe the Capabilities of commercial real time operating systems
9. a) Name the Features Real time operating Systems.
  - b) Define an Operating system? Specify the comparisons between General and Real time

### **TEXT / REFERENCE BOOKS**

1. Andrew N.Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guide: Designing & Optimizing System Software, Elsevier, 2004.
2. Jonathan W.Valvano, Embedded Microcomputer Systems: Real Time Interfacing, Cengage Learning, 2011
3. Wayne Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2008.
4. C.M.Krishna, Kang G.Shin, Real time systems, McGraw Hill, 3rd reprint, 2010.
5. Herma K., Real Time Systems: Design for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
6. William Hohl, ARM Assembly Language, Fundamentals and Techniques, Taylor & Francis, 2009.



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY  
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE  
[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING**  
**DEPARTMENT OF ELECTRONICS & INSTRUMENTATION**

## **UNIT-IV**

**EMBEDDED SYSTEM DESIGN— SBMA5201**

## **Embedded Networks**

Embedded Networks - Distributed Embedded Architecture – Hardware and Software Architectures, Networks for embedded systems– I2C, CAN Bus, Ethernet, Internet, Network–Based design– Communication Analysis, system performance Analysis, Hardware platform design, Allocation and scheduling, Design Example: Elevator Controller

### **1.Introduction**

In a distributed embedded system, several processing elements (PEs) (either microprocessors or ASICs) are connected by a network that allows them to communicate. The application is distributed over the PEs, and some of the work is done at each node in the network. There are several reasons to build network-based embedded systems. When the processing tasks are physically distributed, it may be necessary to put some of the computing power near where the events occur.

Consider, for example an automobile: the short time delays required for tasks such as engine control generally mean that at least parts of the task are done physically close to the engine.

Data reduction is another important reason for distributed processing. It may be possible to perform some initial signal processing on captured data to reduce its volume—for example, detecting a certain type of event in a sampled data stream. Reducing the data on a separate processor may significantly reduce the load on the processor that makes use of that data. Modularity is another motivation for network-based design. For instance, when a large system is assembled out of existing components, those components may use a network port as a clean interface that does not interfere with the internal operation of the component in ways that using the microprocessor bus would. A distributed system can also be easier to debug—the microprocessors in one part of the network can be used to probe components in another part of the network. Finally, in some cases, networks are used to build fault tolerance into systems. Distributed embedded system design is another example of hardware/software co-design, since we must design the network topology as well as the software running on the network nodes. Of course, the microprocessor bus is a simple type of network. However, we use the term network to mean an interconnection scheme that does not provide shared memory communication.

### **2.Distributed Embedded Architectures**

A distributed embedded system can be organized in many different ways, but its basic units are the PE and the network as illustrated in Figure 4.1. A PE may be an instruction set processor such as a DSP, CPU, or microcontroller, as well as a nonprogrammable unit such as the ASICs used to implement PE 4. An I/O device such as PE 1 (which we call here a sensor or actuator, depending on whether it provides input or output) may also be a PE, so long as it can speak the network protocol to communicate with other PEs. The network in this case is a bus, but other network topologies are also possible. It is also possible that the system can use more than one network, such as when relatively independent functions require relatively little communication among them. We often refer to the connection between PEs provided by the network as a communication link.

The system of PEs and networks forms the hardware platform on which the application runs. the distributed embedded system does not have memory on the bus (unless a memory unit is organized as an I/O device that speaks the network protocol). In particular, PEs do not fetch instructions over the network as they do on the microprocessor bus. We take advantage of this fact when analyzing network performance—the speed at which PEs can communicate over the bus would be difficult if not impossible to predict if we allowed arbitrary instruction and data fetches as we do on microprocessor buses

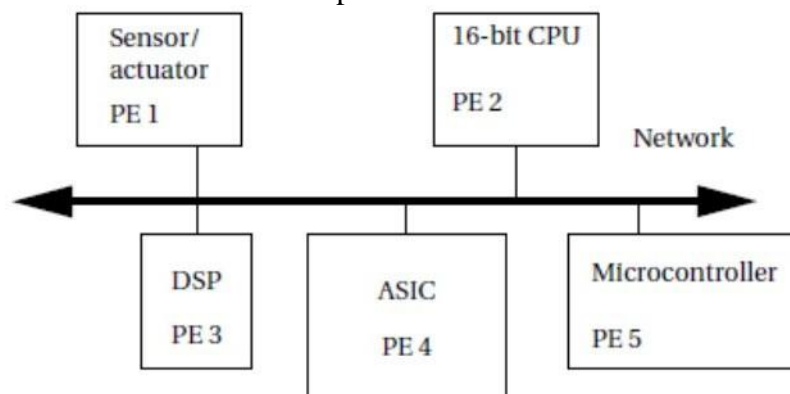


Figure 4.1 Distributed embedded system.

Building an embedded system with several PEs talking over a network is definitely more complicated than using a single large microprocessor to perform the same tasks. So why would anyone build a distributed embedded system? All the reasons for designing accelerator systems also apply to distributed embedded systems, and several more reasons are unique to distributed systems.

In some cases, distributed systems are necessary because the devices that the PEs communicate with are physically separated. If the deadlines for processing the data are short, it may be more cost-effective to put the PEs where the data are located rather than build a higher-speed network to carry the data to a distant, fast PE. An important advantage of a distributed system with several CPUs is that one part of the system can be used to help diagnose problems in another part. Whether you are debugging a prototype or diagnosing a problem in the field, isolating the error to one part of the system can be difficult when everything is done on a single CPU. If you have several CPUs in the system ,you can use one to generate inputs for another and to watch its output.

### 3.Network Abstractions

Networks are complex systems. Ideally, they provide high-level services while hiding many of the details of data transmission from the other components in the system. In order to help understand (and design) networks, the International Standards Organization has developed a seven- layer model for networks known as Open Systems Interconnection (OSI ) models [Sta97A]. Understanding the OSI layers will help us to understand the details of real networks.

The seven layers of the **OSI model**, shown in Figure 4.2, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one

or more layers because the higher layers may be totally missing or an intermediate layer may not be necessary. However, any data network should fit into the OSI model. The OSI layers from lowest to highest level of abstraction are described below.

- **Physical:**

The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), electrical properties, basic functions of the electrical and physical components, and the basic procedures for exchanging bits.

- **Data link:**

The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops, but only within a single hop.

Network:

This layer defines the basic end-to-end data transmission service. The network layer is particularly important in multihop networks.

- **Transport:**

The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links. This layer may also try to optimize network resource utilization.

- **Session:**

A session provides mechanisms for controlling the interaction of end user services across a network, such as data grouping and check pointing.

- **Presentation:**

This layer defines data exchange formats and provides transformation utilities to application programs.

- **Application:**

The application layer provides the application interface between the network and end-user programs.

Although it may seem that embedded systems would be too simple to require use of the OSI model, the model is in fact quite useful. Even relatively simple embedded networks provide physical, data link, and network services. An increasing number of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.

Application	End-use interface
Presentation	Data format
Session	Application dialog control
Transport	Connections
Network	End-to-end service
Data link	Reliable data transport
Physical	Mechanical, electrical



Figure 4.2 OSI model layers.

## 4. Hardware and Software Architectures

Distributed embedded systems can be organized in many different ways depending upon the needs of the application and cost constraints. One good way to understand possible architectures is to consider the different types of interconnection networks that can be used.

A **point-to-point** link establishes a connection between exactly two PEs. Point-to-point links are simple to design precisely because they deal with only two components. We do not have to worry about other PEs interfering with communication on the link.

Figure 4.3 A signal processing system built from point-to-point links shows a simple example of a distributed embedded system built from point-to-point links. The input signal is sampled by the input device and passed to the first digital filter, F1, over a point-to-point link. The results of that filter are sent through a second point-to-point link to filter F2. The results in turn are sent to the output device over a third point-to-point link. A digital filtering system requires that its outputs arrive at strict intervals, which means that the filters must process their inputs in a timely fashion. Using point-to-point connections allows both F1 and F2 to receive a new sample and send a new output at the same time without worrying about collisions on the communications network.

It is possible to build a full-duplex, point-to-point connection that can be used for simultaneous communication in both directions between the two PEs. (A half duplex connection allows for only one-way communication.) A bus is a more general form of network since it allows multiple devices to be connected to it. Like a microprocessor bus, PEs connected to the bus have addresses. Communications on the bus generally take the form of **packets** as illustrated in Figure 4.4. Format of a typical message on a bus.

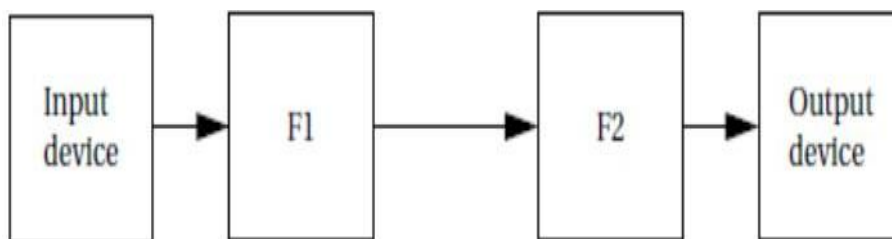
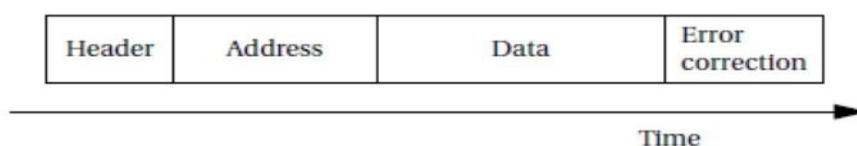


Figure 4.3 A signal processing system built from point-to-point links



**Figure 4. 4 Format of a typical message on a bus.**

A packet contains an address for the destination and the data to be delivered. It frequently includes error detection/correction information such as parity. It also may include bits that serve to signal to other PEs that the bus is in use, such as the header shown in the figure. The data to be transmitted from one PE to another may not fit exactly into the size of the **data payload** on the packet. It is the responsibility of the transmitting PE to divide its data into packets; the receiving PE must of course reassemble the complete data message from the packets.

Distributed system buses must be arbitrated to control simultaneous access, justas with microprocessor buses. Arbitration scheme types are summarized below.

- Fixed-priority arbitration always gives priority to competing devices in the same way. If a high- priority and a low-priority device both have long data transmissions ready at the same time, it is quite possible that the low-priority device will not be able to transmit anything until the high-priority device has sent all its data packets.

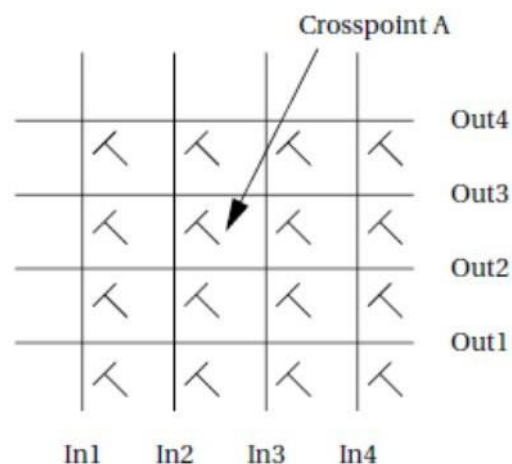
- Fair arbitration schemes make sure that no device is starved.

**Round-robin arbitration** is the most commonly used of the fair arbitration schemes The PCI bus requires that the arbitration scheme used on the bus must be fair, although it does not specify a particular arbitration scheme. Most implementations of PCI use round-robin arbitration.

A bus has limited available bandwidth. Since all devices connect to the bus, communications can interfere with each other. Other network topologies can be used to reduce communication conflicts .At

the opposite end of the generality spectrum from the bus is the **crossbar** network shown in Figure

4.5.A crossbar not only allows any input to be connected to any output, it also allows all combinations of input/output connections to be made.



**Figure 4. 5 A crossbar network.**

Thus, for example, we can simultaneously connect in1 to out4, in2 to out3, in3 to out2, and in4 to out1

or any other combinations of inputs. (Multicast connections can also be made from one input to several outputs.) A **crosspoint** is a switch that connects an input to an output. To connect an input to an output, we activate the cross point at the intersection between the corresponding input and output lines in the crossbar. For example, to connect in2 and out3 in the figure, we would activate crossbar A as shown. The major drawback of the crossbar network is expense: The size of the network grows as the square of the number of inputs (assuming the numbers of inputs and output sare equal).

Many other networks have been designed that provide varying amounts of parallel communication at varying hardware costs. Figure 6 shows an example **multistage network**. The crossbar of Figure 5 is a **direct network** in which messages go from source to destination without going through any memory element. Multistage networks have intermediate routing nodes to guide the data packets. Most networks are **blocking**, meaning that there are some combinations of sources and destinations for which messages cannot be delivered simultaneously.

A bus is a maximally blocking network since any message on the bus blocks messages from any other node. A crossbar is non-blocking. In general, networks differ from microprocessor buses in how they implement communication protocols. Both need handshaking to ensure that PEs do not interfere with each other. But in most networks, most of the protocol is performed in software. Microprocessors rely on bus hardware for fast transfers of instructions and data to and from the CPU. Most embedded network ports on microprocessors implement the basic communication functions (such as driving the communications medium) in hardware and implement many other operations in software.

An alternative to a non-bus network is to use multiple networks. As with PEs, it may be cheaper to use two slow, inexpensive networks than a single high performance, expensive network. If we can segregate critical and noncritical communications onto separate networks, it may be possible to use simpler topologies such as buses. Many systems use serial links for low-speed communication and CPU buses for higher speed and volume data transfers

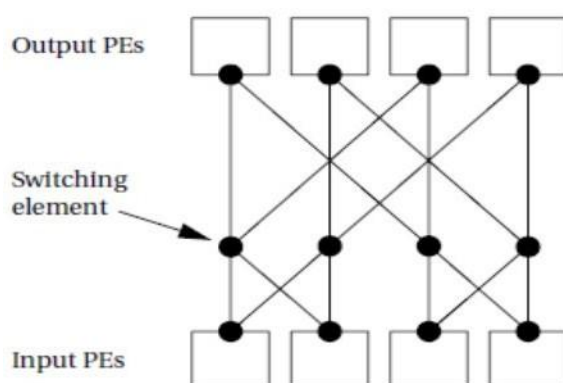


Figure 4.6 A multistage network.

## 5. Networks For Embedded Systems

Networks for embedded computing span a broad range of requirements; many of those requirements are very different from those for general-purpose networks. Some networks are used in safety-critical applications, such as automotive control. Some networks, such as those used in consumer electronics systems, must be very inexpensive. Other networks, such as industrial control networks, must be extremely rugged and reliable.

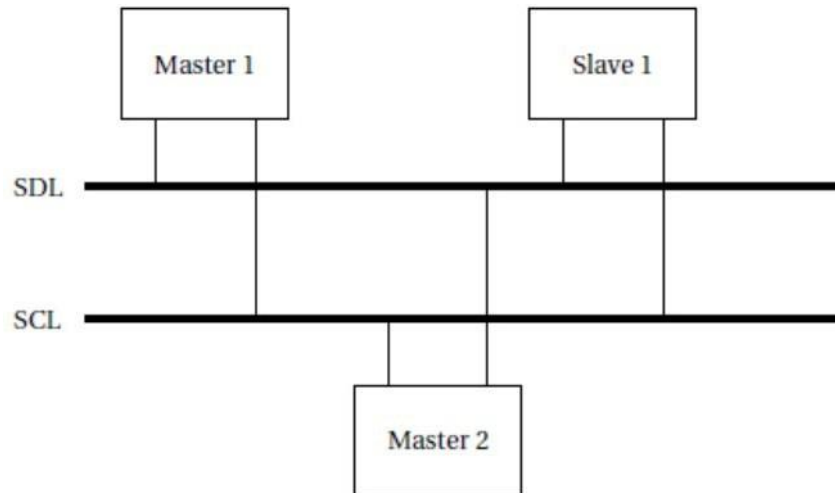
Several interconnect networks have been developed especially for distributed embedded computing:

- The I2C bus is used in microcontroller-based systems.
- The Controller Area Network (CAN) bus was developed for automotive electronics. It provides megabit rates and can handle large numbers of devices.
- Ethernet and variations of standard Ethernet are used for a variety of control applications.

### 5.1 The I2C Bus

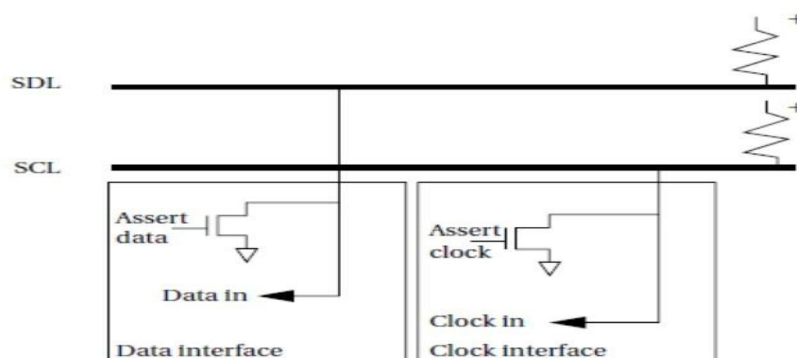
The I2C bus [Phi92] is a well-known bus commonly used to link microcontrollers into systems. It has even been used for the command interface in an MPEG-2 video chip [van97]; while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I2C bus interface. I2C is designed to be low cost, easy to implement, and of moderate speed (up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus). As a result, it uses only two lines: the serial data line (SDL) for data and the serial clock

line (SCL), which indicates when valid data are on the data line. Figure 4.7 shows the structure of a typical I2C bus system. Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus masters and the bus



**Figure 4.7 shows the structure of a typical I2C bus system.**

A pull-up resistor keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted. Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage. The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave. The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock (but not the high period) if necessary. The I2C bus is designed as a multi master bus—any one of several different devices may act as the master at various times. As a result, there is no global master to generate the clock signal on SCL. Instead, a master drives both SCL and SDL when it is sending data. When the bus is idle, both SCL and SDL remain high. When two devices try to drive either SCL or SDL to different values, the open collector/open drain circuitry prevents errors, but each master device must listen to the bus while transmitting to be sure that it is not interfering with another message—if the device receives a different value than it is trying to transmit, then it knows that it is interfering with another message.



**Figure 4.8 Electrical interface to the I2C bus.**

Every I2C device has an address. The addresses of the devices are determined by the system designer, usually as part of the program for the I2C driver. The addresses must of course be chosen so that no two devices in the system have the same address. A device address is 7 bits in the standard I2C definition (the extended I2C allows 10-bit addresses). The address 0000000 is used to signal a **general call** or bus broadcast, which can be used to signal all devices simultaneously. The address 11110XX is reserved for the extended 10-bit addressing scheme; there are several other reserved addresses as well.

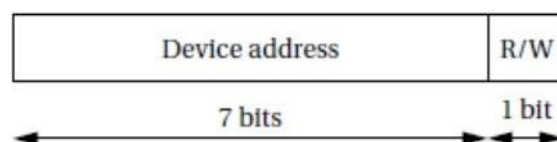
A **bus transaction** comprised a series of 1-byte **transmissions** and an address followed by one or more data bytes. I2C encourages a data-push programming style. When a master wants to write a slave, it transmits the slave's address followed by the data. Since a slave cannot initiate a transfer, the master must send a read request with the slave's address and let the slave transmit the data. Therefore, an address transmission includes the 7-bit address and 1 bit for data direction: 0 for writing from the master to the slave and 1 for reading from the slave to the master. (This explains the 7-bit addresses on the bus.) The format of an address transmission is shown in Figure 9.

A bus transaction is initiated by a start signal and completed with an end signal as follows:

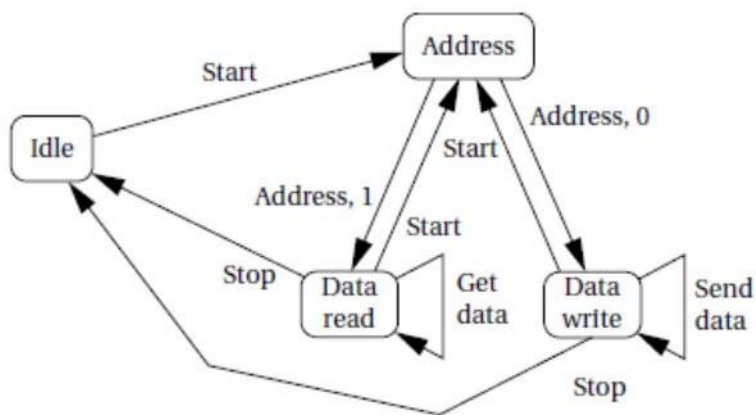
- A start is signalled by leaving the SCL high and sending a 1 to 0 transition on SDL.
- A stop is signalled by setting the SCL high and sending a 0 to 1 transition on SDL.

However, starts and stops must be paired. A master can write and then read(or read and then write) by sending a start after the data transmission, followed by another address transmission and then more data. The basic state transition graph for the master's actions in a bus transaction is shown in Figure 8.10. The formats of some typical complete bus transactions are shown in Figure 4.11. In the first example, the master writes 2 bytes to the addressed slave. In the second, the master requests a read from a slave. In the third, the master writes 1 byte to the slave, and then sends another start to initiate a read from the slave.

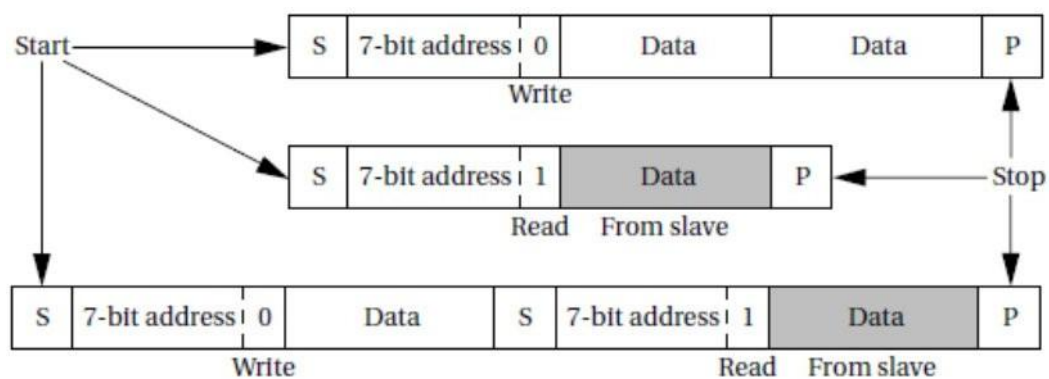
Figure 8.12 shows how a data byte is transmitted on the bus, including start and stop events. The transmission starts when SDL is pulled low while SCL remains high.



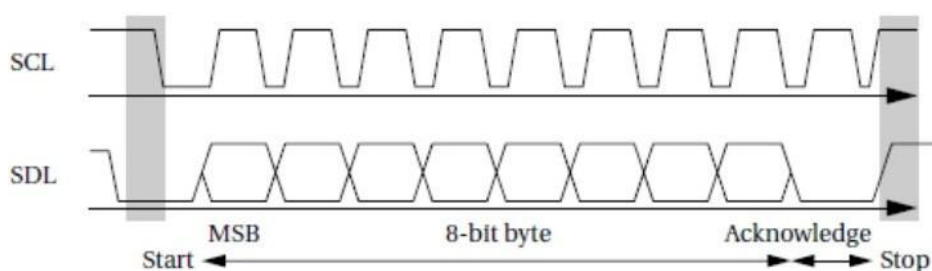
**Figure 4. 9 Format of an I2C address transmission.**



**Figure 4.10** State transition graph for an I2C bus master.



**Figure 4.11** Typical bus transactions on the I2C bus.



**Figure 4. 12** Transmitting a byte on the I2C bus.

After this start condition, the clock line is pulled low to initiate the data transfer. At each bit,

the clock line goes high while the data line assumes its proper value of 0 or 1. An acknowledgment is sent at the end of every 8-bit transmission, whether it is an address or data. For acknowledgment, the transmitter does not pull down the SDL, allowing the receiver to set the SDL to 0 if it properly received the byte. After acknowledgment, the SDL goes from low to high while the SCL is high, signalling the stop condition.

The bus uses this feature to arbitrate on each message. When sending, devices listen to the bus as well. If a device is trying to send a logic 1 but hears a logic 0, it immediately stops transmitting and gives the other sender priority. (The devices should be designed so that they can stop transmitting in time to allow a valid bit to be sent.) In many cases, arbitration will be completed during the address portion of a transmission, but arbitration may continue into the data portion. If two devices are trying to send identical data to the same address, then of course they never interfere and both succeed in sending their message.

The I2C interface on a microcontroller can be implemented with varying percentages of the functionality in software and hardware [Phi89]. As illustrated in Figure 13, a typical system has a 1-bit hardware interface with routines for byte level functions. The I2C device takes care of generating the clock and data. The application code calls routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth. One of the microcontroller's timers is typically used to control the length of bits on the bus. Interrupts may be used to recognize



bits. However, when used in master mode, polled I/O may be acceptable if no other pending tasks can be performed, since masters initiate their own transfers.

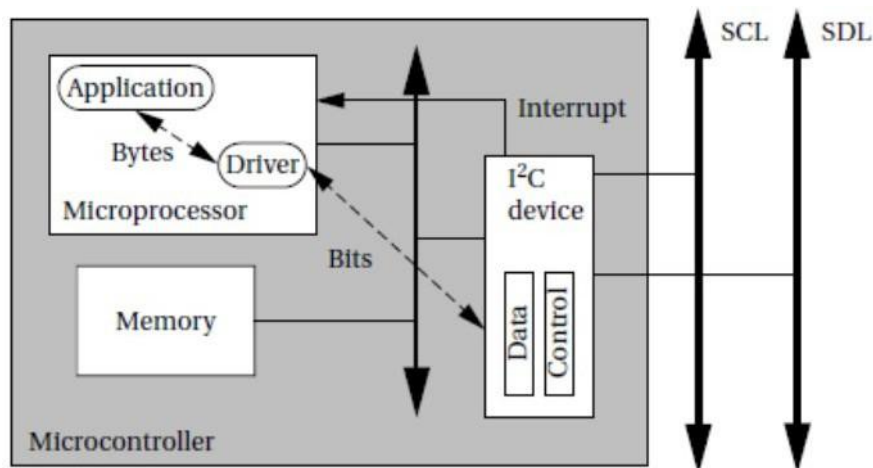


Figure 4.13 An I2C interface in a microcontroller.

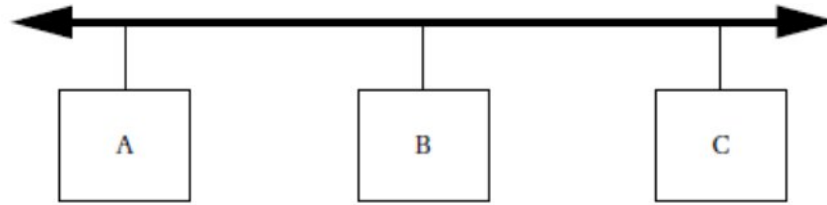
## 6.Ethernet

Ethernet is very widely used as a local area network for general-purpose computing. Because of its ubiquity and the low cost of Ethernet interfaces, it has seen significant use as a network for embedded computing. Ethernet is particularly useful when PCs are used as platforms, making it possible to use standard components, and when the network does not have to meet rigorous real-time requirements. The physical organization of an Ethernet is very simple, as shown in Figure 4.14.

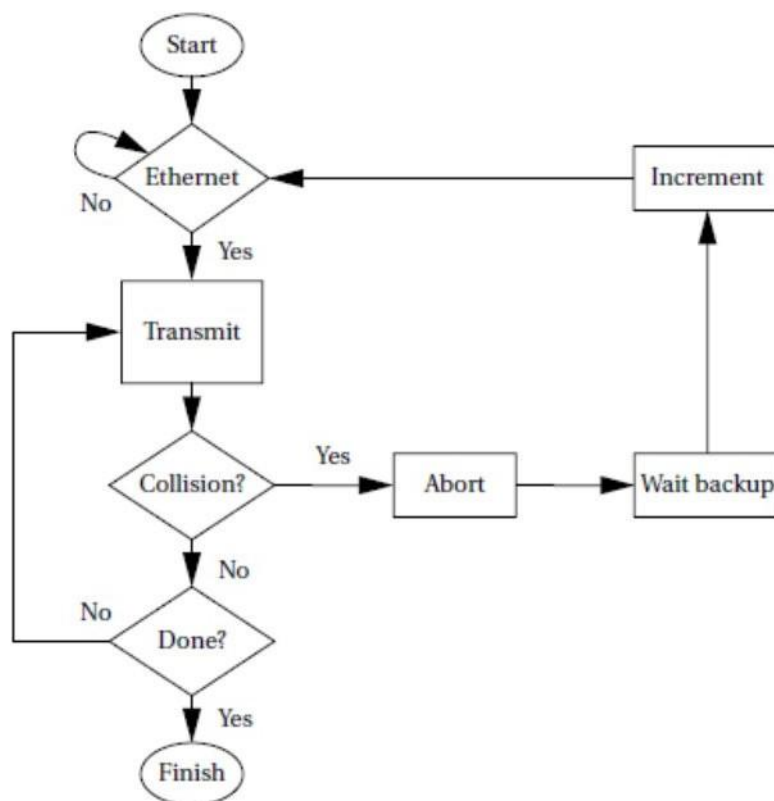
The network is a bus with a single signal path; the Ethernet standard allows for several different implementations such as twisted pair and coaxial cable. Unlike the I2C bus, nodes on the Ethernet are not synchronized—they can send their bits at any time. I2C relies on the fact that a collision can be detected and quashed within a single bit time thanks to synchronization. But since Ethernet nodes are not synchronized, if two nodes decide to transmit at the same time, the message will be ruined. The Ethernet arbitration scheme is known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

The algorithm is outlined in Figure 4.15. A node that has a message waits for the bus to become silent and then starts transmitting. It simultaneously listens, and if it hears another transmission that interferes with its transmission, it stops transmitting and waits to retransmit. The waiting time is random, but weighted by an exponential function of the number of times the message has been aborted. Figure 8.16 shows the exponential backoff function both before and after it is modulated by the random wait time. Since a message may be interfered with several times before it is successfully transmitted, the **exponential backoff** technique helps to ensure that the network does not become overloaded at high demand factors. The random factor in the wait time minimizes the chance that two messages will repeatedly interfere with each other. The maximum length of an Ethernet is determined by the nodes' ability to detect

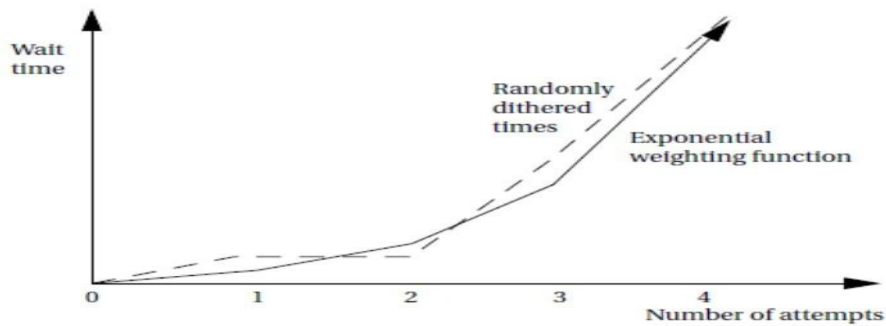
collisions. The worst case occurs when two nodes at opposite ends of the bus are transmitting simultaneously. For the collision to be detected by both nodes, each node's signal must be able to travel to the opposite end of the bus so that it can be heard by the other node. In practice, Ethernets up to several hundred meters.



**Figure 4.14 Ethernet organization.**



**Figure 4. 15The Ethernet CSMA/CD algorithm.**



**Figure 4.16 Exponential backoff times.**

Figure 4.17 shows the basic format of an Ethernet packet. It provides addresses of both the destination and the source. It also provides for a variable-length data payload. The fact that it may take several attempts to successfully transmit a message and that the waiting time includes a random factor makes Ethernet performance difficult to analyze. It is possible to perform data streaming and other real-time activities on Ethernets particularly when the total network load is kept to a reasonable level, but care must be taken in designing such systems.

Ethernet was not designed to support real-time operations; the exponential backoff scheme cannot guarantee delivery time of any data. Because so much Ethernet hardware and software is available

, many different approaches have been developed to extend Ethernet to real-time operation; some of these are compatible with the standard while others are not. As Decotignie points out [Dec05], there are three ways to reduce the variance in Ethernet's packet delivery time: suppress collisions on the network, reduce the number of collisions, or resolve collisions deterministically.

Felser [Fel05] describes several real-time Ethernet architectures

Preamble	Start frame	Destination address	Source address	Length	Data	Padding	CRC
----------	-------------	---------------------	----------------	--------	------	---------	-----

**Figure 4.17 Ethernet packet format.**

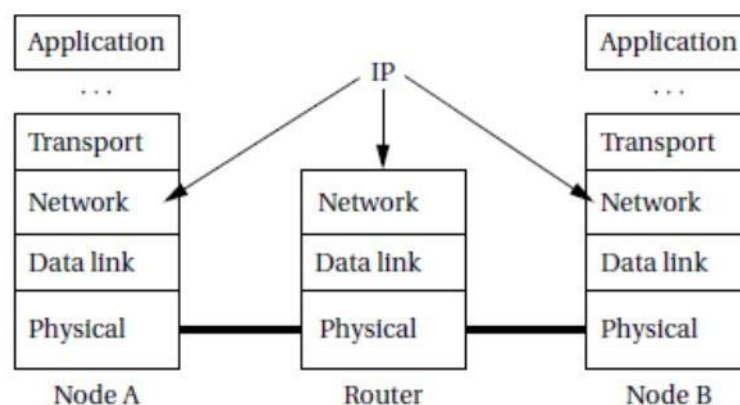
## 7. Internet

The Internet Protocol (IP) is the fundamental protocol on the Internet. It provides connectionless, packet-based communication. Industrial automation has long been a good application area for Internet-based embedded systems. Information appliances that use the Internet are rapidly becoming another use of IP in embedded computing. Internet protocol is not defined over a particular physical implementation—it is an internetworking standard. Internet packets are assumed to be carried by some other network, such as an Ethernet. In general, an Internet packet will travel over several different networks from source to destination. The IP allows data to flow seamlessly through these networks from one end user to another. The relationship between IP and individual networks is illustrated in Figure 19.

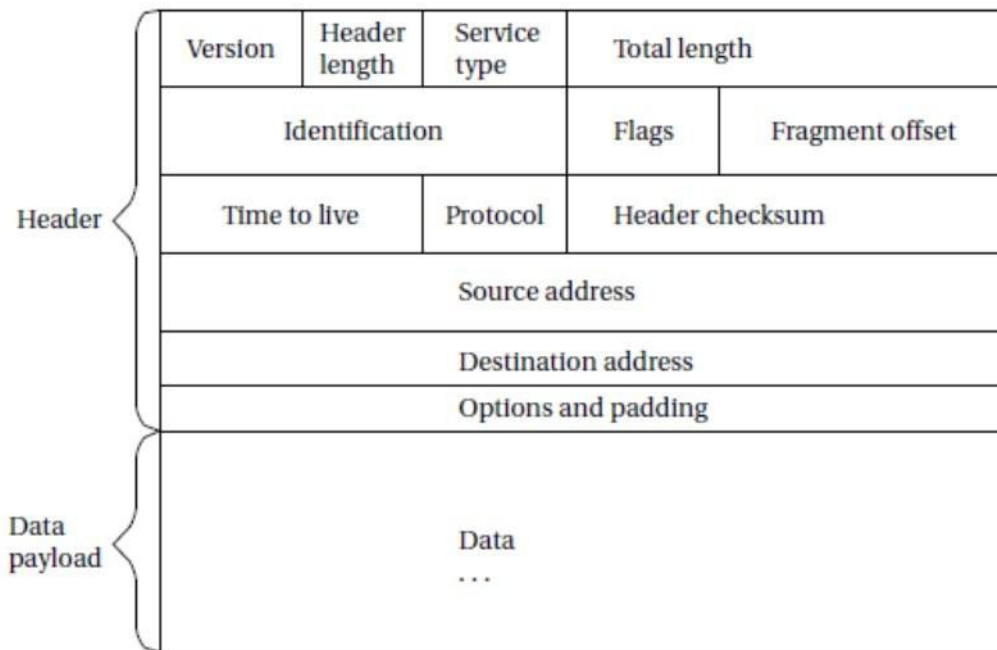
IP works at the network layer. When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to send to the IP. IP creates packets for routing to the destination, which are then sent to the data link and physical layers.

A node that transmits data among different types of networks is known as a **router**. The router's functionality must go up to the IP layer, but since it is not running applications, it does not need to go to higher levels of the OSI model. In general, a packet may go through several routers to get to its destination. At the destination, the IP layer provides data to the transport layer and ultimately the receiving application. As the data pass through several layers of the protocol stack, the IP packet data are encapsulated in packet formats appropriate to each layer. The basic format of an IP packet is shown in Figure 20. The header and data payload are both of variable length. The maximum total length of the header and data payload is 65,535 bytes.

An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xx.xx.xx. The names by which users and applications typically refer to Internet nodes.



**Figure 4.19 Protocol utilization in Internet communication.**



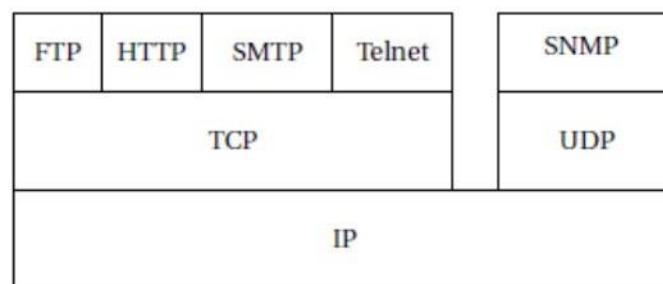
**Figure 4.20 IP packet structure.**

are translated into IP addresses via calls to a **Domain Name Server**, one of the higher-level services built on top of IP. The fact that IP works at the network layer tells us that it does not guarantee that a packet is delivered to its destination. Furthermore, packets that do arrive may come out of order. This is referred to as **best-effort routing**. Since routes for data may change quickly with subsequent packets being routed along very different paths with different delays, real-time performance of IP can

be hard to predict. When a small network is contained totally within the embedded system, performance can be evaluated through simulation or other methods because the possible inputs are limited. Since the performance of the Internet may depend on worldwide usage patterns, its real-time performance is inherently harder to predict. The Internet also provides higher-level services built on top of IP. The **Transmission Control Protocol (TCP)** is one such example. It provides a connection oriented service that ensures that data arrive in the appropriate order, and it uses an acknowledgment protocol to ensure that packets arrive. Because many higher level services are built on top of TCP, the basic protocol is often referred to as TCP. Thus, for example, we can simultaneously use IP.

Figure 4. 21 shows the relationships between IP and higher-level Internet services. Using IP as the foundation, TCP is used to provide File Transport Protocol for batch file transfers, Hypertext Transport Protocol (HTTP) for World Wide Web service, Simple Mail Transfer Protocol for email, and Telnet for virtual terminals. A separate transport protocol, User Datagram Protocol, is used as the basis for the network management services provided by the Simple Network

Management Protocol.



**Figure 4.21 The Internet service stack.**

## 8. Internet Applications

The Internet provides a standard way for an embedded system to act in concert with other devices and with users, such as:

- One of the earliest Internet-enabled embedded systems was the laser printer. High-end laser printers often use IP to receive print jobs from host machines.
- Portable Internet devices can display Web pages, read email, and synchronize calendar information with remote computers.
- A home control system allows the homeowner to remotely monitor and control home cameras, lights, and so on.

## The CAN bus

The **CAN bus** [Bos07] was designed for automotive electronics and was first used in production cars in 1991. CAN is very widely used in cars as well as in other applications.

The CAN bus uses bit-serial transmission. CAN runs at rates of 1 MB/s over a twisted pair connection of 40 m. An optical link can also be used. The bus protocol supports multiple masters on the bus. Many of the details of the CAN and I2C buses are similar, but there are also significant differences.

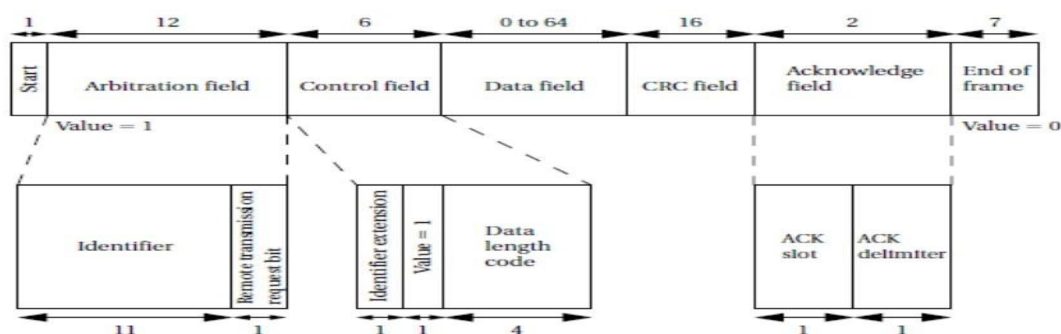
As shown in Figure 4.22, each node in the CAN bus has its own electrical drivers and receivers that connect the node to the bus in wired-AND fashion. In CAN terminology, a logical 1 on the bus is called **recessive** and a logical 0 is **dominant**. The driving circuits on the bus cause the bus to be pulled down to 0 if any node on the bus pulls the bus down (making 0 dominant over 1). When all nodes are transmitting 1s, the bus is said to be in the recessive state; when a node transmits a 0, the bus is in the dominant state. Data are sent on the network in packets known as **data frames**. CAN is a synchronous bus—all transmitters must send at the same time for bus arbitration to work. Nodes synchronize themselves to the bus by listening to the bit transitions on the bus. The first bit of a data frame provides the first synchronization opportunity in a frame. The nodes must also continue to synchronize themselves against later transitions in each frame.

For example, you cannot use an identifier to specify a device and provide a parameter to say which data value you want from that device. Instead, each possible data request must have its own identifier. An error frame can be generated by any node that detects an error on the bus. Upon detecting an error, a node interrupts the current transmission with an error frame, which consists of an error flag field followed by an error delimiter field of 8 recessive bits. The error delimiter field allows the bus to return to the quiescent state so that data frame transmission can resume. The bus also supports an overload frame, which is a special error frame sent during the interframe quiescent period.

.An overload frame signals that a node is overloaded and will not be able to handle the next message. The node can delay the transmission of the next frame with up to two overload frames in a row, hopefully giving it enough time to recover from its overload. The CRC field can be used to check a message's data field for correctness .If a transmitting node does not receive an acknowledgment for a data frame, it should retransmit the data frame until the frame is acknowledged. This action corresponds to the data link layer in the OSI model.

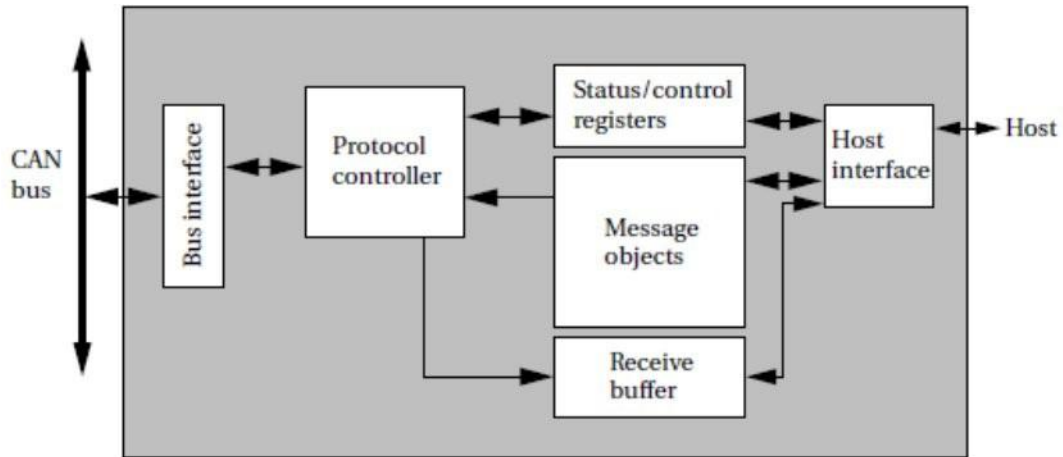
Figure 4.24 shows the basic architecture of a typical CAN controller. The controller implements the physical and data link layers; since CAN is a bus, it does not need network layer services to establish end-to-end connections. The protocol control block is responsible for determining when to send messages, when a message must be resent due to arbitration losses, and when a message should be received. The FlexRay network has been designed as the next generation of system buses for cars. FlexRay provides high data rates—up to 10 MB/s—with deterministic communication. It is also designed to be fault –tolerant .The Local Interconnect Network ( LIN) bus [Bos07] was created to connect components in a small area, such as a single door. The physical medium is a single wire that provides data rates of up to 20 KB/s for up to 16 bus subscribers. All transactions are initiated by the master and responded to by a frame. The software for the network is often generated from a LIN description file that describes the network subscribers, the signals to be generated, and the frames. Several buses have come into use for passenger entertainment. Bluetooth is becoming the standard mechanism for cars to interact with consumer electronics devices such as audio players or phones. The Media Oriented Systems Transport(MOST) bus [Bos07] was designed for entertainment and multimedia information.

The basic MOST bus runs at 24.8 MB/s and is known as MOST 25; 50 and 150 MB/s versions have also been developed. MOST can support up to 64 devices. The network is organized as a ring. Data transmission is divided into channels. A control channel transfers control and system management data. Synchronous channels are used to transmit multimedia data; MOST 25 provides up to 15 audio channels. An asynchronous channel provides high data rates but without the quality-of- service guarantees of the synchronous channels



**Figure 4.23** The CAN data frame format.





**Figure 4.24** Architecture of a CAN controller.

## 9.NETWORK-BASED DESIGN

Designing a distributed embedded system around a network involves some of the same design tasks we faced in accelerated systems. We must schedule computations in time and allocate them to PEs. Scheduling and allocation of communication are important additional design tasks required for many distributed networks. Many embedded networks are designed for low cost and therefore do not provide excessively high communication speed. If we are not careful, the network can become the bottleneck in system design. In this section we concentrate on design tasks unique to network-based distributed embedded systems.

## 10.COMMUNICATIONAL ANALYSIS

We know how to analyze the execution time of programs and systems of processes on single CPUs, but to analyze the performance of networks we must know how to determine the delay incurred by transmitting messages. Let us assume for the moment that messages are sent reliably—we do not have to retransmit a message.

The **message delay** for a single message with no contention (as would be the case in a point-to-point connection) can be modeled as

$$t_m = t_x + t_n + t_r$$

where  $t_x$  is the transmitter-side overhead,  $t_n$  is the network transmission time, and  $t_r$  is the receiver-side overhead. In I2C,  $t_x$  and  $t_r$  are negligible relative to  $t_n$ , as illustrated by Example .

### Simple message delay for an I<sup>2</sup>C message

Let's assume that our I<sup>2</sup>C bus runs at the rate of 100 KB/s and that we need to send one 8-bit byte. Based on the message format shown in Figure 8.9, we can compute the number of bits in the complete packet:

$$\begin{aligned}n_{\text{packet}} &= \text{startbit} + \text{address} + \text{data} + \text{stopbit} \\ &= 1 + 8 + 8 + 1 = 18 \text{ bits}\end{aligned}$$

The time required, then, to transmit the packet is

$$t_n = n_{\text{packet}} \times t_{\text{bit}} = 1.8 \times 10^{-4} \text{ s.}$$

Some of the instructions in the transmitter and receiver drivers—namely, the loops that send bytes to and receive bytes from the network interface—will run concurrently with the message transmission. If we assume that 20 instructions outside of these loops are executed by the transmitter and receiver, overheads on an 8 MHz microcontroller would be as follows:

$$t_x = t_r = 20 \times 0.125 \times 10^{-6} = 2.5 \times 10^{-6}.$$

The total message delay is:

$$t_m = 2.5 \times 10^{-6} + 1.8 \times 10^{-4} + 2.5 \times 10^{-6} = 1.85 \times 10^{-4}.$$

Overhead is <3% of the total message time in this case.

---

If messages can interfere with each other in the network, analyzing communication delay becomes difficult. In general, because we must wait for the network to become available and then transmit the message, we can write the **message delay** as

$$t_y = t_d + t_m$$

where  $t_d$  is the **network availability delay** incurred waiting for the network to become available. The main problem, therefore, is calculating  $t_d$ . That value depends on the type of arbitration used in the network.

- If the network uses fixed-priority arbitration, the network availability delay is unbounded for all but the highest-priority device. Since the highest-priority device always gets the network first, unless there is an application-specific limit on how long it will transmit before relinquishing the network, it can keep blocking the other devices indefinitely.
- If the network uses fair arbitration, the network availability delay is bounded

In the case of round-robin arbitration, if there are  $N$  devices, then the worst case network availability delay is  $N(t_x + t_{\text{arb}})$ , where  $t_{\text{arb}}$  is the delay incurred for arbitration.  $t_{\text{arb}}$  is usually small compared to transmission time. Even when round-robin arbitration is used to bound the network availability delay, the waiting time can be very long. If we add acknowledgment and data corruption into the analysis, figuring network delay is more difficult. Assuming that errors are random, we cannot predict a worst-case delay since every packet may contain an error. We can, however, compute the probability that a packet will be delayed for more than a given amount of time. However, such analysis is beyond the scope of this book.

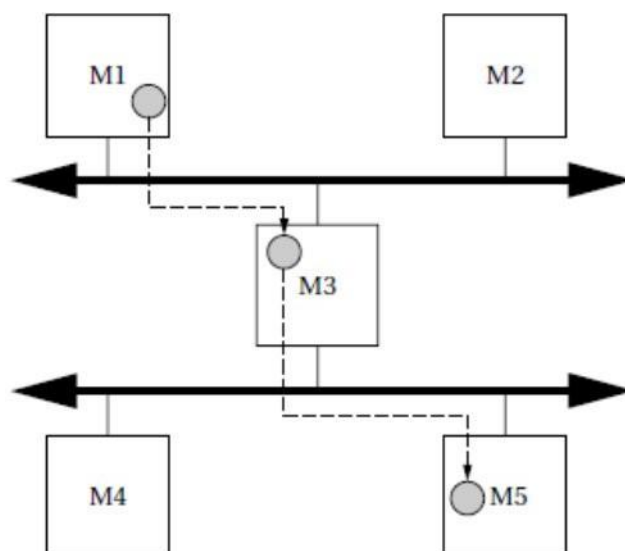
Arbitration on networks is a form of prioritization. In a rate-monotonic communication scheme, the task with the shortest deadline should be assigned the highest priority in the network. Our process scheduling model assumed that we could interrupt processes at any point. But network communications are organized into packets. In most

networks we cannot interrupt a packet transmission to take over the network for a higher priority packet. When a low-priority message is on the network the network is effectively allocated to that low-priority message, allowing it to block higher-priority messages.

This cannot cause deadlock since each message has a bounded length, but it can slow

down critical communications. The only solution is to analyze network behaviour to determine whether priority inversion causes some messages to be delayed for too long. Of course, a round-robin arbitrated network puts all communications at the same priority. This does not eliminate the priority inversion problem because processes still have priorities. Thus far we have assumed a **single-hop network**: A message is received at its intended destination directly from the source, without going through any other network node. It is possible to build **multihop networks** in which messages are routed through network nodes to get to their destinations. (Using a multistage network does not necessarily mean using a multihop network—the stages in a multistage network are generally much smaller than the network PEs.)

Figure 4.18 shows an example of a multihop communication. The hardware platform has two separate networks( perhaps so that communications between subsets of the PEs do not interfere),but there is no direct path from M1 to M5.The message is therefore routed through M3,which reads it from one network and sends it on to the other one.



**Figure 4.18 A multihop communication.**

Analyzing delays through multihop systems is very difficult. For example, the time that the message is held at M3 depends on both the computational load of M3 and the other messages that it must handle

.If there is more than one network we must allocate communications to the networks. We may establish multiple networks so that lower-priority communications can be handled separately without interfering with high-priority communications on the primary network. Scheduling and allocation of computations and communications are clearly interrelated. If we change the allocation of computations, we change not only the scheduling of processes on

those PEs but also potentially the schedules of PEs with which they communicate. For example, if we move a computation to a slower PE, its results will be available later, which may mean rescheduling both the process that uses the value and the communication that sends the value to its destination.

## 11. Hardware Platform Design, Allocation and Scheduling

Now that we know how to compute delay for messages we can develop strategies for designing the schedule and allocation of process and communication . designing the hardware platform is necessarily closely related to our choice in scheduling and allocating processes .we want to use only as much hardware as is necessary , but we cannot know how much hardware to use until we can construct a system schedule . Creating that schedule requires an allocation of process to PEs , which in turn requires knowing the available hardware .

When designing the hardware platform, we have the following design choice to make :

- $\Sigma$  Number of PEs required
- $\Sigma$  Types of all PEs
- $\Sigma$  Number of network required
- $\Sigma$  Types (and data rates) of the networks

In making these choices , we need to construct allocations and schedules for the processes to evaluate the platform. In turn , allocation and scheduling are driven by system performance analysis.

It helps to start with a basic assessment of the computation and communication needs of the system. A lower bound on the computational needs of the system can be obtained by summing up the worst case execution times of the processes

Where  $T_{pi}$  is the execution time of process  $P_i$  and  $T_l$  is the least – common multiple of all the periods  $T_i$  . This formula computes the total execution time over the schedule unrolled to the least-common multiple periods .Similarly, we can compute the communication volume over the least – common multiple of the periods .

The above formula computes the total number of bytes transmitted in the unrolled schedule by counting the output bytes of all the processes in the system . Of course , these figures do not account for overheads such as operating system scheduling or communication interference . They simply provide lower bounds on our needs .

Depending on the types of system we are designing , the following two strategies may be useful to help as quickly come up with efficient system :

For I/O intensive systems we will start with the I/O devices and their associated processing .

For computation – intensive systems we will start with the processes . For systems that do a lot of I/O

, we definitely need to support I/O devices themselves and perhaps do some processing of the data locally before shipping the data over the network .

- Σ Inventory the required I/O devices
- Σ Determine which processing has deadlines short enough that they cannot be met by any network within your price range . I/O devices that do not require local processing may be attached to the network with simplest available interface .
- Σ Determine which devices can share a processing element or network interface .
- Σ Analyze communication times to determine whether critical communications may interfere with each other . Determine whether a complex network or multiple networks may be required to satisfy communication deadlines
- Σ Allocate the minimum required PE to go with each I/O device
- Σ Design the rest of the system using the procedure for computation intensive systems

### 11.1 Computation intensive system design

For computation – intensive systems , we want to consider the process and their deadlines and communication as follows :

- Σ Start with the tasks with the shortest deadlines . The shorter the deadline for a task , the more likely it is to require its own processing element or elements . If a high priority task shares a PE with a low – priority task not only will a more expensive PE be required , but scheduling overhead will be paid for at the non linear rate .
- Σ Analyze communication times to determine whether critical communications may interfere with each other
- Σ Allocate lower – priority tasks to shared PEs where possible

After we have designed a basic system that meets our performance goals , we can improve it to satisfy power consumption or other requirements .

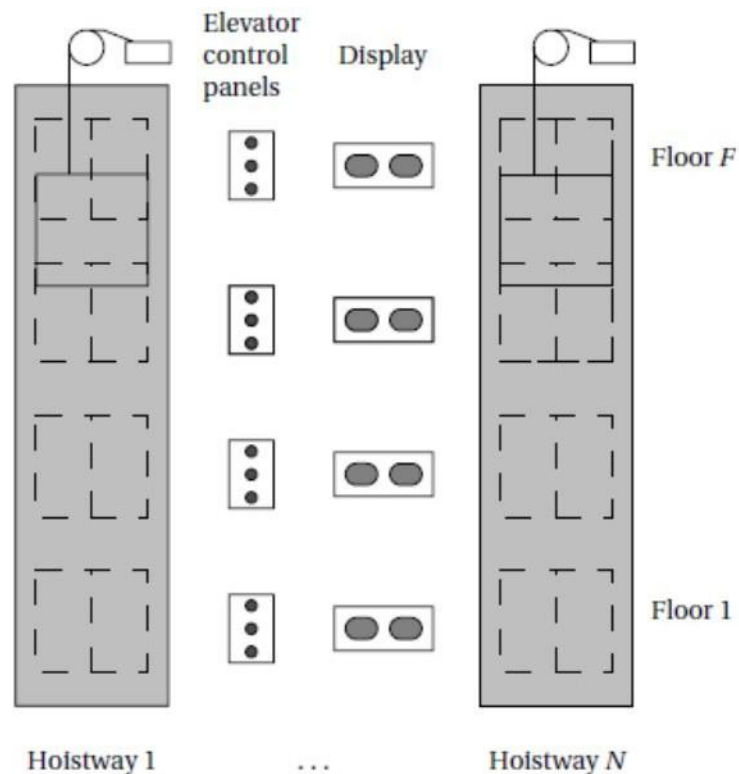
Once you have an initial allocation , use the system schedule as a guide for fine tuning . By reallocating processes you may be able to improve one or more attributes , such as hardware cost , slack time in the schedule , power consumption and so on . In particular , load balancing is often a good idea . If you have some PEs that are more heavily loaded than others , it may be possible to move some of those processes to other PEs . Doing so can reduce the chance that the system fails to meet a deadline due to mistaken estimation of run times

## **11.2ELEVATOR CONTROLLER**

We will use the principles of distributed system design by designing an elevator controller. The components are physically distributed among the elevators and floors of the building, and the system must meet both hard (making sure the elevator stops at the right point) and soft (responding to requests for elevators) deadlines.

### **11.3 Theory of Operation and Requirements**

We design a multiple elevator system to increase the challenge. The configuration of a bank of elevators is shown in Figure 4.25. The elevator car is the unit that runs up and down the hoistway (also known as the shaft) carrying passengers; we will use  $N$  to represent the number of hoistways. Each car runs in a hoist way and can stop at any of  $F$  floors. (For convenience we will number the floors 1 through  $F$ , although some of the elevator doors may in fact be in the basement.) Every elevator car has a car control panel that allows the passengers to select floors to stop at. Each floor has a single floor control panel that calls for an elevator. Each floor also has a set of displays to show the current state of the elevator systems. The user interface consists of the elevator control panels, floor control panels, and displays. The car control panels have  $F$  buttons to request the floors plus an emergency stop button. Each floor control panel has an up button and a down button that request an elevator going in the chosen direction. There is one display



**Figure 4.19 A bank of elevators.**

per hoistway on each floor. Each display has an up light and a down light; if the elevator is idle, neither light is on. The displays for a hoistway always show the same state on all floors.

The elevator control system consists of two types of components.

- Σ First, a single master controller governs the overall behavior of all elevators, and
- Σ second, on each elevator a car controller runs everything that must be done within the car.

The car controller must of course sense button presses on the car control panel, but it must also sense the current position of the elevator.

As shown in Figure 8.26, the car controller reads two sets of indicators on the wall of the elevator hoistway to sense position.

The coarse indicators run the entire length of the hoistway and a sensor determines when the elevator passes each one.

Fine indicators are located only around the stopping point for each floor. There are  $2S - 1$  fine indicators on each floor, one at the exact stopping point and  $S$  on each side of it. The sensor also reads fine indicators; it puts out separate signals for the coarse and fine indicators. The elevator system can stop at the proper position by counting coarse and fine indicators.

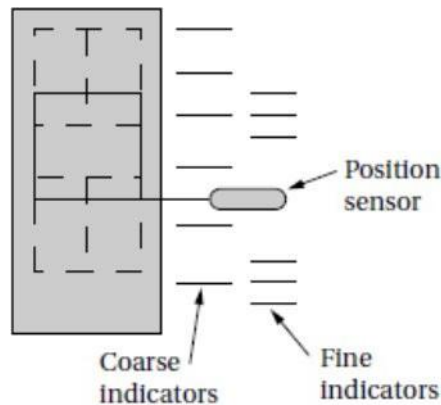
The elevator's movement is controlled by two motor control inputs: one for up and one for down. When both are disabled, the elevator does not move. The system should not enable both up and down on a single hoistway simultaneously. The master controller has several tasks—it must read inputs from the floor control panels, send signals to the lights on the floor displays, read floor requests from the car controllers, and take inputs from the car

sensors. Most importantly, it must tell the elevators when to move and when to stop. It must also schedule the elevators to efficiently answer passenger requests.

The basic requirements for the elevator system follow

Name	Elevator system
Inputs	$F$ floor control inputs, $N$ position sensors, $N$ car control panels, one master control panel
Outputs	$F$ displays, $N$ motor controllers
Functions	Responds to floor, car, and master control panels; operates cars safely
Performance	Control of elevators is time critical
Manufacturing cost	Cost of electronics is small compared to mechanical systems
Power	Not important
Physical size and weight	Cabling is the major concern

In this design ,we are much more aware of the surrounding mechanical elements than we have been in previous examples. The electronics are clearly a small part of the cost and bulk of the elevator system. But because the elevators are controlled by the computers, the proper operation of the embedded hardware and software is very important.



**Figure 4.20 Sensing elevator position.**



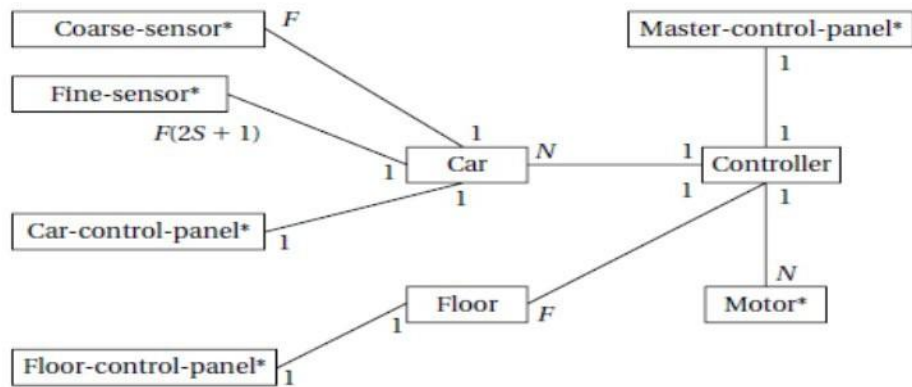
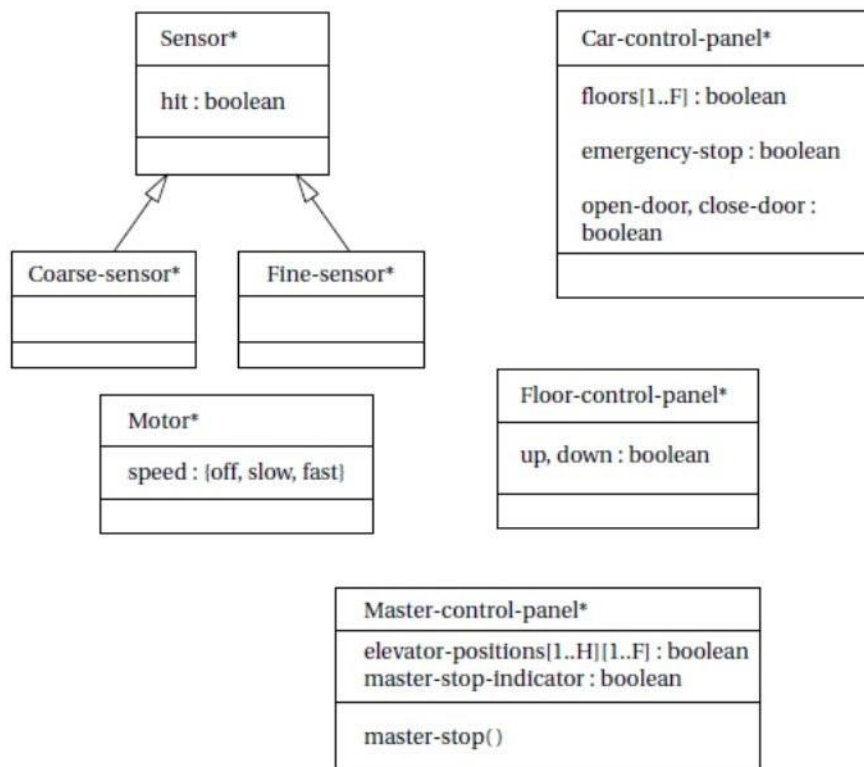


Figure 4.21 Basic class diagram for the elevator system.

## Specification

The basic class diagram for the elevator system is shown in Figure 4.27. This diagram concentrates on the relationships among the classes and the number of objects of each type that the system requires.

The physical interface classes are defined in more detail in Figure 4.28. We have used inheritance to define the sensors, even though these classes represent physical objects. The only difference among the sensors to the elevator controller is whether they indicate coarse or fine positions; other physical distinctions among the sensors do not matter.



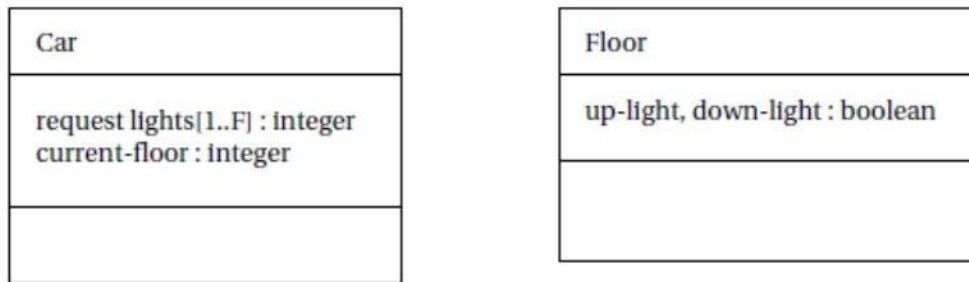
**Figure 4.22** Physical interface classes for the elevator system.

The Car and Floor classes, which describe the control panels on the floors and in the cars, are shown in Figure 4.29. These classes define the basic attributes of the car and floor control panels.

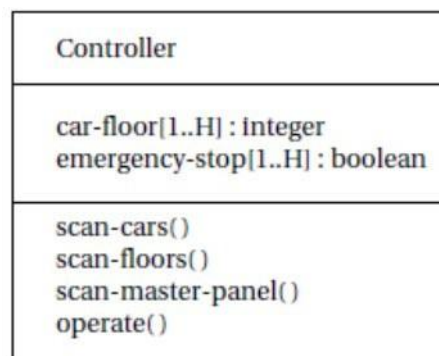
The Controller class is defined in Figure 4.30. This class defines attributes that describe the state of the system, including where each car is and whether the system has made an emergency stop. It also defines several behaviors, such as an operate behavior and behaviors to check the state of parts of the system

## Architecture

Computation and I/O occur at three major locations in this system: the floor control panels/displays, the elevator cabs, and the system controller. Let's consider the basic operation of each of these subsystems one at a time and then go back and design the network that connects them



**Figure 4.23** The Car and Floor classes.



**Figure 4.24** The Controller class for the elevator system.

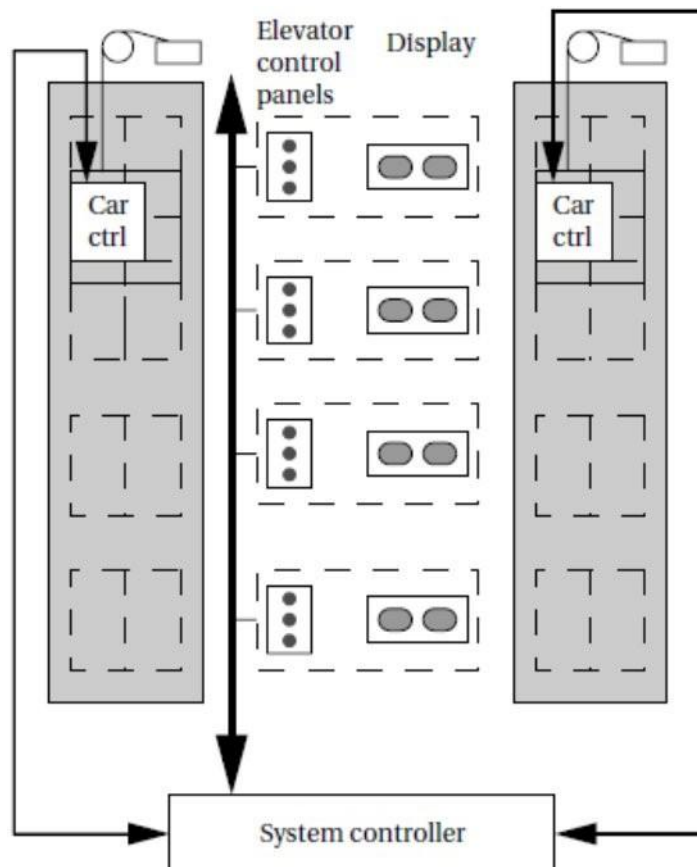
The floor control panels and displays are relatively simple since they have no hard real-time requirements. Each one takes a set of inputs for the up/down indicators and lights the appropriate lights. Each also watches for button events and sends the results to the system controller. We can use a simple microcontroller for all these tasks.

The cab controller must read the cab's buttons and send events to the system controller. It must also read the sensor inputs and send them to the system controller. Reading the sensors is a hard real-time task—proper operation of the elevator requires that the cab controller not miss any of the indicators. We have to decide whether to use one or two PEs in the cab. A conservative design would use separate PEs for the button panel and the sensor. We could also use a single processor to handle both the buttons and the sensor.

The system controller must take inputs from all these units. Its control of the elevators has both hard and soft real-time aspects: It must constantly monitor all moving elevators to be sure they stop properly, as well as choose which elevator to dispatch to a request. Figure 8.31 shows the set of networks we will use in the system. The floor control panels/displays are connected along a single bus network. Each elevator car has its own point-to-point link with the system controller

## 12. Testing

The simplest way to test the controllers is to build an elevator simulator using an FPGA. We can easily program an FPGA to simulate several elevators by keeping registers for the current position of each elevator and using counters to control how often the elevators change state. Using an FPGA-based elevator simulator provides good motivation for this example because we can design the FPGA to indicate when an elevator has crashed through the floor or the ceiling of its shaft. Working with a real-time-oriented elevator simulator helps illustrate the challenges presented by real-time control. We can use a serial link from a PC to provide button inputs, or we can wire up panels of buttons and indicators ourselves.



**Figure 4.25** The networks in the elevator

## **Questions Bank**

1. Explain in detail the design process in embedded system.
2. What are the challenges of Embedded System
3. Discuss the complete design of typical embedded system
4. What do you mean by embedded systems, Explain in brief with an example
5. Discuss the different challenges related to embedded software development
6. Describe the different issues related to embedded software development
7. Explain the different models and languages for embedded software

## **TEXT / REFERENCE BOOKS**

1. Andrew N.Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guide: Designing & Optimizing System Software, Elsevier, 2004.
2. Jonathan W.Valvano, Embedded Microcomputer Systems: Real Time Interfacing, Cengage Learning, 2011
3. Wayne Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2008.
4. C.M.Krishna, Kang G.Shin, Real time systems, McGraw Hill, 3rd reprint, 2010.
5. Herma K., Real Time Systems: Design for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
6. William Hohl, ARM Assembly Language, Fundamentals and Techniques, Taylor & Francis, 2009.



**SATHYABAMA**

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

[www.sathyabama.ac.in](http://www.sathyabama.ac.in)

**SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING**  
**DEPARTMENT OF ELECTRONICS & INSTRUMENTATION**

## **UNIT-V**

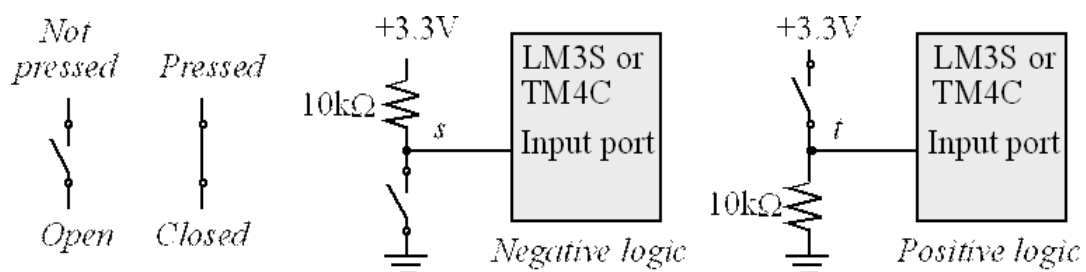
**EMBEDDED SYSTEM DESIGN— SBMA5201**

## SYSTEM DESIGN

Switches and LED interfacing-LCD Display interfacing- Analog sensors interfacing for digital data conversion - Access control using analog keypad - Pulse width modulation technique for motor speed control

### 1.Switch Interfaces

Input/output devices are critical components of an embedded system. The first input device we will study is the **switch**. It allows the human to input binary information into the computer. Typically we define the asserted state, or logic true, when the switch is pressed. Contact switches can also be used in machines to detect mechanical contact (e.g., two parts touching, paper present in the printer, or wheels on the ground etc.) A single pole single throw (SPST) switch has two connections. The switches are shown as little open circles in Figure 8.2. In a normally open switch (NO), the resistance between the connections is infinite (over 100 M $\Omega$  on the B3F tactile switch) if the switch is not pressed and zero (under 0.1  $\Omega$  on the B3F tactile switch) if the switch is pressed. To convert the infinite/zero resistance into a digital signal, we can use a pull-down resistor to ground or a pull-up resistor to +3.3V as shown in Figure 8.2. Notice that 10 k $\Omega$  is 100,000 times larger than the on-resistance of the switch and 10,000 times smaller than its off-resistance. Another way to choose the pull-down or pull-up resistor is to consider the input current of the microcontroller input pin. The current into the microcontroller will be less than 2 $\mu$ A (shown as  $I_{IL}$  and  $I_{IH}$  in the data sheet). So, if the current into microcontroller is 2 $\mu$ A, then the voltage drop across the 10 k $\Omega$  resistor will be 0.02 V, which is negligibly small. With a pull-down resistor, the digital signal will be low if the switch is not pressed and high if the switch is pressed (right Figure 8.2). This is defined as **positive logic** because the asserted state is a logic high. Conversely, with a pull-up resistor, the digital signal will be high if the switch is not pressed and low if the switch is pressed (middle of Figure 8.2). This is defined as **negative logic** because the asserted state is a logic low.

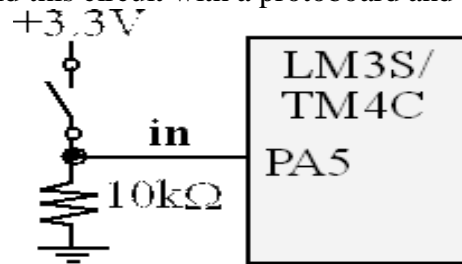


**Figure 5.1 Single Pole Single Throw (SPST) Switch interface.**

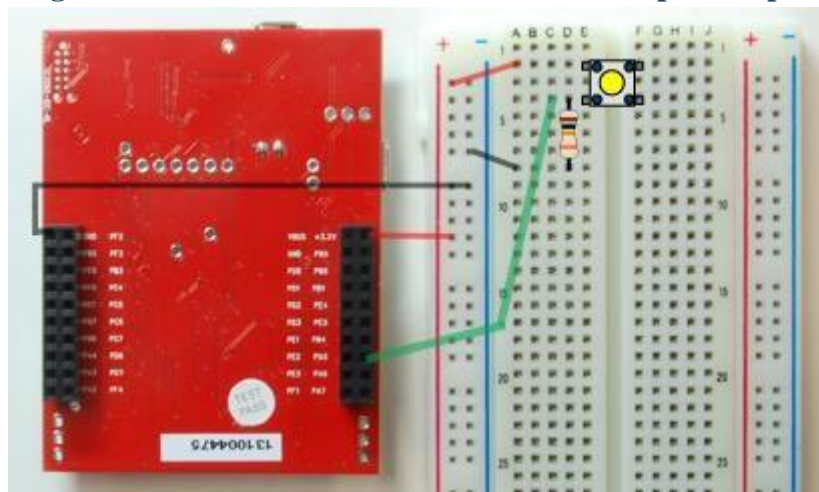
One of the complicating issues with mechanical switches is they can bounce (oscillate on and

off) when touched and when released. The contact bounce varies from switch to switch and from time to time, but usually bouncing is a transient event lasting less than 5 ms. We can eliminate the effect of bounce if we design software that waits at least 10 ms between times we read the switch values.

To interface a switch we connect it to a pin (e.g., Figure 8.3) and initialize the pin as an input. The initialization function will enable the clock, set the direction register to input, turn off the alternative function, and enable the pin. Notice the software is friendly because it just affects PA5 without affecting the other bits in Port A. The input function reads Port A and returns a true (0x20) if the switch is pressed and returns a false (0) if the switch is not pressed. Figure 8.4 shows how we could build this circuit with a protoboard and a LaunchPad.



**Figure 5.2. Interface of a switch to a microcomputer input**



**Figure 5.3.** Construction of the interface of a switch to a microcomputer input. The brown-black-orange resistor is 10k. The switches in the lab-kit should plug into the protoboard. The switch is across the two pins that are closer to each other. It doesn't matter what color the wires are, but in this figure the wires are black, red and green. The two black wires are ground, the red wire is +3.3V, and the green wire is the signal **in**, which connects the switch to PA5 of the microcontroller.

The software in Program 5.1 is called a driver, and it includes an initialization, which is called once, and a second function that can be called to read the current position of the switch. Writing software this way is called an abstraction, because it separates what the switch does (Init, On, Off) from how it works (PortA, bit 5, TM4C123). The first input function uses the bit-specific address to get just PA5, while the second reads the entire port and selects bit 5 using a logical AND.



```

#define PA5 (*(volatile unsigned long *)0x40004080)
void Switch_Init(void){ volatile unsigned long delay;
    SYSCTL_RCGC2_R |= 0x00000001;    // 1) activate clock
    for Port A delay = SYSCTL_RCGC2_R; // allow time for
    clock to start

        // 2) no need to unlock GPIO Port A
    GPIO_PORTA_AMSEL_R &= ~0x20;    // 3) disable analog
    on PA5 GPIO_PORTA_PCTL_R &= ~0x00F00000; // 4)
    PCTL GPIO on PA5

    GPIO_PORTA_DIR_R &= ~0x20;        // 5) direction PA5
    input GPIO_PORTA_AFSEL_R &= ~0x20;
        // 6) PA5 regular port
    function GPIO_PORTA_DEN_R |= 0x20; // 7) enable PA5
    digital port
}

unsigned long Switch_Input(void){

    return PA5; // return 0x20(pressed) or 0(not pressed)

}unsigned long Switch_Input2(void){

    return (GPIO_PORTA_DATA_R&0x20); // 0x20(pressed) or 0(not pressed)

}

```

Program 5.1. Software interface for a switch on PA5 (C8\_Switch).

**Maintenance Tip:** When interacting with just some of the bits of an I/O register it is better to modify just the bits of interest, leaving the other bits unchanged. In this way, the action of one piece of software does not undo the action of another piece.

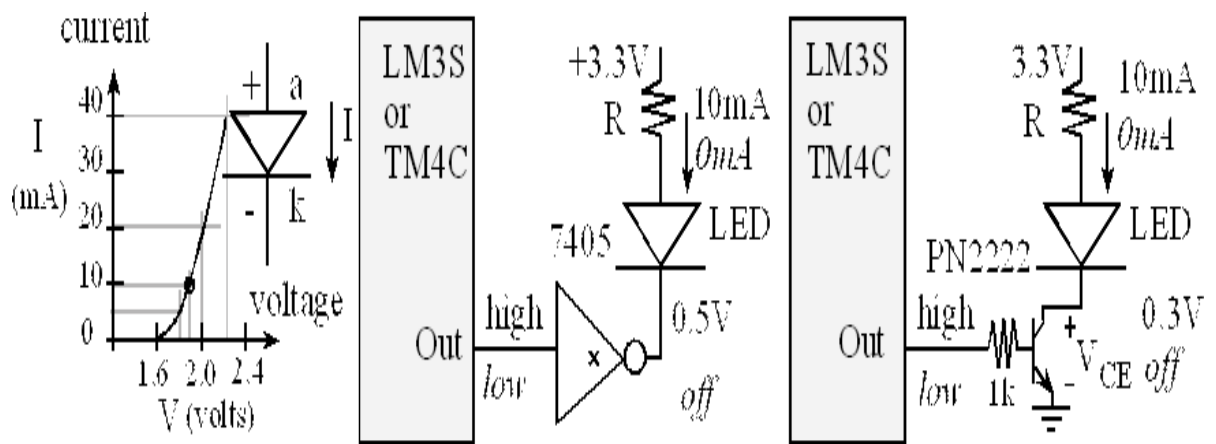
## 2.LED Interfaces

### Driving LEDs

Light emitting diodes (LEDs) are often used as indicators in digital systems and in many cases can simply be directly driven from a logic output provided there is sufficient current and voltage drive .The voltage drive is necessary to get the LED to illuminate in the first place. LEDs will only light up when their diode reversebreakdown voltage is exceeded. This is usually about 2 to 2.2 volts and less than the logic high voltage. The current drive determine show bright the LED will appear and it is usual to have a current limiting resistor in series with the LED to prevent it from drawing too much current and overheating. For a logic device with a 5 vol t supply a 300 resistor will limit the current to about 10 mA. The problem comes if the logic output is only 2.4 or 2.5 volts and not the expected 5 volts. This means that the resistor is sufficient to drop enough voltage so that the LED does not light up.

The solution is to use a buffer so that there is sufficient current drive or alternatively use a transistor to switch on the LED. There are special LED driver circuits packs available that are designed to connect directly to an LED without the need for the current limiting resistor. The resistor or current limiting circuit is included inside the device.

A **light emitting diode** (LED) emits light when an electric current passes through it. LEDs have polarity, meaning current must pass from anode to cathode to activate. The anode is labelled **a** or +, and cathode is labelled **k** or -. The cathode is the short lead and there may be a slight flat spot on the body of round LEDs. Thus, the anode is the longer lead. The brightness of an LED depends on the applied electrical power ( $P=I*V$ ). Since the LED voltage is approximately constant in the active region (see left side of Figure 8.5), we can establish the desired brightness by setting the current.



**Figure 5.4 Positive logic LED interface (Lite-On LTL-10223W).**

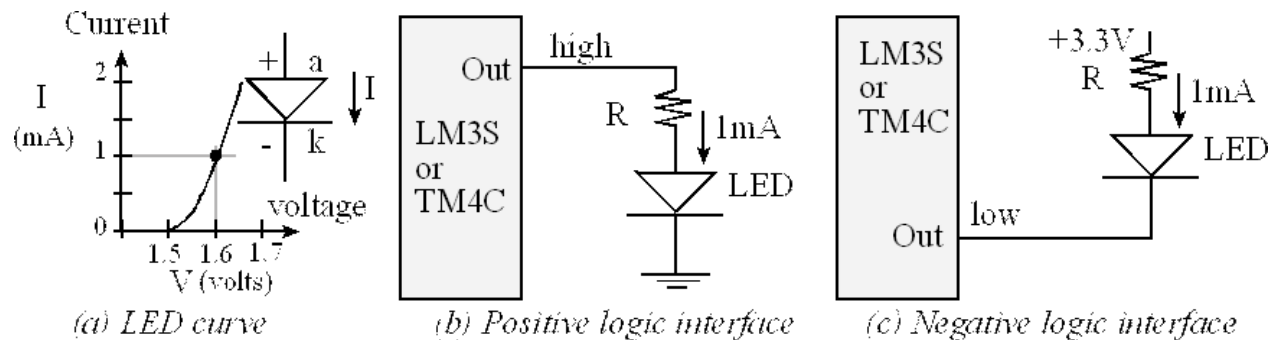
If the LED current is above 8 mA, we cannot connect it directly to the microcontroller because the high currents may damage the chip. Figure 8.5 shows two possible interface circuits we could use. In both circuits if the software makes its output high the LED will be on. If the software makes its output low the LED will be off (shown in Figure 5.5 with *italics*). When the software writes a logic 1 to the output port, the input to the 7405/PN2222 becomes high, output from the 7405/PN2222 becomes low, 10 mA travels through the LED, and the LED is on. When the software writes a logic 0 to the output port, the input to the 7405/PN2222 becomes low, output from the 7405/PN2222 floats (neither high nor low), no current travels through the LED, and the LED is dark. The value of the resistor is selected to

establish the proper LED current. When active, the LED voltage will be about 2 V, and the power delivered to the LED will be controlled by its current. If the desired brightness requires an operating point of 1.9 V at 10 mA, then the resistor value should be

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.9 - 0.5}{0.01} = 90\Omega$$

where  $V_d$ ,  $I_d$  is the desired LED operating point, and  $V_{OL}$  is the output low voltage of the LED driver. If we use a standard resistor value of 100Ω in place of the 90Ω, then the current

will be  $(3.3 - 1.9 - 0.5V)/100\Omega$ , which is about 9 mA. This slightly lower current is usually acceptable.



**Figure 5.5. Low current LED interface (Agilent HLMP-D150).**

When the LED current is less than 8 mA, we can interface it directly to an output pin without using a driver. The LED shown in Figure 5.6a has an operating point of 1.7 V and 1 mA. For the positive logic interface (Figure 5.6b) we calculate the resistor value based on the desired LED voltage and current

$$R = \frac{V_{OH} - V_d}{I_d} = \frac{2.4 - 1.6}{0.001} = 800 \Omega$$

where  $V_{OH}$  is the output high voltage of the microcontroller output pin. Since  $V_{OH}$  can vary from 2.4 to

3.3 V, it makes sense to choose a resistor from a measured value of  $V_{OH}$ , rather than the minimum value of 2.4 V. Negative logic means the LED is activated when the software outputs a zero. For the negative logic interface (Figure 5.6c) we use a similar equation to determine the resistor value

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.6 - 0.4}{0.001} = 1.3 \text{ k}\Omega$$

where  $V_{OL}$  is the output low voltage of the microcontroller output pin.

If we use a 1.2 k $\Omega$  in place of the 1.3 k $\Omega$ , then the current will be  $(3.3 - 1.6 - 0.4V)/1.2k\Omega$ , which is about

1.08 mA. This slightly higher current is usually acceptable. If we use a standard resistor value of 1.5 k $\Omega$  in place of the 1.3 k $\Omega$ , then the current will be  $(3.3 - 1.6 - 0.4V)/1.5k\Omega$ , which is about 0.87 mA. This slightly lower current is usually acceptable.

The software in Program 5.2 is called a driver, and it includes an initialization, which is called once, and two functions that can be called to turn on and off the LED. Writing software this way is called an abstraction, because it separates what the LED does (Init, On, Off) from how it works (PortA, TM4C123).

Checkpoint 8.2: What resistor value in of Figure 8.6 is needed if the desired LED operating point is 1.7V and 2 mA? Use the negative logic interface and,  $V_{OL}$  of 0.4V.

```

void LED_Init(void){ volatile unsigned long delay;
    SYSCTL_RCGC2_R |= 0x01;    // 1) activate clock for Port A
    delay = SYSCTL_RCGC2_R;    // allow time for clock to
    start

    // 2) no need to unlock PA2
    GPIO_PORTA_PCTL_R &= ~0x00000F00; // 3) regular
    GPIO

    GPIO_PORTA_AMSEL_R &= ~0x04;    // 4) disable analog function
    on PA2 GPIO_PORTA_DIR_R |= 0x04;    // 5) set direction to output
    GPIO_PORTA_AFSEL_R &= ~0x04;    // 6) regular port function
    GPIO_PORTA_DEN_R |= 0x04;    // 7) enable digital port

}

// Make PA2 high
void
LED_On(void){

    GPIO_PORTA_DATA_R |= 0x04;

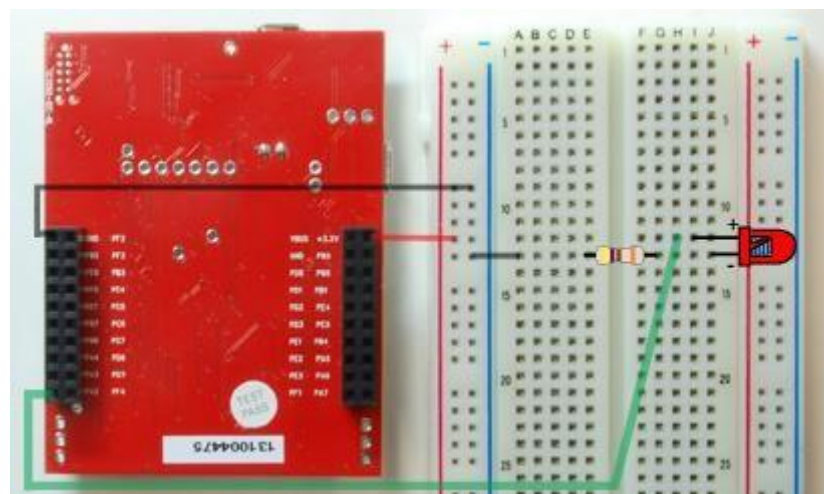
}

// Make PA2 low
void
LED_Off(void){

    GPIO_PORTA_DATA_R &= ~0x04;

}

```



**Figure 5.6. Construction of the interface of an LED to a microcomputer output**

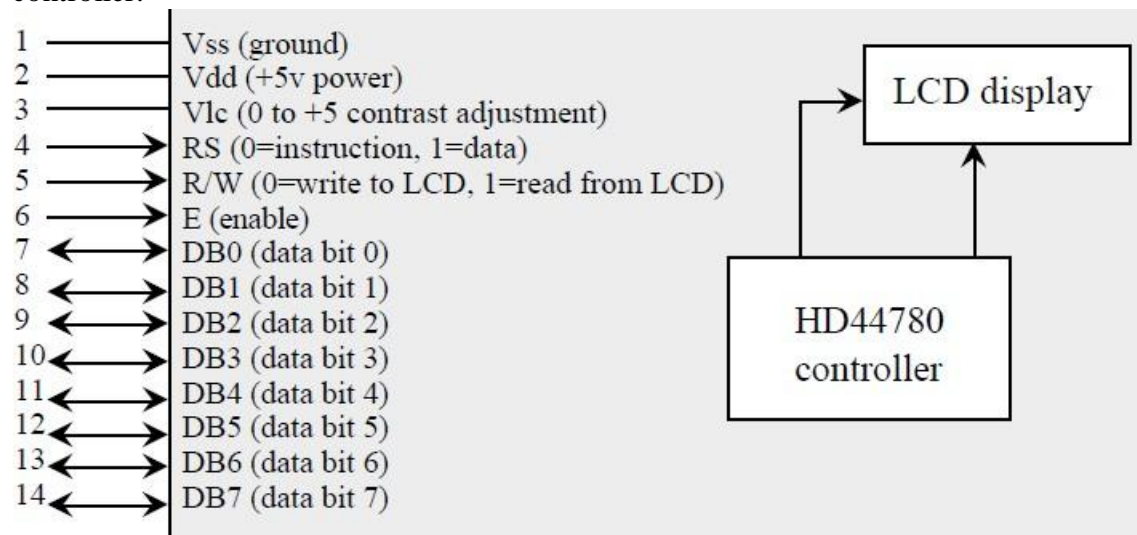
The yellow-purple-brown resistor is 470ohm. It doesn't matter what color the wires are, but in this figure the wires are black, red and green. The two black wires are ground, the red wire is +3.3V, and the green wire is the signal **Out**, which connects PA2 of the microcontroller to the positive side of the LED.

### 3.LCD Display Interface

Liquid Crystal Display (LCD) consists of rod-shaped tiny molecules sandwiched between a flat piece of glass and an opaque substrate. These rod-shaped molecules in between the plates align into two different physical positions based on the electric charge applied to them. When electric charge is applied they align to block the light entering through them, where as when no-charge is applied they become transparent.

Light passing through makes the desired images appear. This is the basic concept behind LCD displays. LCDs are most commonly used because of their advantages over other display technologies. They are thin and flat and consume very small amount of power compared to LED displays and cathode ray tubes (CRTs).

background Microprocessor controlled LCD displays are widely used, having replaced most of their LED counterparts, because of their low power and flexible display graphics. This experiment will illustrate how a handshaked parallel port of the microcomputer will be used to output to the LCD display. The hardware for the display uses an industry standard HD44780 controller. The low-level software initializes and outputs to the HD44780 controller.



**Figure 5.7 - 16 LCD display.**

There are four types of access cycles to the HD44780 depending on RS and R/W RS R/W Cycle  
 0 0 Write to Instruction Register

0 1 Read Busy Flag (bit 7)  
 1 0 Write data from  $\mu$ P to the  
 HD44780 1 1 Read data from  
 HD44780 to the  $\mu$ P

Two types of synchronization can be used, blind cycle and gadfly. Most operations require 40  $\mu$ s to complete while some require 1.64 ms. The example implementation shown in the LCD12.H, LCD12.C uses OC5 to create the blind cycle wait. A gadfly interface provides feedback to detect a faulty interface, but has the problem of creating a software crash if the LCD never finishes. The best interface utilized both gadfly and blind cycle, so that the software can return with an error code if a display operation does not finish on time (due to a broken wire or damaged display.)

In embedded systems like we use, it is OK to provide LCD12.H and LCD12.C files which the user can compile with their application. In our embedded system, linking will be performed by the compiler. You are encouraged to modify/extend this example, and define/develop/test your own format. Normally, we group the device driver software into four categories. We will use interrupts in the later labs.

## 4.Data structures: global, protected

Open Flag Boolean that is true if the display port is open initially false, set to true by LCD Open, set to false by LCD Close static storage (or dynamically created at bootstrap time, i.e., when loaded into memory)

### 1. Initialization routines (called by user)

```
#define LCDscroll 8
#define LCDnoscroll 0
#define LCDleft 0
#define LCDright 4
```

LCDOpen Initialization of display  
 port Sets OpenFlag to true  
 Initialize hardware, other data structures

Returns an error code if unsuccessful  
 hardware non-existent, already open, out of memory, hardware failure, illegal  
 parameter Input Parameters(mode) see the LCD data sheets for various options,  
 e.g., scrolling Output Parameter(none)  
 Typical calling sequence

```
if(!LCDOpen(LCDscroll|LCDright))
error(); LCDClose Release of display
port
Sets OpenFlag to false
Release any dynamically allocated
memory Returns an error code if not
previously open Output Parameter(error
code)
Typical calling
sequence
```

```
if(!LCDClose())  
error();
```

2. Regular I/O calls (called by user to perform I/O)  
LCDPutChar Output an ASCII character to the  
LCD port Returns an error code if unsuccessful  
device not open, hardware failure (happens when a wire is  
loose) Input Parameter(ASCII character)  
Output Parameter(error code)  
Typical calling sequence (you are free to  
change) if(LCDPutChar(letter)) error();

3. Support software (protected, not directly accessible by the user).  
None in this category for this lab, but there will be in later labs.

### Preparation

Show the required hardware connections. Label all hardware chips, pin numbers, and resistor values.  
Write the  
low-level LCD device driver. You must have a separate LCD12.H and LCD12.C files to  
simplify the reuse of these routines. Write a main program that tests all features of the  
interface.

### Procedure

You should look at the +5 V voltage versus time signal on a scope when power is first turned  
on to determine  
if the LCD “power on reset” circuit will be properly activated. The LCD data sheet specifies  
it needs from 0.1 ms to 10 ms rise time from 0.2 V to 4.5 V to generate the power on reset.  
Connect the LCD to your microcomputer. Use the scope to verify the sharpness of the digital  
inputs/outputs. Adjust the contrast potentiometer for the best looking display. Test the device  
driver software and main program in small pieces.

### Checkout

You should be able to demonstrate all the “cool” features of your LCD display system.

### Hints

- 1) Make sure the 14 wires are securely attached to your board.
- 2) One way to test for the first call to open is to test the direction register. After reset, the  
direction registers are usually zero, after a call to open, some direction register bits will be  
one.
- 3) Download from the class web site the files LCDTEST.C LCD12.H and LCD12.C files.  
These C language routines to do low level LCD output to Port H/J. Notice that it does not  
perform any input (either status or data), therefore it leaves DDRJ=0xFF, DDRH=0xFF. If  
you wish to include inputs, then you will have to toggle DDRH, so that PORTH is an output  
for writes and an input for reads.
- 4) Although many LCD displays use the same HD44780 controller, the displays come in  
various sizes ranging from 1 row by 16 columns up to 4 rows by 40 columns.



## Analog to Digital Conversion, Data Acquisition and Control

we have seen that an embedded system uses its input/output devices to interact with the external world. In this chapter we will focus on input devices that we use to gather information about the world. More specifically, we present a technique for the system to measure analog inputs using an analog to digital converter (ADC). We will use periodic interrupts to sample the ADC at a fixed rate. We will then

combine sensors, the ADC, software, PWM output and motor interfaces to implement intelligent control on our robot car.

### 5. Analog to Digital Conversion

An analog to digital converter (ADC) converts an analog signal into digital form, shown in Figure 5.9 An embedded system uses the ADC to collect information about the external world (data acquisition system.) The input signal is usually an analog voltage, and the output is a binary number. The ADC precision is the number of distinguishable ADC inputs (e.g., 4096 alternatives, 12 bits). The ADC **range** is the maximum and minimum ADC input (e.g., 0 to +3.3V). The ADC **resolution** is the smallest distinguishable change in input (e.g., 3.3V/4096, which is about 0.81 mV). The resolution is the change in input that causes the digital output to change by 1.

$$\text{Range(volts)} = \text{Precision(alternatives)} \cdot \text{Resolution(volts)}$$

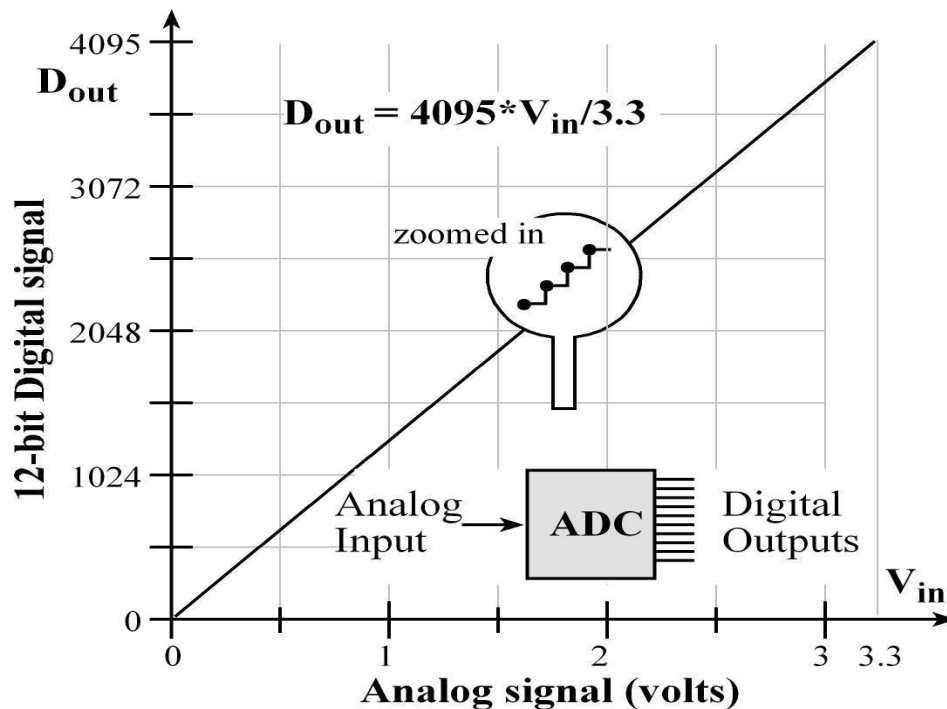
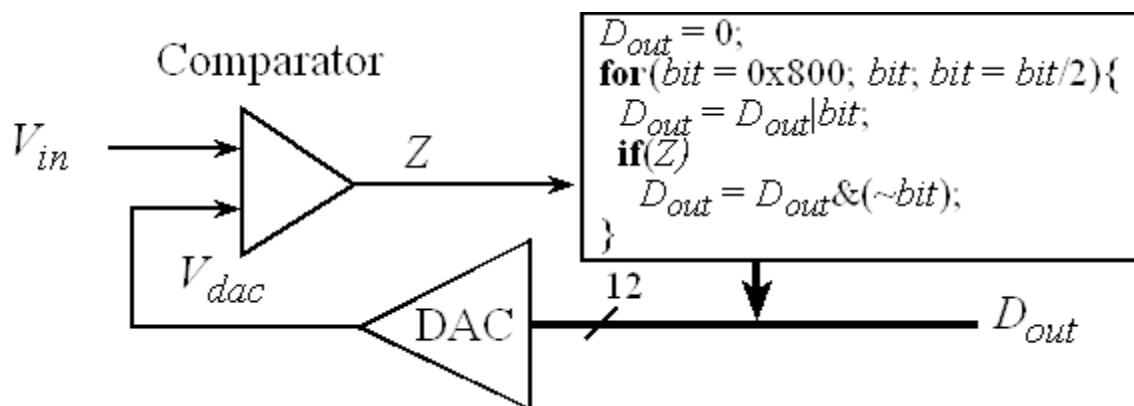


Figure 5.8 A 12-bit ADC converts



The most pervasive method for ADC conversion is the **successive approximation** technique, as illustrated in Figure 5.10. A 12-bit successive approximation ADC is clocked 12 times. At each clock another bit is determined, starting with the most significant bit. For each clock, the successive approximation hardware issues a new "guess" on  $V_{dac}$  by setting the bit under test to a "1". If  $V_{dac}$  is now higher than the unknown input,  $V_{in}$ , then the bit under test is cleared. If  $V_{dac}$  is less than  $V_{in}$ , then the bit under test remains 1. In this description, bit is an unsigned integer that specifies the bit under test. For a 12-bit ADC, bit goes 2048, 1024, 512, 256, ..., 1.  $D_{out}$  is the ADC digital output, and  $Z$  is the binary input that is true if  $V_{dac}$  is greater than  $V_{in}$ . 0 to 3.3V on its input into a digital number from 0 to 4095



**Figure 5.9 A 12-bit successive approximation ADC.**

Normally we don't specify accuracy for just the ADC, but rather we give the accuracy of the entire system (including transducer, analog circuit, ADC and software). An ADC is **monotonic** if it has no missing codes as the analog input slowly rises. This means if the analog signal is a slowly rising voltage, then the digital output will hit all values one at a time, always going up, never going down. The **figure of merit** of an ADC involves three factors: precision (number of bits), speed (how fast can we sample), and power (how much energy does it take to operate). How fast we can sample involves both the ADC conversion time (how long it takes to convert), and the bandwidth (what frequency components can be recognized by the ADC). The ADC cost is a function of the number and quality of internal components. Two 12-bit ADCs are built into the TM4C123/LM4F120 microcontroller. You will use ADC0 to collect data and we will use ADC1 and the PD3 pin to implement a voltmeter and oscilloscope.

## 6.ADC on the TM4C123/LM4F120

Table 1 shows the ADC0 register bits required to perform sampling on a single channel. There are two ADCs; you will use ADC0 and the grader uses ADC1. For more complex configurations refer to the specific data sheet. Bits 8 and 9 of the **SYSTCTL\_RCGC0\_R** specify the maximum sampling rate, see Table 14.2. The TM4C123 can sample up to 1 million samples per second. Bits 8 and 9 of the **SYSTCTL\_RCGC0\_R** specify how fast it COULD sample; the actual sampling rate is determined by the rate at which we trigger the ADC. In this chapter we will use software trigger mode, so the actual sampling rate is determined by the SysTick periodic interrupt rate; the SysTick ISR will take one ADC sample. On the TM4C123, we will need to set bits in the **AMSEL** register to activate the analog interface.

**Table5. 1. The TM4C ADC0 registers. Each register is 32 bits wide.**

Address	31-17	16	15-10	9	8		7-0		Name
0x400F.E100		ADC		MAXA	DCSPD				SYSCTL_RCGC0_R
	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
0x4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R
	31-16			15-12		11-8	7-4	3-0	
0x4003.8014				EM3		EM2	EM1	EM0	ADC0_EMUX_R
	31-4			3	2	1	0		
0x4003.8000				ASEN3	ASEN2	ASEN1	ASEN0		ADC0_ACTSS_R
0x4003.80A0				MUX0					ADC0_SSMUX3_R
0x4003.80A4				TS0	IE0	END0	D0		ADC0_SSCTL3_R
0x4003.8028				SS3	SS2	SS1	SS0		ADC0_PSSI_R
0x4003.8004				INR3	INR2	INR1	INR0		ADC0_RIS_R
0x4003.800C				IN3	IN2	IN1	IN0		ADC0_ISC_R
	31-12			11-0					
0x4003.80A8				DATA					ADC0_SSFIFO3

**Table 5.2 . The ADC MAXADCSPD bits in the SYSCTL\_RCGC0\_R register.**

<i>Value</i>	<i>Description</i>
0x3	1M samples/second
0x2	500K samples/second
0x1	250K samples/second
0x0	125K samples/second

Table 5.3 shows which I/O pins on the TM4C123 can be used for ADC analog input channels.

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PB4	Ain10	Port		SSI2Clk		M0PWM2			T1CCP0	CAN0Rx		
PB5	Ain11	Port		SSI2Fss		M0PWM3			T1CCP1	CAN0Tx		
PD0	Ain7	Port	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0		WT2CCP0			
PD1	Ain6	Port	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1		WT2CCP1			
PD2	Ain5	Port	SSI3Rx	SSI1Rx		M0Fault0			WT3CCP0	USB0ep		
PD3	Ain4	Port	SSI3Tx	SSI1Tx				IDX0	WT3CCP1	USB0pflt		
PE0	Ain3	Port	U7Rx									
PE1	Ain2	Port	U7Tx									
PE2	Ain1	Port										
PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I2C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I2C2SDA	M0PWM5	M1PWM3			CAN0Tx		

The ADC has four sequencers, but you will use only sequencer 3 in Labs 14 and 15. We set the ADC0\_SS PRI\_R register to 0x0123 to make sequencer 3 the highest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (EM3) in the ADC0\_EMUX\_R register to specify how the ADC will be triggered.

Table 4 shows the various ways to trigger an ADC conversion. More advanced ADC triggering techniques are presented in the book *Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers*. However in this course, we use software start (EM3=0x0). The software writes an 8 (SS3) to the ADC0\_PSSI\_R to initiate a conversion on sequencer 3. We can enable and disable the sequencers using the ADC0\_ACTSS\_R register. There are twelve ADC channels on the LM4F120/TM4C123. Which channel we sample is configured by writing to the ADC0\_SSMUX3\_R register. The mapping between channel number and the port pin is shown in Table 3. For example channel 9 is connected to the pin PE4. The ADC0\_SSCTL3\_R register specifies the mode of the ADC sample. We set TS0 to measure temperature and clear it to measure the analog voltage on the ADC input pin. We set IE0 so that the INR3 bit is set when the ADC conversion is complete, and clear it when no flags are needed. When using sequencer 3, there is only one sample, so END0 will always be set, signifying this sample is the end of the sequence. In this class, the

*sequence* will be just one ADC conversion. We set the D0 bit to activate differential sampling, such as measuring the analog difference between two ADC pins. In our example, we clear D0 to sample a single-ended analog input. Because we set the IE0 bit, the INR3 flag in the ADC0\_RIS\_R register will be set when the ADC conversion is complete. We clear the INR3 bit by writing an 8 to the ADC0\_ISC\_R register.

Table 5.4. The ADC EM3, EM2, EM1, and EM0 bits in the ADC\_EMUX\_R register

<i>Value</i>	<i>Event</i>
0x0	Software start
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	Analog Comparator 2
0x4	External (GPIO PB4)
0x5	Timer

0x6	PWM0
0x7	PWM1
0x8	PWM2
0x9	PWM3
0xF	Always (continuously sample)

We perform the following steps to configure the ADC for software start on one channel. Program 14.1 shows a specific details for sampling PE4, which is channel 9. The function **ADC0\_InSeq3** will sample PE4 using software start and use busy-wait synchronization to wait for completion.

**Step 1.** We enable the port clock for the pin that we will be using for the ADC input.

**Step 2.** Make that pin an input by writing zero to the **DIR** register.

**Step 3.** Enable the alternative function on that pin by writing one to the **AFSEL** register.

**Step 4.** Disable the digital function on that pin by writing zero to the **DEN** register.

**Step 5.** Enable the analog function on that pin by writing one to the **AMSEL** register.

**Step 6.** We enable the ADC clock by setting bit 16 of the **SYSCTL\_RCGC0\_R** register.

**Step 7.** Bits 8 and 9 of the **SYSCTL\_RCGC0\_R** register specify the maximum sampling rate of the ADC. In this example, we will sample slower than 125 kHz, so the maximum sampling rate is set at 125 kHz. This will require less power and produce a longer sampling time, creating a more accurate conversion.

**Step 8.** We will set the priority of each of the four sequencers. In this case, we are using just one sequencer, so the priorities are irrelevant, except for the fact that no two sequencers should have the same priority.

**Step 9.** Before configuring the sequencer, we need to disable it. To disable sequencer 3, we write a 0 to bit 3 (**ASEN3**) in the **ADC\_ACTSS\_R** register. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.

**Step 10.** We configure the trigger event for the sample sequencer in the **ADC\_EMUX\_R** register. For this example, we write a 0000 to bits 15–12 (**EM3**) specifying software start mode for sequencer 3.

**Step 11.** Configure the corresponding input source in the **ADCSSMUXn** register. In this example, we write the channel number to bits 3–0 in the **ADC\_SSMUX3\_R** register. In this example, we sample channel 9, which is PE4.

**Step 12.** Configure the sample control bits in the corresponding nibble in the **ADC0SSCTLn** register. When programming the last nibble, ensure that the **END** bit is set. Failure to set the **END** bit causes unpredictable behavior. Sequencer 3 has only one sample, so we write a 0110 to the **ADC\_SSCTL3\_R** register. Bit 3 is the **TS0** bit, which we clear because we are not measuring temperature. Bit 2 is the **IE0** bit, which we set because we want the **RIS** bit to be set when the sample is complete. Bit 1 is the **END0** bit, which is set because this is the last (and only) sample in the sequence. Bit 0 is the **D0** bit, which we clear because we do not wish to use differential mode.

**Step 13.** We enable the sample sequencer logic by writing a 1 to the corresponding **ASENn**. To enable sequencer 3, we write a 1 to bit 3 (**ASEN3**) in the **ADC\_ACTSS\_R** register.

```

void ADC0_InitSWTriggerSeq3_Ch9(void){ volatile unsigned long delay;
SYSCTL_RCGC2_R |= 0x00000010; // 1) activate clock for Port
E delay = SYSCTL_RCGC2_R; // allow time for clock to
stabilize GPIO_PORTE_DIR_R &= ~0x04; // 2) make
PE4 input

GPIO_PORTE_AFSEL_R |= 0x04; // 3) enable alternate function on
PE2 GPIO_PORTE_DEN_R &= ~0x04; // 4) disable digital I/O on
PE2 GPIO_PORTE_AMSEL_R |= 0x04; // 5) enable analog
function on PE2 SYSCTL_RCGC0_R |= 0x00010000; // 6) activate
ADC0

delay = SYSCTL_RCGC2_R

SYSCTL_RCGC0_R &= ~0x00000300; // 7) configure for 125K
ADC0_SSRI_R = 0x0123; // 8) Sequencer 3 is highest
priority ADC0_ACTSS_R &= ~0x0008;
// 9) disable sample sequencer 3
ADC0_EMUX_R &= ~0xF000; // 10) seq3 is software trigger
ADC0_SSMUX3_R &= ~0x000F;
// 11) clear SS3 field
ADC0_SSMUX3_R += 9; // set channel Ain9 (PE4)
ADC0_SSCTL3_R = 0x0006; // 12) no TS0 D0, yes IE0 END0

ADC0_ACTSS_R |= 0x0008; // 13) enable sample sequencer 3
}

```

Program 5.3 Initialization of the ADC using software start and busy-wait (C14\_ADCSWTrigger).

Program 5.4 gives a function that performs an ADC conversion. There are four steps required to perform a software-start conversion. The range is 0 to 3.3V. If the analog input is 0, the digital output will be 0, and if the analog input is 3.3V, the digital output will be 4095.

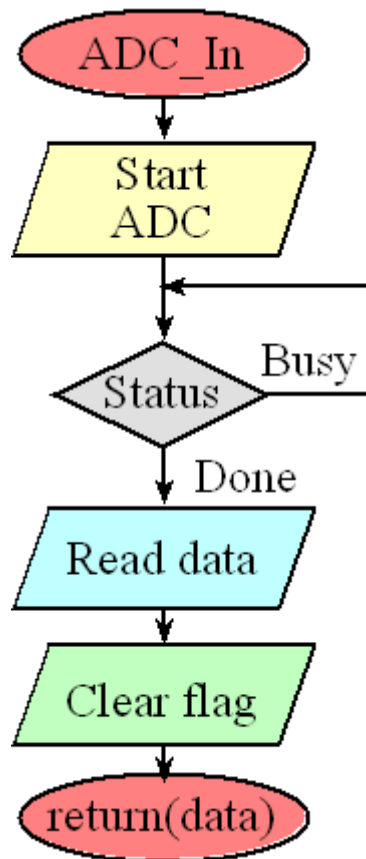
Digital Sample = (Analog Input (volts) • 4095) / 3.3V(volts)

**Step 1.** The ADC is started using the software trigger. The channel to sample was specified earlier in the initialization.

**Step 2.** The function waits for the ADC to complete by polling the RIS register bit 3.

**Step 3.** The 12-bit digital sample is read out of sequencer 3.

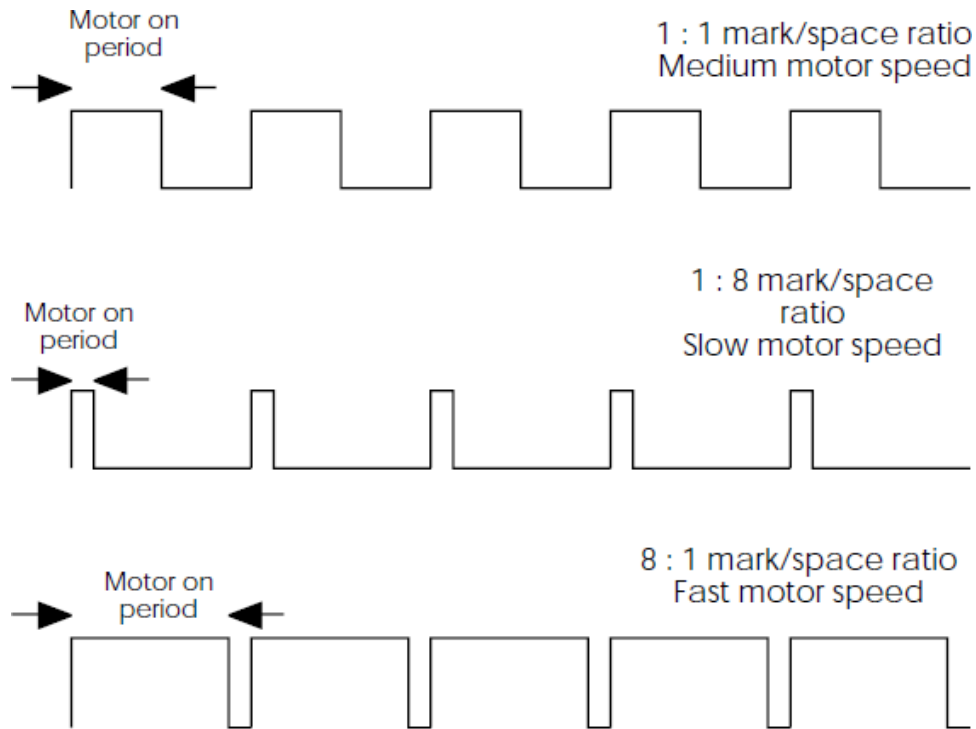
**Step 4.** The RIS bit is cleared by writing to the ISC register.



**Figure 5.10.** The four steps of analog to digital conversion: 1) initiate conversion, 2) wait for the ADC to finish, 3) read the digital result, and 4) clear the completion flag.

## 7 .Interfacing to DC motors

So far with controlling DC motors, the emphasis has been simple on-off type switching. It is possible with a digital system to actually provide speed control using a technique called pulse width modulation



**Figure 5.11. Using different PWM waveforms to control a DC motor speed**

With a DC motor, there are two techniques for controlling the motor speed: the first is to reduce the DC voltage to the motor. The higher the voltage, the faster it will turn. At low voltages, the control can be a bit hit and miss and the power control is inefficient. The alternative technique called pulse width modulation(PWM) will control a motor speed not by reducing the voltage to the motor but by reducing the time that the motor is switched on.

This is done by generating a square wave at a frequency of several hundred hertz and changing the mark/space ratio of the wave form. With a large mark and a low space, the voltage is applied to the motor for almost all of the cycle time, and thus the motor will rotate very quickly. With a small mark and a large space, the opposite is true. The diagram shows the waveforms for medium, slow and fast motor control. The only difference between this method of control and that for a simple on-off switch is the timing of the pulses from the digital output to switch the motor on and off. There are several methods that can be used to generate these waveforms.

### Software only

With a software-only system, the waveform timing is done by creating some loops that provide the timing functions. The program pseudo code shows a simple structure for this. The first action is to switch the motor on and then to start counting through a delay loop. The length of time to count through the delay loop determines the motor-on period. When the

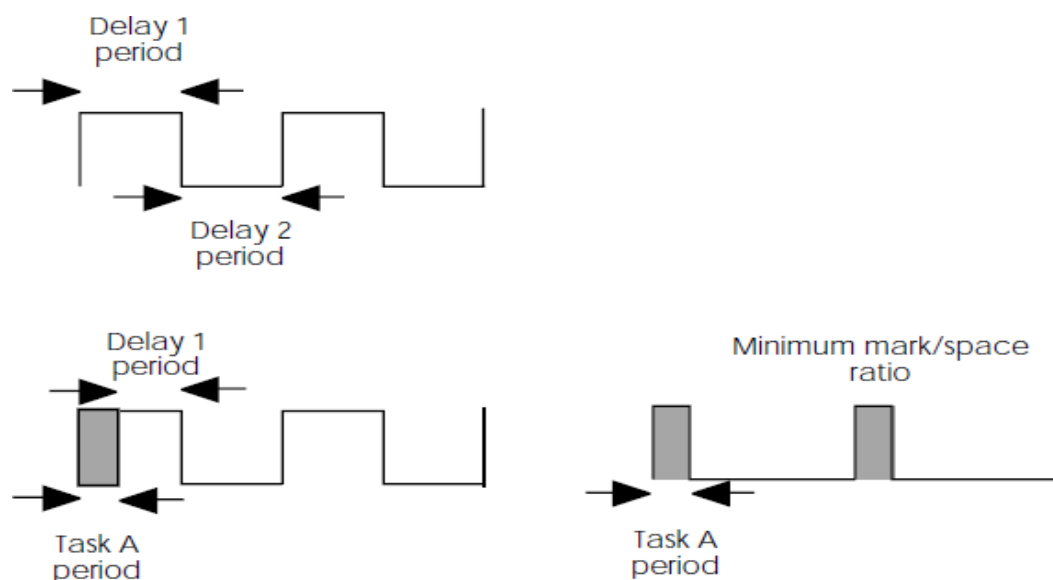


count is finished, the motor is switched off. The next stage is to count through a second delay loop to determine the motor-off period.

```
repeat (forever)
{
}
```

This whole procedure is repeated for as long as the motor needs to be driven. By changing the value of the two delays, the mark/space ratio of the waveform can be altered. The total time taken to execute the repeat loop gives the frequency of the waveform. This method is processor intensive in that the program has to run while the motor is running. On first evaluation, it may seem that while the motor is running, nothing else can be done. This is not the case. Instead of simply using delay loops, other work can be inserted in here whose duration now becomes part of the timing for the PWM waveform. If the work is short, then the fine control over the mark/space ratio is not lost because the contribution that the work delay makes compared to the delay loop is small. If the work is long, then the minimum motor-on time and thus motor speed is determined by this period.

```
repeat (forever)
{
  switch      on
  motor perform
  task a delay
  loop1 switch
  off        motor
  delay loop2
}
```



**Figure 5.12 The timing diagrams for the software PWM implementation**

The timing diagrams for the software loop PWM waveforms are shown in the diagrams above. In general, software only timing loops are not efficient methods of generating PWM waveforms for motor control. The addition of a single timer greatly improves the mechanism.

### ***Using a single timer***

By using a single timer, PWM waveforms can be created far easier and free up the processor to do other things without impacting the timing. There are several methods that can be used to do this. The key principle is that the timer can be programmed to create a periodic interrupt.

#### **Method 1 — using the timer to define the on period**

With this method, the timer is used to generate the onperiod. The processor switches the motor on and then starts thetimer to count down. While the timer is doing this, the processor is free to do what ever work is needed. The timer will eventually time out and generate a processor interrupt. The processor services the interrupt and switches the motor off. It then goes into a delay loop still within the service routine until the time period arrives to switch the motor on again. The processor switches the motor on, resets the timer and starts it counting and continues with its work by returning from the interrupt service routine.

#### ***Method — using the timer to define frequency period***

With this method, the timer is used to generate a periodicinterrupt whose frequency is set by the timer period. When theprocessor services the interrupt, it uses a software loop to determinethe on period. The processor switches on the motor and uses the software delay to calculate the on period. When the delay loop is completed, it switches off the motor and can continue with otherwork until the timer generates the next interrupt.

#### ***Method 3 — using the timer to define both the on and off periods***

With this method, the timer is used to generate both the on and off periods. The processor switches the motor on, loads the timer with the on-period value and then starts the timer to count down. While the timer is doing this, the processor is free to do what ever work is needed. The timer will eventually time out and generate a processor interrupt, as before. The processor services the interrupt and switches the motor off. It then loads the timer with the value for the off period. The processor then starts the timer counting and continues with its work by returning from the interrupt service routine. The timer now times out and generates an interrupt. The processor services this by switching the motor on, loading the timer with the one delay value and setting the timer counting before returning from the interrupt.

As a result, the processor is only involved when interruptedby the timer to switch the motor on or off and load the timer withthe appropriate delay value and start it counting. Of all these threemethods, this last method is the most processor efficient. Withmethods 1 and 2, the processor is only free to do other work when the mark/space ratio is such that there is time to do it. With a longmotor-off period, the processor performs the timing in softwareand there is little time to do anything else. With a short motor-offperiod, there is more processing time and far more work can be done. The problem is that the work load that can be achieved is dependent on the mark/space ratio of the PWM waveform andengine speed. This can be

a major restriction and this is why the third method is most commonly used.

### *Using multiple timers*

With two timers, it is possible to generate PWM waveforms with virtually no software intervention. One timer is set up to generate a periodic output at the frequency of the required PWM waveform. This output is used to trigger a second timer which is configured as a monostable. The second timer output is used to provide the motor-on period. If these timers are set to automatically reload, the first timer will continually trigger the second and thus generate a PWM waveform. By changing the delay value in the second timer, the PWM mark/space ratio can be altered as needed.

## **8.DAC and ADC**

Suppose a system needs to give an analog output of a control circuit for automation. The analog output may be to a power system for a d.c. motor or furnace. A Pulse Width Modulator (PWM) unit in the microcontroller operates as follows: Pulse width is made proportional to the analog-output needed. PWM inputs are from 00000000 to 11111111 for an 8-bit DAC operation. The PWM unit outputs to an external integrator and then provides

the desired analog output. Suppose an integrator circuit (external to the microcontroller) gives an output of 1.024 Volt when the pulse width is 50% of the total pulse time period, and 2.047V when the width is 100%. When the width is made 25% by reducing by half the value in PWM output control-register, the integrator output will become 0.512 Volt.

Now assume that the integrator operates with a dual (plus-minus) supply. Also assume that when an integrator circuit gives an output of 1.023 Volt, the pulse width is 100% of total pulse time period and -1.024 Volt when the width is 0%. When the width is made 25% by reducing by half the value in an output control register, the integrator output will be 0.512 Volt; at 50% the output will be 0.0 Volt. From this information, finding the formulae to obtain converted bits for a given PWM register bits ranging from 00000000 to 11111111 in both the situations is left as an exercise for the reader. The ADC in the system microcontroller can be used in many applications such as Data Acquisition System (DAS), analog control system and voice digitizing system. Suppose a system needs to read an analog input from a sensor or transducer circuit. If converted to bits by the ADC unit in the system, then these bits, after processing, can also give an output. This provides a control for automation by a combined use of ADC and DAC features. The converted bits can be given to the port meant for digital display. The bits may be transferred to a memory address, a serial port or a parallel port. A processor may process the converted bits and generate a Pulse Code Modulated (PCM) output. PCM signals are used digitizing the voice in the digital format].

Important points about the ADC are as follows:

1. Either a single or dual analog reference voltage source is required in the ADC. It sets either the analog input's upper limit only or the lower and upper limits both. For a single reference source, the lower limit is set to 0V (ground potential). When the analog input equals the lower limit the ADC generates all bits as 0s, and when it equals the upper limit it generates all bits as 1s. [As an example, suppose in an ADC the upper limit or reference voltage is set as 2.255 Volt. Let the lower limit reference Voltage be 0.255V. Difference in the limits is 2 Volt. Therefore, the resolution will be  $(2/256)$  Volt. If the 8-bit ADC analog-input is 0.255V, the converted 8 bits will be 00000000. When the input is  $(0.255V + 1.000V) = 1.255V$ , the bits will be 10000000. When the analog input is  $(0.255V + 0.50V)$ , the converted bits will be 01000000. [From this information, finding a formula to obtain converted bits for a given analog input = v Volt is left as an exercise for the reader].
2. An ADC may be of eight, ten, twelve or sixteen bits depending upon the resolution needed for conversion.
3. The start of the conversion signal (STC) signal or input initiates the conversion to 8 bits. In a system, an instruction or a timer signals the STC
4. There is an end of conversion (EOC) signal. In a system, a flag in a register is set to indicate the end of conversion and generate an interrupt.
5. There is a conversion time limit in which the conversion is definite.
6. A Sample and Hold (S/H) unit is used to sample the input for a fixed time and hold till conversion is over. An ADC unit in the embedded system microcontroller may have multi-channels. It can then take the inputs in succession from the various pins interconnected to different analog sources

## 9.LCD and LED displays

A system requires an interfacing circuit and software to display the status or message for a line, for multi-line displays, or flashing displays. An LCD screen may show up a multi-line display of characters or also show a small graph or icon (called pictogram). A recent innovation in the mobile phone system turns the screen blue to indicate an incoming call. Third generation system phones have both image and graphic displays. An LCD needs little power. It is powered by a supply or battery (a solar panel in the calculator). LCD is a diode that absorbs or emits light on application of 3 V to 4 V and 50 or 60 Hz voltage-pulses with currents less than ~50 mA. The pulses are applied with the same polarity on crystal front and back plane for no light, or with opposite polarity for light. Here polarity at annstance means logic '1' or '0']. An LSI (Lower Scale Integrated Circuit) display-controller is often used in the case of matrix displays. For indicating ON status of the system there may be an LED, which glows when it is ON. A flashing LED may indicate that a specific task is under completion or is running. It may indicate a wait status for a message. The LED is a diode that emits yellow, green, red (or infrared light in a remote controller), on application of a forward voltage of 1.6 to 2 V. An LED needs current up to 12 mA above 5 mA (less in

flashing display mode) and is much brighter than the LCD. Therefore, for flashing display and for display limited to few digits, LEDs are used in a system

## **10.keypad or keyboard**

The keypad or keyboard is an important device for getting user inputs. The system must provide the necessary interfacing and key-debouncing circuit as well as the software for the system to receive input from a set of keys or from a keyboard or keypad. A keypad has up to a maximum of 32 keys. A keyboard may have 104 or more keys. The keypad or keyboard may interface serially or as parallel to processor directly through a parallel or serial port or through a controller.

### **Question Bank**

1. Explain the LED interfacing with ARM Processor
2. Explain with neat diagram the analog to digital data conversion
3. Explain the Pulse width modulation technique for motor speed control
4. Discuss about the motor interfacing techniques.
5. Describe the LCD interfacing Techniques

### **TEXT / REFERENCE BOOKS**

1. Andrew N.Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guide: Designing & Optimizing System Software, Elsevier, 2004.
2. Jonathan W.Valvano, Embedded Microcomputer Systems: Real Time Interfacing, Cengage Learning, 2011
3. Wayne Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2008.
4. C.M.Krishna, Kang G.Shin, Real time systems, McGraw Hill, 3rd reprint, 2010.
5. Herma K., Real Time Systems: Design for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
6. William Hohl, ARM Assembly Language, Fundamentals and Techniques, Taylor & Francis, 2009.