SCSA1307	EMBEDDED SYSTEM	L	T	P	Credits	Total Marks
		3	0	0	3	100

COURSE OBEJCTIVES

- To understand the technologies behind the embedded computing systems
- To acquire knowledge about microcontrollers embedded processors and their applications
- To analyze and develop software programs for embedded systems
- To have knowledge about the working of a microcontroller system and its programming in assembly language
- To provide experience to integrate hardware and software for microcontroller application systems

UNIT 1 INTRODUCTION AND REVIEW OF EMBEDDED HARDWARE 9 Hrs.

Terminology Gates Timing diagram Memory Microprocessor buses Direct memory access Interrupts Built interrupts Interrupts basis Shared data problems Interrupt latency - Embedded system evolution trends Round-Robin Round Robin with interrupt function Rescheduling architecture algorithm.

UNIT 2 REAL TIME OPERATING SYSTEM 9 Hrs.

Task and Task states Task and data Semaphore and shared data operating system services Message queues timing functions Events Memory management Interrupt routines in an RTOS environment Basic design using RTOS.

UNI 3 EMBEDDED HARDWARE, SOFTWARE AND PERIPHERAL 9 Hrs.

Custom single purpose processors: Hardware Combination Sequence Processor design RT level design optimizing software: Basic Architecture Operation Programmers view Development Environment ASIP Processor Design Peripherals Timers, counters and watch dog timers UART Pulse width modulator LCD controllers Key pad controllers Stepper motor controllers A/D converters Real time clock.

UNIT 4 MEMORY AND INTERFACING 9 Hrs.

Memory write ability and storage performance Memory types composing memory Advance RAM interfacing communication basic Microprocessor interfacing I/O addressing Interrupts Direct memory access Arbitration multilevel bus architecture Serial protocol Parallel protocols Wireless protocols Digital camera example.

UNIT 5 PROCESS MODELS AND HARDWARE SOFTWARE CO-DESIGN 9 Hrs.

Modes of operation Finite state machine HCFSL and state charts language state machine models Concurrent process model Concurrent process Communication among process Synchronization among process Implementation – Data Flow mode

MAX. 45 Hrs.

Course Outcomes:

On completion of the course, student will be able to

CO1: Understand basic concepts of embedded systems hardware.

CO2: Implement the RTOS development tools in building real time embedded systems.

CO3: Develop the hardware for embedded system applications based on the processors.

CO4: Develop prototype circuit on breadboard including micro processor interfacing.

CO5: Design Hardware and Software using process models.

CO6: Develop and implement embedded based applications.

TEXT / REFERENCE BOOKS

1.David E.Simon, “An Embedded Software Primer”, Pearson Education,2001

2. Frank Vahid and Tony Gwargie, “ Embedded System Design”, John Wiley & Sons,2002

3. Steve Heath, “Embedded System Design”, Elsevier, Second Edition,2004.

4. Shibu.K.V, “Introduction to Embedded Systems”,Mc Graw Hill.

5. Raj Kamal, “Embedded Systems”,TMH.

6.Lyla, “Embedded Systems”,Pearson,2013.

7.Peter Marwadel, “Embedded System Design: Embedded Systems, Foundations of Cyber - Physical Systems, and the Internet of Things, Springer, Third Edition,2018.

8. Perry Xiao, “Designing Embedded Systems and the Internet of Things (IoT) with the ARM @ Mbed, John Wiley & Sons,2018.

9. Rob Toulson & Tim Wilmshurst, “Fast and Effective Embedded Systems Design,Second Edition: Applying the ARM mbed, Newnes,2018.

END SEMESTER EXAM QUESTION PAPER PATTERN

Max. Marks : 100 **Exam Duration : 3 Hrs.**

PART A : 10 Questions of 2 marks each-No choice **20 Marks**

PART B : 2 Questions from each unit with internal choice, each carrying 16 marks **80 Marks**

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

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT- 1 INTRODUCTION AND REVIEW OF EMBEDDED HARDWARE

UNIT 1 INTRODUCTION AND REVIEW OF EMBEDDED HARDWARE

Terminology Gates Timing diagram Memory Microprocessor buses Direct memory access Interrupts Built interrupts Interrupts basis Shared data problems Interrupt latency - Embedded system evolution trends Round-Robin Round Robin with interrupt function Rescheduling architecture algorithm.

1.1 Embedded systems terminology

Embedded systems are ubiquitous. These dedicated small computers are present in communications systems, vehicles, manufacturing machinery, detection systems, and many machines that make our lives easier.

The open nature of Android Linux and its availability for many different hardware architectures makes it an excellent candidate for embedded platforms.

The following are the most common concepts you should know while working with embedded devices.

Bootloader

A bootloader is a small piece of software that executes soon after you power up a computer. On adesktop PC, the bootloader resides on the master boot record (MBR) of the hard drive, and is executed after the PC BIOS performs various system initializations. The bootloader then passes system information to the kernel (for instance, the hard drive partition to mount as root) and then executes the kernel.

In an embedded system, the role of the bootloader is more complicated, since an embedded system does not have a BIOS to perform the initial system configuration. The low-level initialization of the microprocessor, memory controllers, and other board-specific hardware varies from board to board and CPU to CPU. These initializations must be performed before a kernel image can execute.

At a minimum, a bootloader for an embedded system performs the following functions:

- Initializes the hardware, especially the memory controller.
- Provides boot parameters for the operating system image.
- Starts the operating system image.

Additionally, most bootloaders also provide convenient features that simplify development andupdate of the firmware, such as:

- Reading and writing arbitrary memory locations.
- Uploading new binary images to the board's RAM via a serial line or Ethernet.
- Copying binary images from RAM to Flash memory.

Kernel

The kernel is the fundamental part of an operating system. It is responsible for managing theresources and the communication between hardware and software components.

The kernel offers hardware abstraction to the applications and provides secure access to the system memory. It also includes an interrupt handler that handles all requests or completed I/O operations.

Kernel modules

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without requiring a system reboot.

For example, one type of module is the device driver, which allows the kernel to access hardwareconnected to the system. Without these modules, Linux developers would have to build monolithic kernels and add new functionality directly into the kernel image. The result would be a large, cumbersome kernel. Another disadvantage of working without a kernel module is that you would have to rebuild and reboot the kernel every time you add new functionality.

In embedded systems, where functionality can be activated depending on the needs, kernel modules become a very effective way of adding features without enlarging the kernel image size.

Root file system

Operating systems normally rely on a set of files and directories. The root file system is the top of the hierarchical file tree. It contains the files and directories critical for system operation, including the device directory and programs for booting the system. The root file system also contains mount points where other file systems can be mounted to connect to the root file system hierarchy.

Applications

Software applications are programs that employ the capabilities and resources of a computer todo a particular task.

Applications make use of hardware devices by communicating with device drivers, which arepart of the kernel.

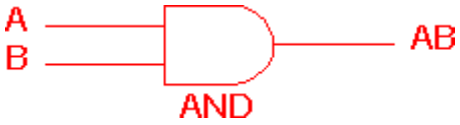
Cross-compilation

If you generate code for an embedded target on a development system with a different microprocessor architecture, you need a cross-development environment. A cross-development compiler is one that executes in the development system (for example, an x86 PC), but generates code that executes in a different processor (for example, if the target is ARM).

1.2 Logic gates

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

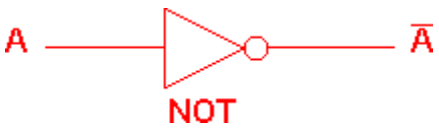
The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

OR gate

2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

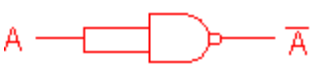
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of itsinputs are high. A plus (+) is used to show the OR operation.

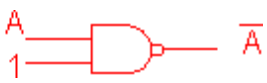
NOT gate



NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configuredto produce a NOT gate. It can also be done using NOR logic gates in the same way.





NAND gate

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. Theoutputs of all NAND gates are high if **any** of the inputs are low. The symbol is an ANDgate with a small circle on the output. The small circle represents inversion.

NOR gate

2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. Theoutputs of all NOR gates are low if **any** of the inputs are high.

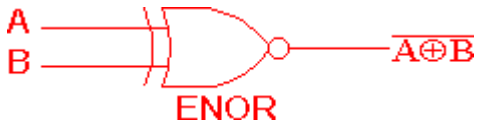
The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

EXOR gate

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign ( ) is used to show the EORoperation.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR**' gate circuit does the opposite to the EOR gate. It will give a low output if **either, but not both**, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

1.3 Timing Diagram

Timing Diagram is a graphical representation. It represents the execution time takenby each instruction in a graphical format. The execution time is represented in T-states.

Instruction Cycle:

The time required to execute an instruction is called instruction cycle.

or

The time taken by the processor to complete the execution of an instruction. Aninstruction cycle consists of one to six machine cycles.

Machine Cycle:

The time required to access the memory or input/output devices is called machine cycle.

or

The time required to complete one operation; accessing either the memory or I/Odevice. A machine cycle consists of three to six T-states.

T-State:

The machine cycle and instruction cycle takes multiple clock periods. A portion of an operation carried out in one system clock period is called as T-state.

or

Time corresponding to one clock period. It is the basic unit to calculate execution of instructions or programs in a processor.

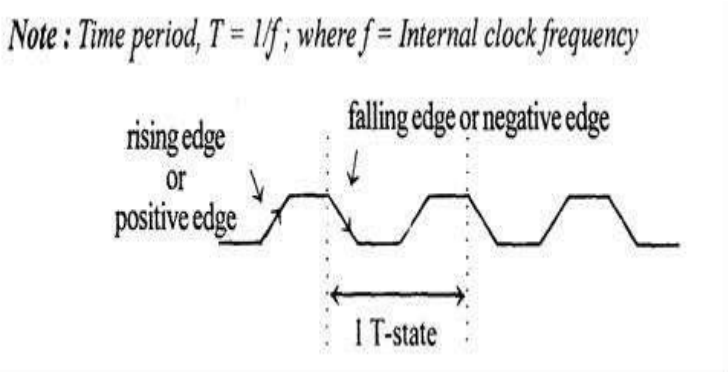
Fetch cycle:

The fetch cycle in a microprocessor comprises of several time states during which thenext instruction to be executed is copied (fetched) from the memory location (whose addressis in the Program Counter) to the Instruction Register.

Concept of Timing Diagram

The 8085 microprocessor has 5 (seven) basic machine cycles. They are

1. Opcode fetch cycle (4T)
2. Memory read cycle (3 T)
3. Memory write cycle (3 T)
4. I/O read cycle (3 T)
5. I/O write cycle (3 T)



- Each instruction of the 8085 processor consists of one to five machine cycles, i.e., when the8085 processor executes an instruction, it will execute some of the machine cycles in a specific order.

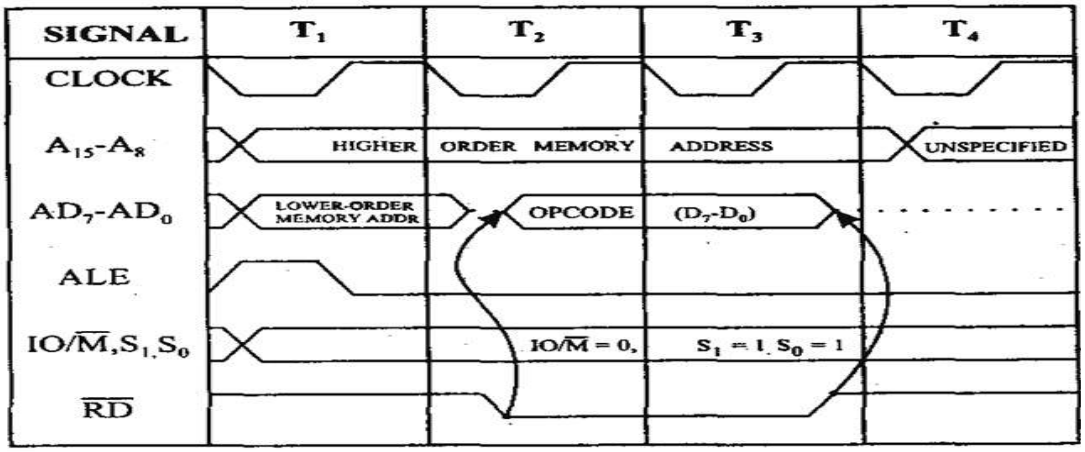
- The processor takes a definite time to execute the machine cycles. The time taken by the processor to execute a machine cycle is expressed in T-states.

- One T-state is equal to the time period of the internal clock signal of the processor.

- The T-state starts at the falling edge of a clock.

<u>Opcode Fetch Machine Cycle:</u>

- It is the first step in the execution of any instruction. The timing diagram of this cycle is givenbelow.



 The following points explain the various operations that take place and the signals that are changed during the execution of opcode fetch machine cycle:

T1 clock cycle:

 The content of PC is placed in the address bus; AD0 - AD7 lines contains lower bit addressand A8 – A15 contains higher bit address.

 IO/M' signal is low indicating that a memory location is being accessed. S1 and S0 alsochanged to the levels.

 ALE is high, indicates that multiplexed AD0 – AD7 act as lower order bus.

T2 clock cycle:

 Multiplexed address bus is now changed to data bus.

 The **(RD)'** signal is made low by the processor. This signal makes the memory device load thedata bus with the contents of the location addressed by the processor.

T3 clock cycle:

 The opcode available on the data bus is read by the processor and moved to the instruction register.

 The **(RD)'** signal is deactivated by making it logic 1.

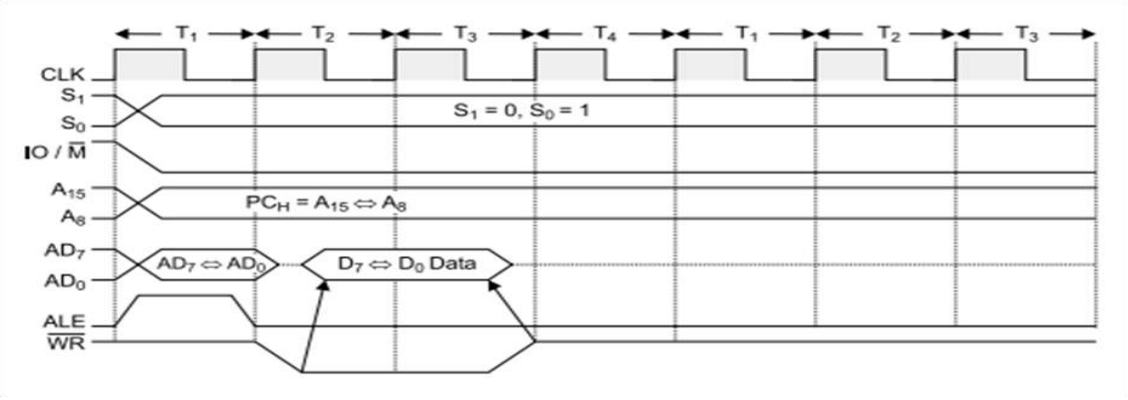
T4 clock cycle:

 The processor decode the instruction in the instruction register and generate the necessary control signals to execute the instruction. Based on the instruction further operations such as fetching, writing into memory etc. takes place.

DRAW TIMING DIAGRAM FOR MEMORY READ, MEMORY WRITE, I/O READ, I/O WRITE MACHINE CYCLE

Memory Read Machine Cycle:

 The memory read cycle is executed by the processor to read a data byte from memory. The machine cycle is exactly same to opcode fetch except: a) It has three T-states b) The S0 signalis set to 0.



T1 state:

- The higher order address bus (A8-A15) and lower order address and data multiplexed (AD0-AD7) bus.
- ALE goes high so that the memory latches the (AD0-AD7) so that complete 16-bit address

are available.

- The microprocessor identifies the memory read machine cycle from the status signals

IO/M'=0, S1=1, S0=0. This condition indicates the memory read cycle.

T2 state:

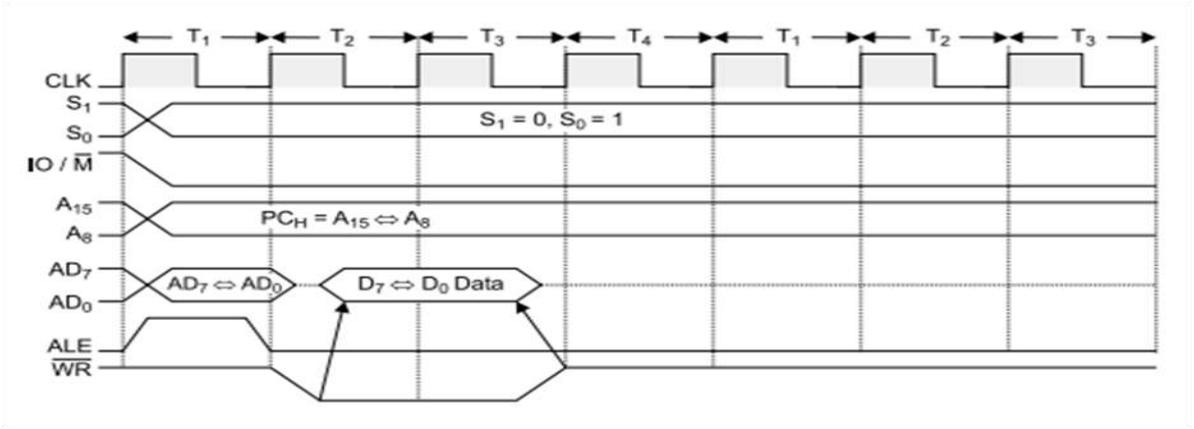
- Selected memory location is placed on the (D0-D7) of the A/D multiplexed bus. RD' goes LOW

T3 State:

- The data which was loaded on the previous state is transferred to the microprocessor.
- In the middle of the T3 state RD' goes high and disables the memory read operation.
- The data which was obtained from the memory is then decoded.

Memory Write Machine Cycle:

- The memory write cycle is executed by the processor to write a data byte in a memory location. The processor takes three T-states and **(WR)'**signal is made low.



T1 state:

- The higher order address bus (A8-A15) and lower order address and data multiplexed (AD0-AD7) bus.
- ALE goes high so that the memory latches the (AD0-AD7) so that complete 16-bit addressare available.
- The microprocessor identifies the memory read machine cycle from the status signals IO/M'=0, S1=0, S0=1. This condition indicates the memory read cycle.

T2 state:

- Selected memory location is placed on the (D0-D7) of the A/D multiplexed bus. WR' goes LOW

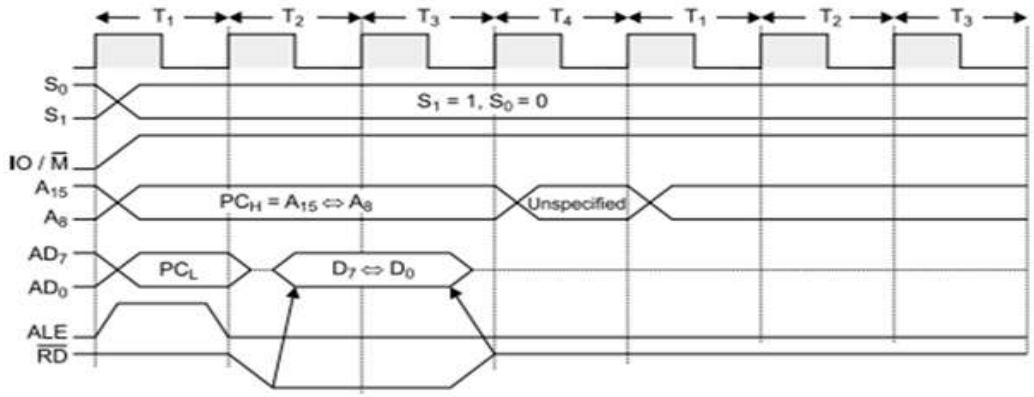
T3 State:

- In the middle of the T3 state WR' goes high and disables the memory write operation. The data which was obtained from the memory is then decoded.

I/O Read Cycle:

The I/O read cycle is executed by the processor to read a data byte from I/O port or from

peripheral, which is I/O mapped in the system. The 8-bit port address is placed both in the lower and higher order address bus. The processor takes three T-states to execute this machine cycle.



T1 state:

- The higher order address bus (A8-A15) and lower order address and data multiplexed (AD0-AD7) bus.
- ALE goes high so that the memory latches the (AD0-AD7) so that complete 16-bit address areavailable.
- The microprocessor identifies the I/O read machine cycle from the status signals IO/M'=1, S1=1, S0=0. This condition indicates the I/O read cycle.

T2 state:

- Selected memory location is placed on the (D0-D7) of the A/D multiplexed bus. RD' goes LOW

T3 State:

- The data which was loaded on the previous state is transferred to the microprocessor.
- In the middle of the T3 state RD' goes high and disables the I/O read operation.
- The data which was obtained from the I/O is then decoded.

I/O Write Cycle:

The I/O write cycle is executed by the processor to write a data byte to I/O port or toa peripheral, which is I/O mapped in the system. The processor takes three T-states to execute this machine cycle.

T1 state:

- The higher order address bus (A8-A15) and lower order address and data multiplexed (AD0-AD7) bus.
- ALE goes high so that the memory latches the (AD0-AD7) so that complete 16-bit address are available.
- The microprocessor identifies the I/O read machine cycle from the status signals IO/M'=1, S1=0, S0=1. This condition indicates the I/O read cycle.

T2 state:

- Selected memory location is placed on the (D0-D7) of the A/D multiplexed bus. WR' goes LOW

T3 State:

- In the middle of the T3 state WR' goes high and disables the I/O write operation. The data which was obtained from the I/O is then decoded.

1.4 Memory

Area where the program instruction and data are retained for processing is called memory, like human brain, computer also requires some space to store data and instruction for addressing their processing.

CPU does not have the capacity to store programs or large set of data permanently. It contains only basic instruction needed to operate the computer. Therefore memory is required.

Types of Memory

Memories primarily is of two types as given here:

- Random Access Memory (RAM)
  - Static RAM (SRAM)
  - Dynamic RAM (DRAM)
- Read Only Memory (ROM)
  - Masked Read Only Memory (MROM)
  - Programmable Read Only Memory (PROM)
  - Erasable and Programmable Read Only Memory (EPROM)
  - Electrically Erasable and Programmable Read Only Memory (EEPROM)

Random Access Memory (RAM)

A RAM constitutes the internal memory of the CPU for storing data, program and program result. It is read/write memory. It is called Random Access Memory (RAM).

Since access time in RAM is independent of the address to the word that is, each storage location inside the memory is as easy to reach as other location and takes the same amount of time. We can reach into the memory at random and extremely fast but can also be quite expensive.

RAM is volatile, that is data stored in it is lost when we switch off or turn off the computer or if there is a power Failure. Hence, a backup un-interruptible power system (UPS) is often used with computers. RAM is a small, both in terms of its physical size and in the amount of data that can hold.

Types of RAM

RAM is of two types:

1. Static RAM (SRAM)
2. Dynamic Ram (DRAM)

Static RAM (SRAM)

The word static indicates that the memory retains its contents as long as power remains applied.

However, data is lost when the power gets down due to volatile nature.

Static RAM chips use a matrix of 6 transistors and no capacitors.

Transistors do not require power to prevent leakage, so static RAM need not have to be refreshed on a regular basis. Because of the extra space in the matrix, static RAM uses more chips than dynamic RAM for the same amount of storage space, thus making the manufacturing costs higher.

Static RAM is used as cache memory needs to be very fast and small.

Dynamic Ram (DRAM)

Dynamic RAM, unlike static RAM, must be continually replaced in order for it to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second.

Dynamic RAM is used for most system memory because it is cheap and small.

All dynamic rams are made up of memory cells. These cells are composed of one capacitor and one transistor.

Read Only Memory (ROM)

ROM stands for read only memory. The memory from which we can only read but cannot write on it.

This type of memory is non-volatile. The information is stored permanently in such memories during manufacture.

A ROM, stores such instruction as are required to start computer when electricity is first turned on, this operation is referred to as bootstrap.

ROM chip are not only used in the computer but also in other electronic items like washing machine and microwave oven.

Types of ROM

The following list of ROM available in computer:

1. Masked Read Only Memory (MROM)
2. Programmable Read Only Memory (PROM)
3. Erasable and Programmable Read Only Memory (EPROM)
4. Electrically Erasable and Programmable Read Only Memory (EEPROM)

Masked Read Only Memory (MROM)

The very first ROMs were hardware devices that contained a pre-programmed set of data or instructions. This kind of ROMs are known as masked ROMs. Tt is inexpensive ROM.

Programmable Read Only Memory (PROM)

PROM is read only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM programmer.

Inside the PROM, there are small fuses which are burnt open during programming. It can be programmed only once and it's not erasable.

Erasable and Programmable Read Only Memory (EPROM)

The EPROM can be erased by exposing it to ultra-violet light for a duration of upto 40 minutes.

Usually, an EPROM eraser achieves this function. during programming, an electrical charge is trapped in an insulated Gate region.

The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultraviolet light is passed through a quartz crystal window (lid). This exposure to ultraviolet light dissipates the charge. During normal use the quartz lid is sealed with a sticker.

Electrically Erasable and Programmable Read Only Memory (EEPROM)

The EEPROM is programmed and erased electrically. It can be erased and re-programmed about ten thousand times.

Both erasing and programming take about 4 to 10 milliseconds. In EEPROM, any location can be selectively erased and programmed.

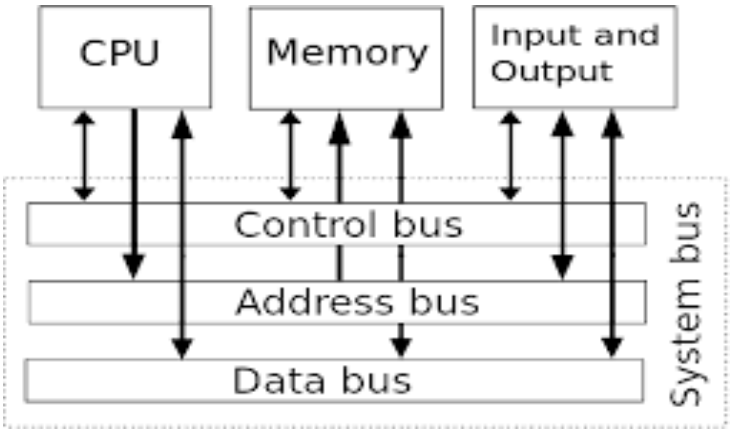
EEPROMs can be erased 1 byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

1.5 MICROPROCESSOR BUS

Bus is **a group of conducting wires which carries information**, all the peripherals are connected to microprocessor through Bus. Diagram to represent bus organization system of 8085 Microprocessor. There are three types of buses. It is a group of conducting wires which carries address only.

There are three types of buses in a microprocessor

- **Data Bus** − Lines that carry data to and from memory are called data bus. It is a bidirectional bus with width equal to word length of the microprocessor.
- **Address Bus** − It is a unidirectional responsible for carrying address of a memory location or I/O port from CPU to memory or I/O port.
- **Control Bus** − Lines that carry control signals like **clock signals, interrupt signal** or **ready signal** are called control bus. They are bidirectional. Signal that denotes that a device is ready for processing is called **ready signal**. Signal that indicates to a device to interrupt its process is called an **interrupt signal**.



1.6 DIRECT MEMORY ACCESS (DMA)

DMA is a technique for transferring blocks of data directly between two hardware devices.

In the absence of DMA the processor must read the data from one device and write it to the other one byte or word at a time.

DMA Absence Disadvantage: If the amount of data to be transferred is large or frequency of transfer is high the rest of the software might never get a chance to run.

DMA Presence Advantage: The DMA Controller performs entire transfer with little help from the Processor.

Working of DMA

The Processor provides the DMA Controller with source and destination address & total number of bytes of the block of data which needs transfer.

After copying each byte each address is incremented & remaining bytes are reduced by one.

When number of bytes reaches zeros the block transfer ends & DMA Controller sends an Interrupt to Processor.

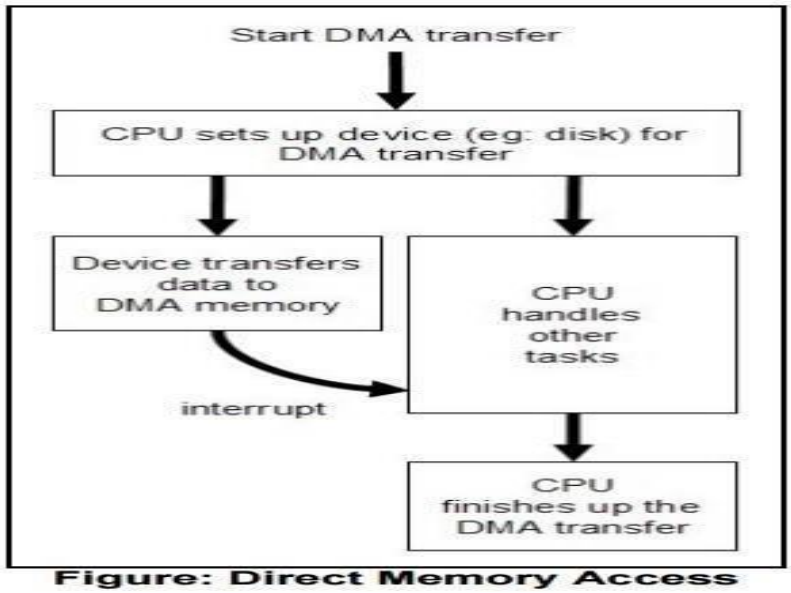


Figure: Direct Memory Access

1.7 INTERRUPT

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine** (ISR) or **Interrupt Handler**. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

Hardware Interrupt

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on thekeyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

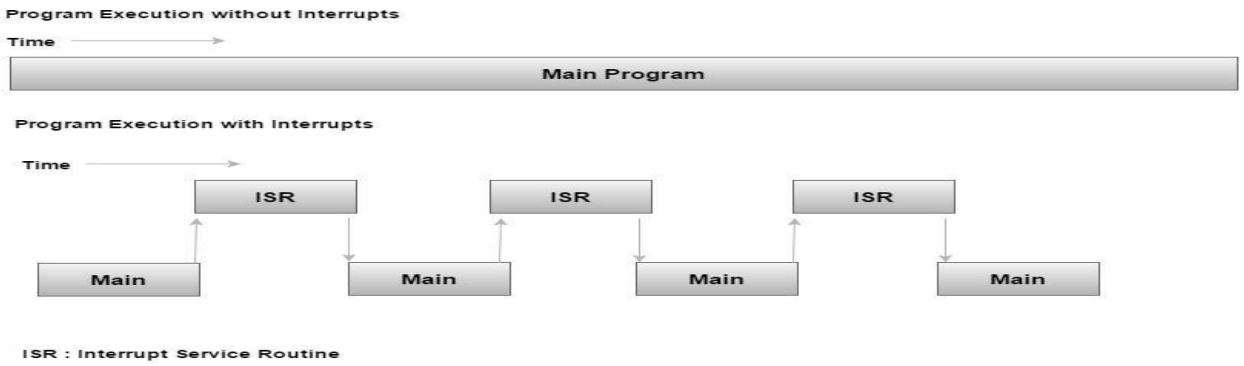
Software Interrupt

A software interrupt is caused either by an exceptional condition or a special instruction in the

instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or **interrupt handler**. When an interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR. The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.



Interrupt Vector Table

There are six interrupts including RESET in 8051.

Interrupts	ROM Location (Hex)	Pin
Interrupts	ROM Location (HEX)	
Serial COM (RI and TI)	0023	
Timer 1 interrupts(TF1)	001B	
External HW interrupt 1 (INT1)	0013	P3.3 (13)
External HW interrupt 0 (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
Reset	0000	9

- When the reset pin is activated, the 8051 jumps to the address location 0000. This is power-up reset.
- Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations are 000BH and 001BH respectively in the interrupt vector table.
- Two interrupts are set aside for hardware external interrupts. Pin no. 12 and Pin no. 13 inPort 3 are for the external hardware interrupts INT0 and INT1, respectively. Memory locations

are 0003H and 0013H respectively in the interrupt vector table.

- Serial communication has a single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps −

- The microcontroller closes the currently executing instruction and saves the address of the next instruction (PC) on the stack.
- It also saves the current status of all the interrupts internally (i.e., not on the stack).
- It jumps to the memory location of the interrupt vector table that holds the address of the interrupts service routine.
- The microcontroller gets the address of the ISR from the interrupt vector table and jumpsto it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).
- Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then, it start to execute from that address.

1.8 THE SHARED DATA PROBLEM

A big problem in embedded systems occurs in embedded software when an interrupt service routine and the main program share the same data. What happens if the main program is in the middle of doing some important calculations using some piece of data…an interrupt occurs that alters that piece of data…and then the main program finishes its calculation? Oops!

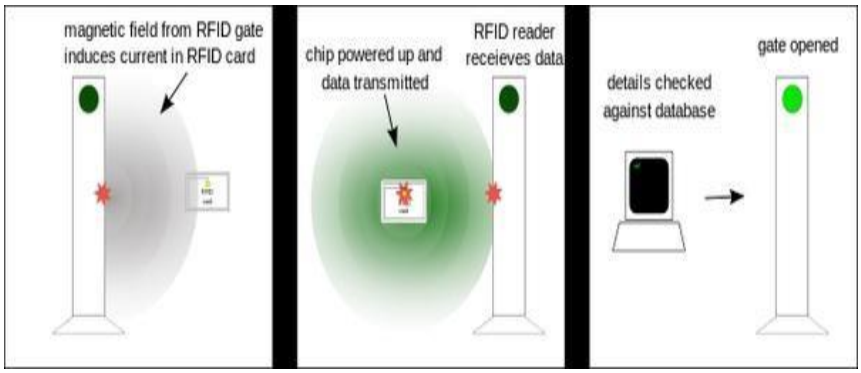
The calculation performed by the main program might be corrupted because it is based off the wrong/different data value. This is known as the shared data problem.

Example of Shared Data Problem

Imagine you are a software engineer working at a company. Your team is responsible for designing an automatic dog entry door. This embedded device can be wirelessly updated with RFID tags for dogs or other pets to be allowed entry.

The door needs to automatically unlock for dogs that are in the vicinity of the door. A pet must be allowed to enter even when the table of RFID tags is being updated. The RFID tag IDs are shared data since the interrupt service routine that must update the tag IDs and the main () program that is responsible for automatically unlocking the door when dogs are in the vicinity both share and use this data. A problem will occur when the doggy door is in the middle of an RFID tag ID update when a dog needs to get through the door. We wouldn't want to let the poor dog wait outside in the freezing cold while the device is in the middle of an RFID tag update!

How do we create a solution that solves the shared data problem? The RFID tags need to be updated regularly but that same data is needed regularly by the main () program to let dogs enter when they need to. Let's solve this now.

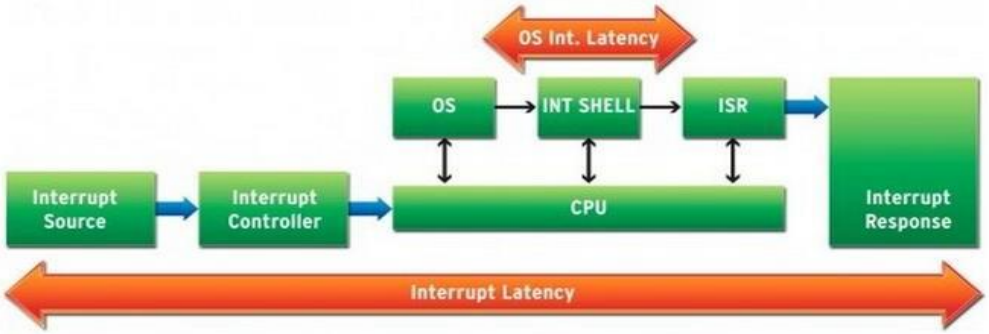


- This embedded device can be wirelessly updated with RFID tags.
- Dogs or other pets must be allowed entry when they are in the vicinity of the door.
- Dog must be allowed to enter even when the table of RFID tags is being updated.
- RFID tag IDs are shared data which must be managed.
- In the shared data problem for the doggy door controller, we need to make sure the dog can enter at all times while the RFID tags are being updated. Because this is a dog, it is unacceptable for the door to remain locked and keep a dog waiting.

1.9 INTERRUPTS LATENCY

Interrupt latency refers primarily to the software interrupt handling latencies. In other words, the amount of time that elapses from the time that an external **interrupt arrives** at the processor until the time that the **interrupt processing begins**.

One of the most important aspects of kernel real-time performance is the ability to service an interrupt request (IRQ) within a specified amount of time.



Here are the sources contributing the interrupt latency (abstracts from Reduce RTOS latency in interrupt-intensive apps):

Operating system (OS) interrupt latency

An RTOS must sometimes disable interrupts while accessing critical OS data structures. The maximum time that an RTOS disables interrupts is referred to as the OS interrupt latency. Although this overhead will not be incurred on most interrupts since the RTOS disables interrupts relatively infrequently, developers must always factor in this interrupt latency to understand the worst-case scenario.

Low-level interrupt-related operations

When an interrupt occurs, the context must be initially saved and then later restored after the interrupt processing has been completed. The amount of context that needs to be saved depends on how many registers would potentially be modified by the ISR (Interrupt Service Routine).

Enabling the ISR to interact with the RTOS

An ISR will typically interact with an RTOS by making a system call such as a semaphore post.

To ensure the ISR function can complete and exit before any context switch to a task is made, the RTOS interrupt dispatcher must disable preemption before calling the ISR function.

Once the ISR function completes, preemption is re-enabled and the application will context switch to the highest priority thread that is ready to run. If there is no need for an ISR to make an RTOS system call, the disable/enable kernel preemption operations would again add overhead. Itis logical to handle such an ISR outside of the RTOS.

Context switching

When an ISR defers processing to an RTOS task or other thread, a context switch needs to occur for the task to run. Context switching will still typically be the largest part of any-RTOS related interrupt processing overhead.

IRQ (Interrupt Request)

An (or IRQ) is a hardware signal sent to the processor that temporarily stops a running program and allows a special program, an interrupt handler, to run instead. Interrupts are used to handle such events as data receipt from a modem or network, or a key press or mouse movement.

FIQ (Fast Interrupt Request)

An FIQ is just a higher priority interrupt request, that is prioritized by disabling IRQ and other FIQ handlers during request servicing. Therefore, no other interrupts can occur during the processing of the active FIQ interrupt.

1.10 EMBEDDED SYSTEM EVOLUTION TRENDS

Embedded systems are on the rise as the technology paves the way for the future of smart manufacturing across a range of industries. Microcontrollers — the hardware at the center of embedded systems — are improving quickly, allowing for better machine control and monitoring. In this article, we will discuss the emerging trends for embedded systems in 2019that will enable enhanced security, better control, and improved scalability.

Current Trends in Embedded Systems Applications

An embedded system is an application-specific system designed with a combination of hardware and software to meet real-time constraints. The key characteristics of embedded industrial systems include speed, security, size, and power. The major trends in the embedded systems market revolve around the improvement of these characteristics.

To give context into how large the embedded systems industry is, here are a few statistics

- The global market for the embedded systems industry was valued at $68.9 billion in 2017 and is expected to rise to $105.7 billion by the end of 2025.
- 40% of the industrial share for embedded systems market is shared by the top 10 vendors.

- In 2015, embedded hardware contributed to 93% of the market share and it is expected to dominate the market over embedded software in the upcoming years as well.

Future Trends of Embedded Systems Industry

The industry for embedded systems is growing and there are still several barriers that must be overcome. Below are five notable trends of the embedded systems market for 2019.

Improved Security for Embedded Devices

With the rise of the Internet of Things (IoT), the primary focus of developers and manufacturers is on security. In 2019, advanced technologies for embedded security will emerge as key generators for identifying devices in an IoT network, and as microcontroller security solutions that isolate security operations from normal operations.

Cloud Connectivity and Mesh Networking

Getting embedded industrial systems connected to the internet and cloud can take weeks and months in the traditional development cycle. Consequently, cloud connectivity tools will be an important future market for embedded systems. These tools are designed to simplify the process of connecting embedded systems with cloud-based services by reducing the underlying hardwarecomplexities.

A similar yet innovative market for low-energy IoT device developers is Bluetooth mesh networks. These solutions can be used for seamless connectivity of nearby devices while reducing energy consumption and costs.

Reduced Energy Consumption

A key challenge for developers is the optimization of battery-powered devices for low power consumption and maximum uptime. Several solutions are under development for monitoring andreducing the energy consumption of embedded devices that we can expect to see in 2019. These include energy monitors and visualizations that can help developers fine-tune their embedded systems, and advanced Bluetooth and Wi-Fi modules that consume less power at the hardware layer.

Visualization Tools with Real Time Data

Developers currently lack tools for monitoring and visualizing their embedded industrial systemsin real time. The industry is working on real-time visualization tools that will give software engineers the ability to review embedded software execution. These tools will enable developers to keep a check on key metrics such as raw or processed sensor data and event-based context switches for tracking the performance of embedded systems.

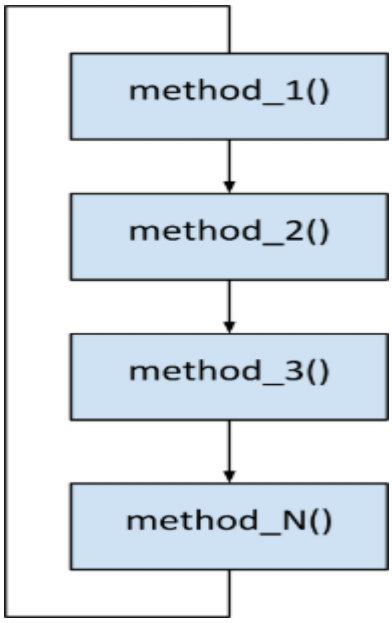
Deep Learning Applications

Deep learning represents a rich, yet unexplored embedded systems market that has a range of applications from image processing to audio analysis. Even though developers are primarily focused on security and cloud connectivity right now, deep learning and artificial intelligence concepts will soon emerge as a trend in embedded systems.

Embedded System Innovations

The industrial sector for embedded systems is undergoing numerous transformations that will enable developers to build systems that are high-performing, secure, and robust. As a developer and manufacturer in this industry, it is important to stay updated with the latest technologies and trends. For 2019, the embedded systems market is shaping up for simplified cloud connectivity, improved security tools, real-time visualizations, lower power consumption, and deep learning solutions.

1.11 ROUND ROBIN ARCHITECTURE



The Round Robin architecture is the easiest architecture for embedded systems. The main method consists of a loop that runs again and again, checking each of the I/O devices at each turn in order to see if they need service. No fancy interrupts, no fear of shared data…just a plain single execution time

Example: Multimeter

- very small number of I/O: (switch, display, probes)
- no particularly lengthy processing (even very simple microprocessors can check switch, take measurement and update display several times per second.)
- measurements can be taken at any time.
- display can be written to at any speed.
- small delays in switch position changes will go unnoticed thread that gets executed again and again.

Advantages:

- Simplest of all the architectures
- No interrupts
- No shared data

- No latency concerns
- No tight response requirements

Disadvantages:

- A sensor connected to the Arduino that urgently needs service must wait its turn.
- Fragile. Only as strong as the weakest link. If a sensor breaks or something else breaks,everything breaks.
- Response time has low stability in the event of changes to the code

Round-Robin Problems

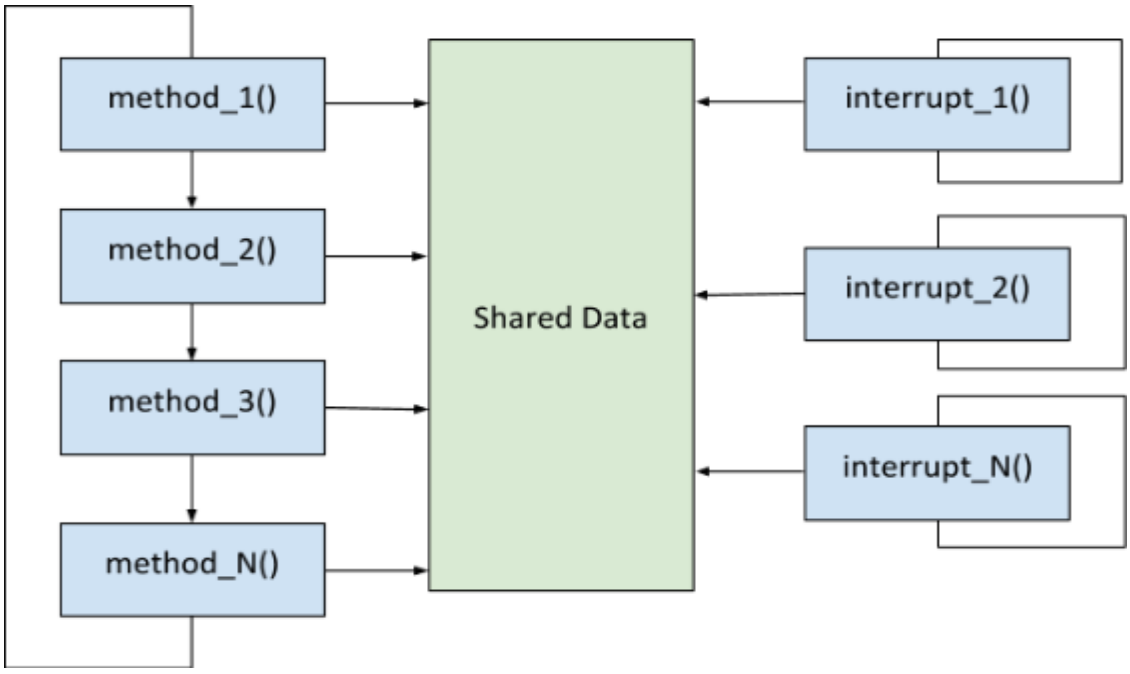
If any device needs a response in less time than the worst duration of the loop the system won't function.

If A and B take 5ms each and Z needs a response time of less than 7ms its not possible. This can be mitigate somewhat by doing (A,Z,B,Z) in a loop instead of (A,B,Z).

Scalability of this solution is poor. Even if absolute deadlines do not exist, overall response time may become unacceptably poor.

Round-Robin architecture is fragile – Even if the programmer manages to tune the loop sufficiently to provide a functional system a single addition or change can ruin everything.

Round Robin with Interrupts



This Round Robin with Interrupts architecture is similar to the Round Robin architecture, except

it has interrupts. When an interrupt is triggered, the main program is put on hold and control shifts to the interrupt service routine. Code that is inside the interrupt service routines has a higher priority than the task code.

Advantages:

- Greater control over the priority levels
- Flexible
- Fast response time to I/O signals
- Great for managing sensors that need to be read at prespecified time intervals

Disadvantages:

- Shared data
- All interrupts could fire off concurrently

PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING

Prerequisite – CPU Scheduling

1. Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queuetill it gets next chance to execute.

Algorithms based on preemptive scheduling are: Round Robin (RR),Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

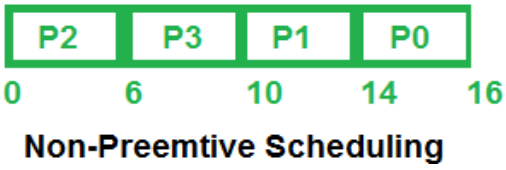
Preemptive Scheduling

2. Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state. In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU toanother process.

Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non preemptive) and Priority (non preemptive version), etc.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Non-Preemtive Scheduling

Key Differences between Preemptive and Non-Preemptive Scheduling:

1. In preemptive scheduling the CPU is allocated to the processes for the limited timewhereas in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.

2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and wait till its execution.

3. In Preemptive Scheduling, there is the overhead of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. Whereas in case of non- preemptive scheduling has no overhead of switching the process from running state to ready state.

4.In preemptive scheduling, if a high priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process having larger burst time then the processes with small burst time may have to starve.

5. Preemptive scheduling attain flexible by allowing the critical processes to access CPU

asthey arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queuethe process running CPU is not disturbed.

6. The Preemptive Scheduling has to maintain the integrity of shared data that's why it iscost associative as it which is not the case with Non-preemptive Scheduling.

Comparison Chart

Parameter	PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
Basic	In this resources (CPU Cycle) are allocated to a process for a limited time.	Once resources (CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process cannot be interrupted until it terminates itself or its time is up.
Starvation	If a process having high priority frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then later coming process with less CPU burst time may starve.
Overhead	It has overheads of scheduling the processes.	It does not have overheads.
Flexibility	Flexible	rigid
Cost	cost associated	no cost associated
CPU Utilization	In preemptive scheduling, CPU utilization is high.	It is low in non preemptive scheduling.
Examples	Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First.	Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First.

Part A

1.What is an embedded system? What are the components of embedded system?

2.What are the applications of an embedded system?

3.Interpret about embedded microcontroller.

4.What are the various classifications of embedded systems?

5.Define interrupt latency? How to avoid it.

6.Identify some of the hardware parts of embedded systems?

7.What are the various types of memory in embedded systems?

8.What are the requirements of embedded system?

9.Identify the functions of memory?

10.What is shared data problem?

11.Summarize the ways to eliminate Shared Data problem?

12.What is Round Robin Scheduling?

13.Compare round robin scheduling with and without interrupt.

14.Identify the functions of DMA

15.Interpret purpose of a bus?

Part B

1.Analyze in detail about the data transfer mechanism using DMA in Embedded System.

2.Explain in detail about Interrupt servicing Mechanism in an embeddeddevice.

3.Elaborate the basic processors and hardware units in the embedded system.

4.Explain in detail about interrupt latency and their solutions.

5.Appraise in detail about Round Robin Scheduling with and without interrupt. Give example also.

6.Explain in detail about shared data problem and how to avoid it. Give example

TEXT/ REFEENCE BOOKS

1.David E.Simon, “An Embedded Software Primer”, Pearson Education,2001

2. Frank Vahid and Tony Gwargie,“Embedded System Design”, John Wiley & Sons,2002

3. Steve Heath, “Embedded System Design”, Elsevier, Second Edition,2004.

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

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT- 2 REAL TIME OPERATING SYSTEM

UNIT 2 REAL TIME OPERATING SYSTEM

Task and Task states Task and data Semaphore and shared data operating system services Message queues timing functions Events Memory management Interrupt routines in an RTOS environment Basic design using RTOS.

Embedded systems are microcontroller-based systems that are designed to perform specific functions such as reading sensor data, responding to external events, communicating with other systems, controlling processes, etc. The tricky part is to make the distinction of what exactly qualifies such a system as **real-time**. Aren't all embedded systems operating in real-time? In order for an embedded system to be classified as real-time, it must guarantee a strictly defined response time to the events it is tasked with observing and controlling. It should be noted that all systems have a response time (latency). Real-time embedded systems do not react immediately to every **event but can guarantee a worse case response time.**

Real-time operating systems (RTOS) provide a framework that enables guaranteed response times and deterministic behaviour. This is achieved using a scheduling mechanism. This mechanism is at the heart of every RTOS. We can design a real -time embedded system without the use of RTOS, however, using one can make the design process shorter and the whole system easier to manage.

As part of the embedded system abstraction layers, an RTOS is placed above the low - level device drives and below the user application. The RTOS does not provide low - level drivers for microcontroller peripherals. Some RTOS may contain middleware software such as networking, file systems, etc.(fig 2.1)

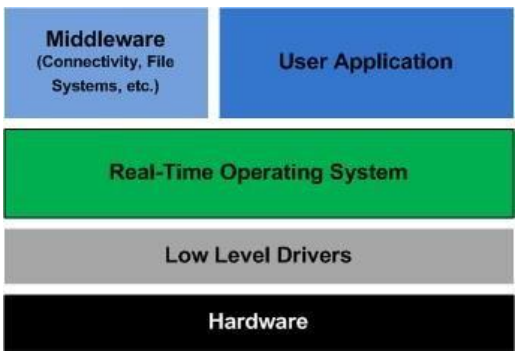


Fig.2.1 RTOS within the embedded system abstraction layers

2.1 Task and Task States

Tasks

Task is a piece of code or program that is separate from another task and can be executedindependently of the other tasks.

In embedded systems, the operating system has to deal with a limited number of tasksdepending on the functionality to be implemented in the embedded system.

Multiple tasks are not executed at the same time instead they are executed in pseudo parallel i.e. the tasks execute in turns as the use the processor.

From a multitasking point of view, executing multiple tasks is like a single book being read by multiple people, at a time only one person can read it and then take turns to read it. Different bookmarks may be used to help a reader identify where to resume reading next time.

An Operating System decides which task to execute in case there are multiple tasks to be executed. The operating system maintains information about every task and information aboutthe state of each task.

The information about a task is recorded in a data structure called the ***task context.*** When a task is executing, it uses the processor and the registers available for all sorts of processing. When a task leaves the processor for another task to execute before it has finished its own, it should resume at a later time from where it stopped and not from the first instruction. This requires the information about the task with respect to the registers of the processor to be storedsomewhere. This information is recorded in the task context.

A C++ version of a Task that holds all information needed by operating system is as follows:

```
class Task
{
        public:
                Task(void (*function)(), Priority p, int stackSize);

                TaskId id; Context context;
                TaskState state; Priority
                priority;
                int * pStack; Task * pNext;
```

```
        void (*entryPoint)();

    private:
        static TaskId nextId;
};
```

Task States

In an operation system there are always multiple tasks. At a time only one task can be executed. This means that there are other tasks which are waiting their turn to be executed.

Depending upon execution or not a task may be classified into the following three states (Fig 2.2):

Running state - Only one task can actually be using the processor at a given time that task is said to be the "running" task and its state is "running state". No other task can be in that same state at the same time

Ready state - Tasks that are not currently using the processor but are ready to run are in the "ready" state. There may be a queue of tasks in the ready state.

Waiting state - Tasks that are neither in running nor ready state but that are waiting for some event external to themselves to occur before the can go for execution on are in the "waiting" state**.**

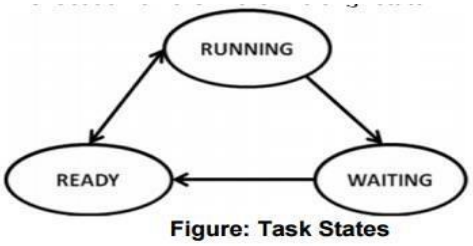


Fig 2.2 Task States

A transition of state between the ready and running state occurs whenever the operating system selects a new task to run.

The task that was previously in running state becomes ready and the new task is promoted to running state.

A task will leave running state only if it needs to wait for some event external to itself to occur before continuing.

A task's state can be defined as follows:

enum TaskState {Ready, Running, Waiting};

SCHEDULER

The heart and soul of any operating system is its scheduler.

This is the piece of the operating system that decides which of the ready tasks has the right to use the processor at a given time.

It simple checks to see if the running task is the highest priority ready task.Some of the more

common scheduling algorithms:

First-in-first-out

First-in-first-out (FIFO) scheduling describes an operating system which is not a multitasking operating system.

Each task runs until it is finished, and only after that is the next task started on a first come first served basis.

Shortest job first

Shortest job first scheduling uses algorithms that will select always select a task that will require the least amount of processor time to complete.

Round robin.

Round robin scheduling uses algorithms that allow every task to execute for a fixed amount to time.

A running task is interrupted an put to a waiting state if its execution time expires.

Scheduling Points

The scheduling points are the set of operating system events that result in an invocation of the scheduler.

There are three such events: **task creation** and **task deletion.** During each of these events a method is called to select the next task to be run.

A third scheduling point called the **clock tick** is a periodic event that is triggered by a timer interrupt. When a timer expires, all of the tasks that are waiting for it to complete are changed from the waiting state to the ready state.

Ready List

The scheduler uses a data structure called the **ready list** to track the tasks that are in the ready state.

The ready list is implemented as an ordinary linked list, ordered by priority. So the head of this

list is always the highest priority task that is ready to run.**Idle task**

If there are no tasks in the ready state when the scheduler is called, the idle task will be

executed.

The idle task looks the same in every operating system. The idle task is always considered to be in the ready state. **Scheduler**

The **scheduler** is an integral part of every RTOS. It controls which task should be executed at any given point in time. The scheduler may use various types of algorithms for performing the scheduling of the tasks. Almost all of these algorithms can be classified into two main types:

- **Preemptive Scheduling** – this algorithm allows the interruption of a currently running task, so another one with higher priority can be run.
- **Non-preemptive Scheduling** (**C-operative Scheduling**) – once a task is started it can't be interrupted, it will run until it decides that it should release the CPU to another task.

Advantages:

- **Better Structure and Scalability** – Using an RTOS gives you a well-defined mechanism for adding and removing software modules.
- **Timing Constraints** -Using RTOS makes it easier to fulfill the timing requirements of the many modules used in complex embedded systems.
- **Better Focus** – RTOS allows you to focus on the actual application by offloading the development of components such as memory management, exception handling, power management, etc.
- **Functional Safety** – There are RTOS distributions that are pre-certified for standards such as IEC 61508 and ISO 26262. This can greatly reduce the development effort in systems that must comply with such standards.

Disadvantages:

- **Learning Curve** – Even the simpler real-time operating systems will require time for learning their specifics and how to properly use them.
- **Price and Licensing** – Although there are many free RTOS, their licenses may differ a lot. If you want to use a free RTOS for commercial products there maybe some limitations or fees.

Popular real-time operating systems are Free RTOS, mBed, TinyOS, Riot, Zephyr, etc.

2.3 Semaphore

Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources.

To address these requirements, RTOS kernels provide a semaphore object and associated semaphore management services.

Semaphores

A semaphore (sometimes called a semaphore token) is a kernel object that one or more threadsof execution can acquire or release for the purposes of synchronization or mutual exclusion.

When a semaphore is first created, the kernel assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list, as shown in Figure 2.3 .

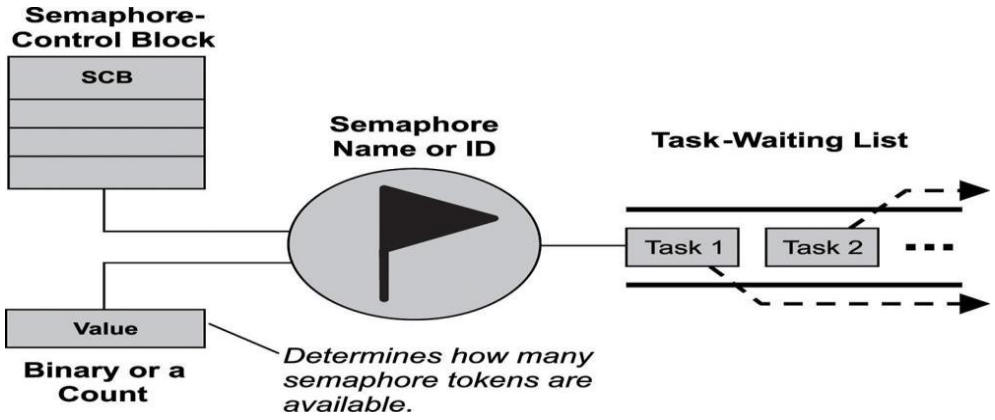


Fig 2.3 Semaphore

A semaphore is like a key that allows a task to carry out some operation or to access a resource. If the task can acquire the semaphore, it can carry out the intended operation or access the resource.

A single semaphore can be acquired a finite number of times.

In this sense, acquiring a semaphore is like acquiring the duplicate of a key from an apartment manager when the apartment manager runs out of duplicates, the manager can give out no more keys.

Likewise, when a semaphore's limit is reached, it can no longer be acquired until someone gives a key back or releases the semaphore.

The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created.

As a task acquires the semaphore, the token count is decremented; as a task releases the

semaphore, the count is incremented.

If the token count reaches 0, the semaphore has no tokens left.

A requesting task, therefore, cannot acquire the semaphore, and the task blocks if it chooses to wait for the semaphore to become available.

The task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore.

These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order orhighest priority first order.

When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it.

The kernel moves this unblocked task either to the running state, if it is the highest prioritytask, or to the ready state, until it becomes the highest priority task and is able to run.

Note that the exact implementation of a task-waiting list can vary from one kernel to another. A kernel can support many different types of semaphores, including binary, counting, andmutual-exclusion (mutex) semaphores.

1- Binary Semaphores :-

A binary semaphore can have a value of either 0 or 1.

When a binary semaphore's value is 0, the semaphore is considered unavailable (or empty);when the value is 1, the binary semaphore is considered available (or full).

Note that when a binary semaphore is first created, it can be initialized to either available orunavailable (1 or 0, respectively).

The state diagram of a binary semaphore is shown in Figure 2.4

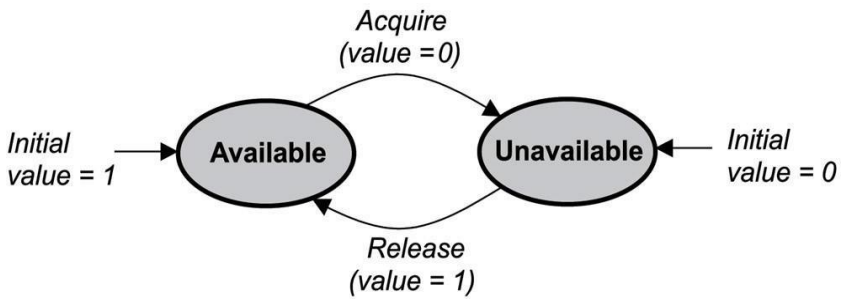


Fig 2.4 Binary Semaphore

Binary semaphores are treated as global resources, which means they are shared among alltasks that need them.

Making the semaphore a global resource allows any task to release it, even if the task did notinitially acquire it.

2- Counting Semaphores :-

A counting semaphore uses a count to allow it to be acquired or released multiple times.

When creating a counting semaphore, assign the semaphore a count that denotes the numberof semaphore tokens it has initially.
If the initial count is 0, the counting semaphore is created in the unavailable state.
If the count is greater than 0, the semaphore is created in the available state, and the number oftokens it has equals its count, as shown in Figure 2.5

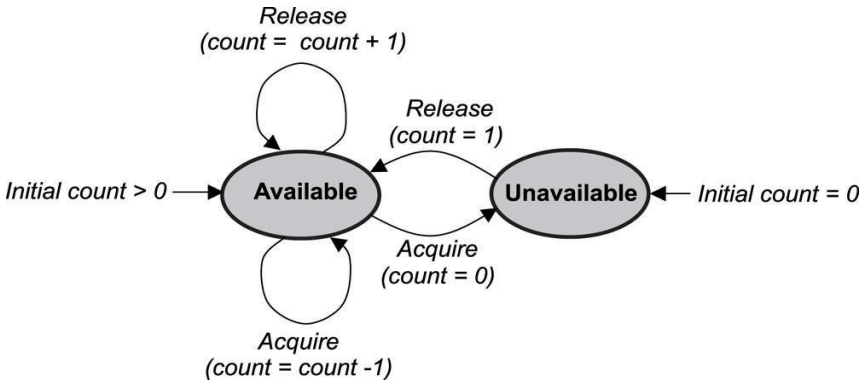


Fig 2.5 Counting Semaphore

One or more tasks can continue to acquire a token from the counting semaphore until no tokens are left.
When all the tokens are gone, the count equals 0, and the counting semaphore moves from theavailable state to the unavailable state.
To move from the unavailable state back to the available state, a semaphore token must bereleased by any task.
Note that, as with binary semaphores, counting semaphores are global resources that can beshared by all tasks that need them.
This feature allows any task to release a counting semaphore token.
Each release operation increments the count by one, even if the task making this call did notacquire a token in the first place.
Some implementations of counting semaphores might allow the count to be bounded.
A bounded count is a count in which the initial count set for the counting semaphore, determined when the semaphore was first created, acts as the maximum count for the semaphore.

An unbounded count allows the counting semaphore to count beyond the initial count to the maximum value that can be held by the count's data type (Ex :- an unsigned integer or an unsigned long value).

3- Mutual Exclusion (Mutex) Semaphores :-

A mutual exclusion (mutex) semaphore is a special binary semaphore thatsupports ownership, recursive access, task deletion safety, and one or more protocols foravoiding problems inherent to mutual exclusion.

Figure 2.6 illustrates the state diagram of a mutex.

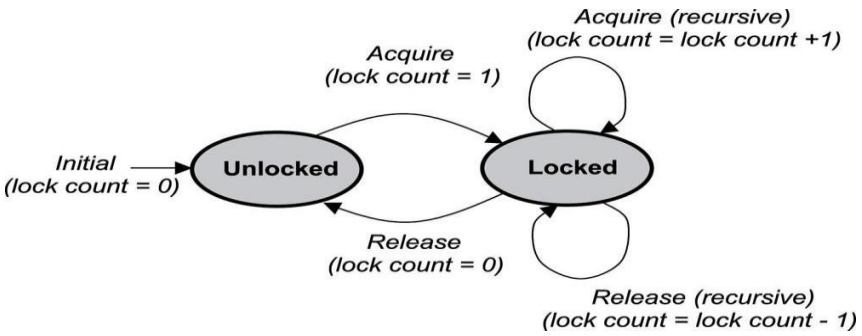


Fig 2.6 State diagram of a mutex

As opposed to the available and unavailable states in binary and counting semaphores, thestates of a mutex are unlocked or locked (0 or 1, respectively).

A mutex is initially created in the unlocked state, in which it can be acquired by a task.After being acquired, the mutex moves to the locked state.

Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Note that some kernels might use the terms lock and unlock for a mutex instead of acquire and release.

Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores.

These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.

Mutex Ownership :-

Ownership of a mutex is gained when a task first locks the mutex by acquiring it. Conversely, a task loses ownership of the mutex when it unlocks it by releasing it.

When a task owns the mutex, it is not possible for any other task to lock or unlock that mutex. Contrast this concept with the binary semaphore, which can be released by any task, even a task that did not originally acquire the semaphore.

Recursive Locking :-

Many mutex implementations also support recursive locking , which allows the task that ownsthe mutex to acquire it multiple times in the locked state.

Depending on the implementation, recursion within a mutex can be automatically built into the mutex, or it might need to be enabled explicitly when the mutex is first created.

The mutex with recursive locking is called a recursive mutex.

This type of mutex is most useful when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource.

A recursive mutex allows nested attempts to lock the mutex to succeed, rather than cause deadlock , which is a condition in which two or more tasks are blocked and are waiting on mutually locked resources.

As shown in the above figure, when a recursive mutex is first locked, the kernel registers the task that locked it as the owner of the mutex.

On successive attempts, the kernel uses an internal lock count associated with the mutex to track the number of times that the task currently owning the mutex has recursively acquired it. To properly unlock the mutex, it must be released the same number of times.

In this example, a lock count tracks the two states of a mutex (0 for unlocked and 1 for locked),as well as the number of times it has been recursively locked (lock count > 1).

In other implementations, a mutex might maintain two counts: a binary value to track its state, and a separate lock count to track the number of times it has been acquired in the lock state by the task that owns it.

Do not confuse the counting facility for a locked mutex with the counting facility for a counting semaphore.

The count used for the mutex tracks the number of times that the task owning the mutex has locked or unlocked the mutex.

The count used for the counting semaphore tracks the number of tokens that have been acquired or released by any task. Additionally, the count for the mutex is always unbounded, which allows multiple recursive accesses.

Task Deletion Safety :-

Some mutex implementations also have built-in task deletion safety.

Premature task deletion is avoided by using task deletion locks when a task locks and unlocksa mutex.

Enabling this capability within a mutex ensures that while a task owns the mutex, the taskcannot be deleted.

Typically protection from premature deletion is enabled by setting the appropriate initializationoptions when creating the mutex.

Priority Inversion Avoidance :-

Priority inversion commonly happens in poorly designed real-time embedded applications. Priority inversion occurs when a higher priority task is blocked and is waiting for a resourcebeing used by a lower priority task, which has itself been preempted by an unrelated medium-priority task.

In this situation, the higher priority task's priority level has effectively been inverted to thelower priority task's level.

Enabling certain protocols that are typically built into mutexes can help avoid priority inversion.

Two common protocols used for avoiding priority inversion include:-

A- Priority Inheritance Protocol :- ensures that the priority level of the lower priority task that has acquired the mutex is raised to that of the higher priority task that has requested the mutex when inversion happens. The priority of the raised task is lowered to its original value after the task releases the mutex that the higher priority task requires.

B- Ceiling Priority Protocol :- ensures that the priority level of the task that acquires the mutex is automatically set to the highest priority of all possible tasks that might request that mutex when it is first acquired until it is released.

When the mutex is released, the priority of the task is lowered to its original value.

2.4 Message Queue

A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data. A message queue is like a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them. This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.

A message queue has several associated components that the kernel uses to manage the queue. When a message queue is first created, it is assigned an associated queue control block (QCB), a message queue name, a unique ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists, as illustrated in Fig 2.7

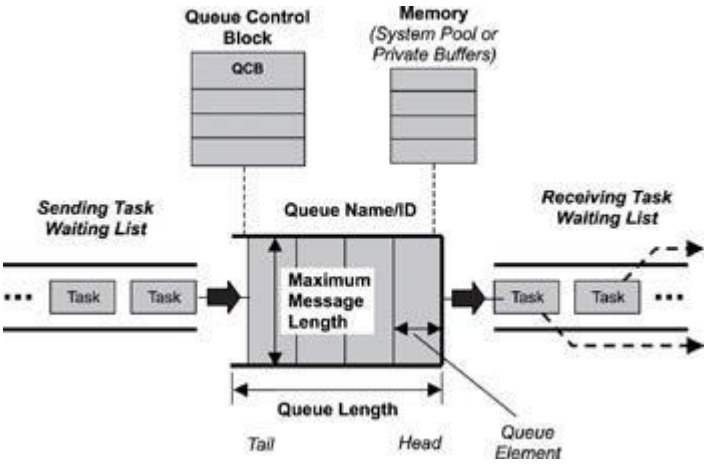


Figure 2.7 : A message queue, its associated parameters, and supporting data structures.

It is the kernel's job to assign a unique ID to a message queue and to create its QCB and task-waiting list. The kernel also takes developer-supplied parameters—such as the lengthof the queue and the maximum message length—to determine how much memory is required for the message queue. After the kernel has this information, it allocates memory for the message queue from either a pool of system memory or some private memory space.

The message queue itself consists of a number of elements, each of which can hold a single message. The elements holding the first and last messages are called the *head* and *tail* respectively. Some elements of the queue may be empty (not containing a message). The total number of elements (empty or not) in the queue is the *total length of the queue* . The developer specified the queue length when the queue was created.

As fig 2.7 shows, a message queue has two associated task-waiting lists. The receiving task-waiting list consists of tasks that wait on the queue when it is empty. The sending list consists of tasks that wait on the queue when it is full.

Message Queue States

As with other kernel objects, message queues follow the logic of a simple FSM, as shown in fig 2.8 When a message queue is first created, the FSM is in the empty state. If a task attempts to receive messages from this message queue while the queue is empty, the task blocks and, if it chooses to, is held on the message queue's task-waiting list, in either a FIFO or priority-based order.

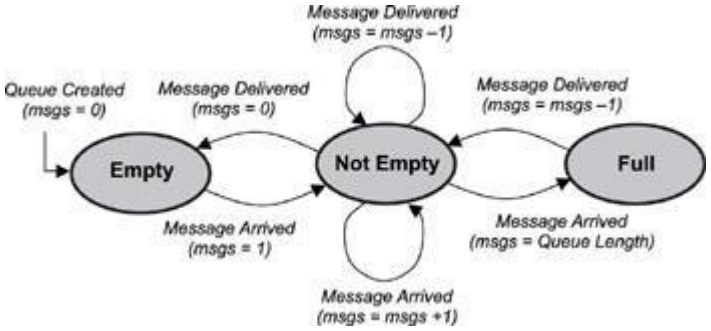


Figure 2.8 : The state diagram for a message queue.

In this scenario, if another task sends a message to the message queue, the message is delivered directly to the blocked task. The blocked task is then removed from the task-waiting list and moved to either the ready or the running state. The message queue in this case remains empty because it has successfully delivered the message.

If another message is sent to the same message queue and no tasks are waiting in the message queue's task-waiting list, the message queue's state becomes not empty.

As additional messages arrive at the queue, the queue eventually fills up until it has exhausted its free space. At this point, the number of messages in the queue is equal to the queue's length, and the message queue's state becomes full. While a message queue is in this state, any task sending messages to it will not be successful unless some other task first requests a message from that queue, thus freeing a queue element.

In some kernel implementations when a task attempts to send a message to a full message queue, the sending function returns an error code to that task. Other kernel implementations allow such a task to block, moving the blocked task into the sending task-waiting list, which is separate from the receiving task-waiting list (fig. 2.9).

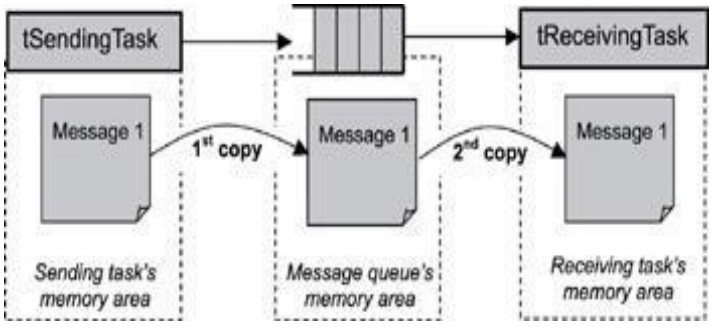


Figure 2.9: Message copying and memory use for sending and receiving messages.

Message Queue Content

Message queues can be used to send and receive a variety of data. Some examples include:

- a temperature value from a sensor,
- a bitmap to draw on a display,
- a text message to print to an LCD,
- a keyboard event, and
- a data packet to send over the network.

Some of these messages can be quite long and may exceed the maximum message length, which is determined when the queue is created. (Maximum message length should not be confused with total queue length, which is the total number of messages the queue can hold.) One way to overcome the limit on message length is to send a pointer to the data, rather than the data itself. Even if a long message might fit into the queue, it is sometimes better to send apointer instead in order to improve both performance and memory utilization.

When a task sends a message to another task, the message normally is copied twice, as shown in fig 2.9. The first time, the message is copied when the message is sent from the sending task's memory area to the message queue's memory area. The second copy occurs when the message is copied from the message queue's memory area to the receiving task's memory area. An exception to this situation is if the receiving task is already blocked waiting at the message queue. Depending on a kernel's implementation, the message might be copied just once in this case—from the sending task's memory area to the receiving task's memory area, bypassing the copy to the message queue's memory area.

Because copying data can be expensive in terms of performance and memory requirements, keep copying to a minimum in a real-time embedded system by keeping messages small or, if that is not feasible, by using a pointer instead.

Message Queue Storage

Different kernels store message queues in different locations in memory. One kernel might use a system pool, in which the messages of all queues are stored in one large shared area of memory. Another kernel might use separate memory areas, called private buffers, for each message queue.

System Pools

Using a system pool can be advantageous if it is certain that all message queues will never be filled to capacity at the same time. The advantage occurs because system pools typically save

on memory use. The downside is that a message queue with large messages can easily use most of the pooled memory, not leaving enough memory for other message queues. Indications that this problem is occurring include a message queue that is not full that starts rejecting messages sent to it or a full message queue that continues to accept more messages.

Private Buffers

Using private buffers, on the other hand, requires enough reserved memory area for the full capacity of every message queue that will be created. This approach clearly uses up more memory; however, it also ensures that messages do not get overwritten and that room is available for all messages, resulting in better reliability than the pool approach.

Typical Message Queue Operations

Typical message queue operations include the following:

- creating and deleting message queues,
- sending and receiving messages, and
- obtaining message queue information.

Typical Message Queue Use

The following are typical ways to use message queues within an application:

- non-interlocked, one-way data communication,
- interlocked, one-way data communication,
- interlocked, two-way data communication, and
- broadcast communication.

2.5 Interrupt routines in RTOS environment

ISRs have the higher priorities over the RTOS functions and the tasks. An ISR should not wait for a semaphore, mailbox message or queue message An ISR should not also wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run. Only the IPC accept function for these events (semaphore, mailbox, queue) can be used, not the post function

Interrupt Routine Rules

Interrupt routines in RTOS must follow two rules that do not apply to task code:

- An interrupt routine must not call any RTOS functions that might block.
- could block the highest priority task
- might not reset the hardware or allow further interrupts

- An interrupt routine must not call any RTOS function that might cause the RTOS to switchtasks
- causing a higher priority task to run may cause the interrupt routine to take a very long time tocomplete.

Low- and high-level ISRs

Low-level ISR

A low-level interrupt service routine (LISR) executes as a normal ISR, which includes using the current stack. Nucleus RTOS saves context before calling an LISR and restores context after the LISR returns. Therefore LISRs may be written in C and may call other C routines. However, there are only a few Nucleus RTOS services available to an LISR. If the interrupt processing requires additional Nucleus RTOS services, a high-level interrupt service routine (HISR) must be activated. Nucleus RTOS supports nesting of multiple LISRs.

High-level ISR

HISRs are created and deleted dynamically. Each HISR has its own stack space and its own control block. The memory for each is supplied by the application. Of course, the HISR must be created before it is activated by an LISR. Since an HISR has its own stack and control block, it can be temporarily blocked if it tries to access a Nucleus RTOS data structure that is already being accessed.

2.6 Memory Management

A kernel manages program code within an embedded system via tasks. The kernel must also have some system of loading and executing tasks within the system, since the CPU only executes task code that is in cache or RAM. With multiple tasks sharing the same memory space, an OS needs a security system mechanism to protect task code from other independent tasks. Also, since an OS must reside in the same memory space as the tasks it is managing, the protection mechanism needs to include managing its own code in memory and protecting it from the task code it is managing. It is these functions, and more, that are the responsibility ofthe memory management components of an OS. In general, a kernel's memory management responsibilities include:

- Managing the mapping between logical (physical) memory and task memory references.
- Determining which processes to load into the available memory space.
- Allocating and deallocating of memory for processes that make up the system.

- Supporting memory allocation and deallocation of code requests (within a process),
- such as the C language “alloc” and “dealloc” functions, or specific buffer allocation and
- deallocation routines.
- Tracking the memory usage of system components.
- Ensuring cache coherency (for systems with cache).
- Ensuring process memory protection.

Physical memory is composed of two-dimensional arrays made up of cells addressed by a unique row and column, in which each cell can store 1 bit.

Again, the OS treats memory as one large one-dimensional array, called a *memory map* . Eithera hardware component integrated in the master CPU or on the board does the conversion between logical and physical addresses (such as a *memory management unit (MMU)*), or it must be handled via the OS.

How OSs manage the logical memory space differs from OS to OS, but kernels generally run kernel code in a separate memory space from processes running higher level code (i.e., middleware and application layer code). Each of these memory spaces (*kernel* containing kernel code and *user* containing the higher-level processes) is managed differently. In fact, most OS processes typically run in one of two modes: *kernel mode* and *user mode* , dependingon the routines being executed. Kernel routines run in *kernel mode* (also referred toas *supervisor mode*), in a different memory space and level than higher layers of software suchas middleware or applications. Typically, these higher layers of software run in *user mode* , and can only access anything running in kernel mode via *system calls* , the higher-level interfaces to the kernel’s subroutines. The kernel manages memory for both itself and user processes.

User Memory Space

Because multiple processes are sharing the same physical memory when being loaded into RAM for processing, there also must be some protection mechanism so processes cannot inadvertently affect each other when being *swapped* in and out of a single physical memory space. These issues are typically resolved by the OS through memory “swapping,” where partitions of memory are *swapped* in and out of memory at runtime. The most common partitions of memory used in swapping are *segments* (fragmentation of processes from within) and *pages* (fragmentation of logical memory as a whole). Segmentation and paging not only simplify the swapping – memory allocation and deallocation – of tasks in memory, but allow for code reuse and memory protection, as well as providing the foundation for *virtual memory* . Virtual memory is a mechanism managed by the OS to allow a device’s limited memory space to be shared by multiple competing “user” tasks, in essence enlarging the device’s actual

physical memory space into a larger “virtual” memory space.

User Memory Space

Because multiple processes are sharing the same physical memory when being loaded into RAM for processing, there also must be some protection mechanism so processes cannot inadvertently affect each other when being *swapped* in and out of a single physical memory space. These issues are typically resolved by the OS through memory “swapping,” where partitions of memory are *swapped* in and out of memory at runtime. The most common partitions of memory used in swapping are *segments* (fragmentation of processes from within) and *pages* (fragmentation of logical memory as a whole). Segmentation and paging not only simplify the swapping – memory allocation and deallocation – of tasks in memory, but allow for code reuse and memory protection, as well as providing the foundation for *virtual memory* .Virtual memory is a mechanism managed by the OS to allow a device’s limited memory space to be shared by multiple competing “user” tasks, in essence enlarging the device’s actual physical memory space into a larger“ virtual” memory space.

Segmentation

A process encapsulates all the information that is involved in executing a program, including source code, stack, and data. All of the different types of information within a process are divided into “logical” memory units of variable sizes, called segments. A segment is a set of logical addresses containing the same type of information. Segment addresses are logical addresses that start at 0, and are made up of a segment number, which indicates the base address of the segment, and a segment offset, which defines the actual physical memory address. Segments are independently protected, meaning they have assigned accessibility characteristics, such as shared (where other processes can access that segment), read-only, or read/write.

Most OSs typically allow processes to have all or some combination of five types of information within segments: text (or code) segment, data segment, BSS (block started by symbol) segment, stack segment, and the heap segment. A text segment is a memory space containing the source code. A data segment is a memory space containing the source code’s initialized variables (data). A BSS segment is a statically allocated memory space containing the source code’s un-initialized variable (data). The data, text, and BSS segments are all fixed in size at compile time, and are as such static segments; it is these three segments that typically are part of the executable file.

Executable files can differ in what segments they are composed of, but in general they contain a header, and different sections that represent the types of segments, including name, permissions, etc., where a segment can be made up of one or more sections.

The OS creates a task's image by memory mapping the contents of the executable file, meaning loading and interpreting the segments(sections) reflected in the executable into memory. There are several executable file formats supported by embedded OSs, the most common including:

2.7 Basic Design Using RTOS

Most operating systems are put together based on kernel designs. Kernel design has been used for almost 4 decades because it separates the operating system from the different applications running on it. The different applications are allocated in different memory locations. The OS processes utilize kernel functionality through conducting system calls. System calls are software interrupts that allow users to switch from the operating system to applications and vice versa. Therefore, the kernel must install an interrupt handler that tackles different modes of operation in order to ensure effective switches. The interrupt handler is enabled in the program status (i.e., the supervisor mode and user mode). As such, protection is conducted on the modern system on a chip (SoCs) at the peripheral side. However, some processor registers can be changed if the CPU indicates a particular execution mode like master mode through additional HW signals.

All processes outside the operating system are implemented within the user mode and cannot execute any instructions availed in supervisor mode only. Meaning that user mode instructions hold a non-critical subset of instructions under the supervisor mode. During a process runtime, the supervisor mode under the PSW is disabled and only gets enabled once an interrupt like external interrupt or system call occurs. The OS activates the user mode once the user process is activated. Note that, a user process contains a virtual memory address space that separates it from the kernel entirely. However, this feature is only available to embedded microcontrollersthat constitute a memory management unit that allows the use of virtual memory. Virtual memory usage must be upheld without other unbound memory accesses such as swapping on an external disk or changing (TLB) translation lookaside buffer entries by examining a dynamically sized page table.

To utilize the functionality offered by the OS kernel design, you must identify an interface that allows applications to run effectively while using it. The interface is known as the application binary interface (ABI). ABI delineates a registered usage convention, a set of system calls, a stack layout and facilitates binary compatibility. On the other hand, an API (application programming interface) facilitates source code compatibility by defining a set of function signatures that offer a fixed interface for calling the required functions. The kernel can have

many designs, but it must provide basic activities like; process communication, process synchronization, process management and interrupt handling.

Process management ensures that process termination, creation, dispatching, scheduling, and switching context among other related activities run as required. In a real-time operating system, interrupt handling differs from the standardized implementation of a regular operating system. Interrupts in regular operating systems can preempt all running processesunexpectedly. This leads to unbound delays that are intolerable in a real-time operating system. As such, handling of interruptions is assimilated into the scheduler so that it is scheduled alongwith other important processes and feasibility is guaranteed even when interruption requests are made.

Part A

1.Define task and Task state.
2.Define Task Control Block
3.Define Inter process communication
4.Define Semaphore.
5.Interpret Priority inversion?
6.Define Message Queue.
7.List the functions of a kernel.
8.What is a thread?
9.What are the problems of semaphore?
10.What is memory management in embedded system?
11.What is ISR?

Part B

1.Explain in detail about semaphores and its applications.
2.What is IPC? Mention the two methods available for it.
3.Explain in detail about messagequeues.
4.Discuss in detail about the following. A) Timer function events.
management functions.
5.Elaborate in detail about task and task state with suitable diagra

TEXT/ REFEENCE BOOKS

1.David E.Simon, “An Embedded Software Primer”, Pearson Education,2001
2. Frank Vahid and Tony Gwargie, “ Embedded System Design”, John Wiley & Sons,2002
3. Steve Heath, “Embedded System Design”, Elsevier, Second Edition,2004.

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT-3 EMBEDDED HARDWARE, SOFTWARE AND PERIPHERAL

UNIT 3 EMBEDDED HARDWARE, SOFTWARE AND PERIPHERAL

Custom single purpose processors: Hardware Combination Sequence Processor design RT level design optimizing software: Basic Architecture Operation Programmers view Development Environment ASIP Processor Design Peripherals Timers, counters and watch dog timers UART Pulse width modulator LCD controllers Key pad controllers Stepper motor controllers A/D converters Real time clock.

3.1 Custom single-purpose processors: Hardware

A single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single- purpose processor to execute a specific task within our embedded system. An embedded system designerchoosing to use a custom single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits.

First, performance may be fast, due to fewer clock cycles resulting from a customized data path, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler data path and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task.

However, because we probably won't manufacture as many of the custom processor as a standard processor, we may not be able to invest as much NRE, unless the embedded system we are building will be sold in large quantities or does not have tight cost constraints. This fact could actually penalize performance and size.

Combinational logic design

A transistor is the basic electrical component of digital systems. Combinations of transistors form more abstract components called logic gates, which designers primarily use when

building digital systems. Thus, we begin with a short description of transistors before discussing logic design.

A transistor acts as a simple on/off switch. One type of transistor (CMOS -- Complementary Metal Oxide Semiconductor) is shown in Figure 3.1(a). The *gate* (not to be confused with logic gate) controls whether or not current flows from the *source* to the *drain*. When a high voltage (typically +5 Volts, which we'll refer to aslogic 1) is applied to the gate, the transistor conducts, so current flows. When low voltage (which we'll refer to as logic 0, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustratedin in Figure 3.1(b). When logic 0 is applied to the gate, the transistor conducts, and when logic 1 is applied, the transistor does not conduct. Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in in Figure 3.1(c). When the input *x* is logic 0, the top transistor conducts (and the bottom does not), so logic 1 appears at the output *F*. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic 0, as shown in Figure 3.1(d). When at least one of the inputs *x* and *y* is logic 0, then at least one of the top transistors conducts (and the bottom transistors do not), sologic 1 appears at *F*. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones do, so logic 0 appears at *F*. Likewise, we can easily build a circuit whose output is logic 1 when both of its inputs are logic 0, as illustrated in Figure 3.1(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

Figure 3.1: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

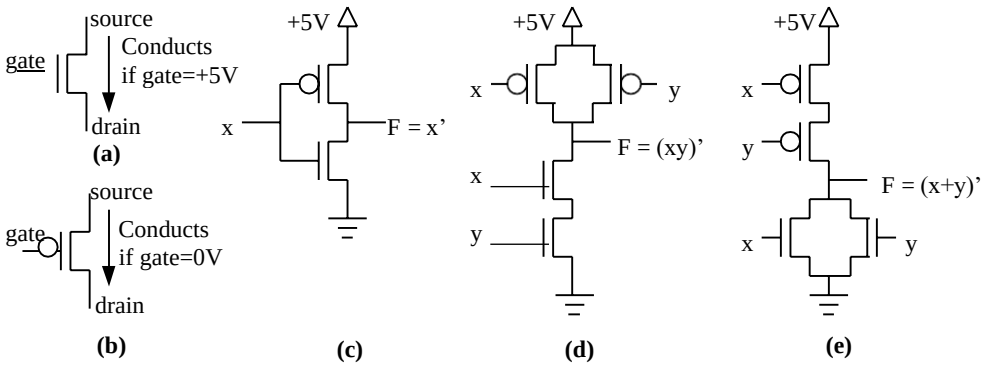
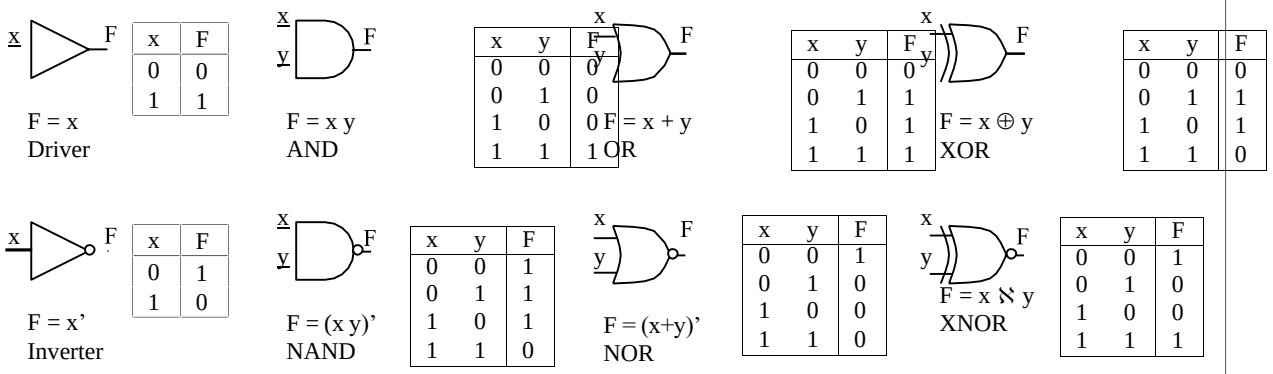


Figure 3.2: Basic logic gates



x	F
0	0
1	1

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

x	F
0	1
1	0

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

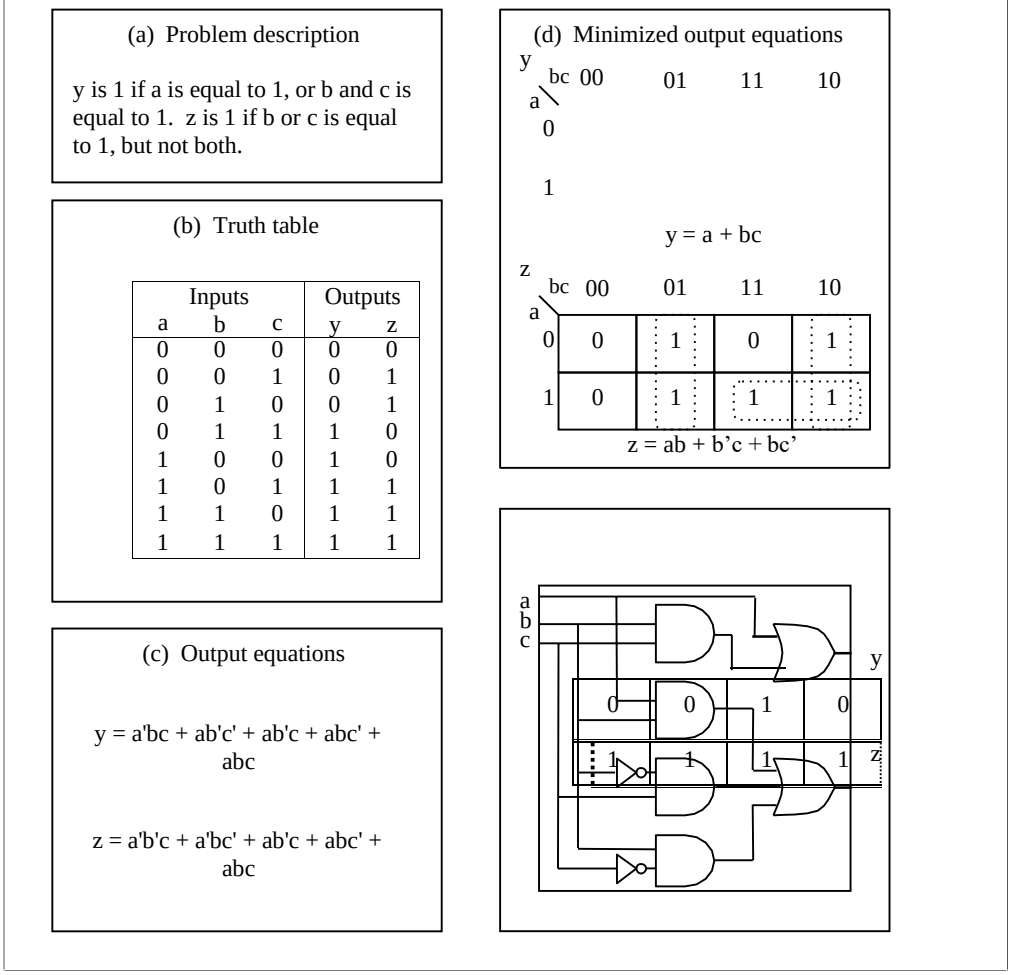
x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Digital system designers usually work with logic gates, not transistors. Figure 3.2 describes 8 basic logic gates. Each gate is represented symbolically, with a Booleanequation, and with a truth table. The truth table has inputs on the left, and output on the right. The AND gate outputs 1 if and only if both inputs are 1. The OR gate outputs 1 if and only if at least one of the inputs is 1. The XOR (exclusive-OR) gateoutputs 1 if and only if exactly one of its two inputs is 1. The NAND, NOR, and XNOR gates output the complement of AND, OR, and XOR, respectively. As you might have noticed from our transistor implementations, the NAND and NOR gatesare actually simpler to build than AND and OR gates.

A combinational circuit is a digital circuit whose output is purely a function of its current inputs; such a circuit has no memory of past inputs. We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in Figure 3.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often want to minimize the logic gates in the circuit. We can minimize the output equations by algebraically manipulating the equations. Alternatively, we can use Karnaugh maps, as shown in the figure. Once we've obtained the desired output equations (minimized or not), we can draw the circuitdiagram.

Figure 3.3: Combinational logic design.



Although we can design all combinational circuits in the above manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2^{16}, or 64K, rows in its truth table. One way to reduce the complexity isto use components that are more abstract than logic gates. Figure 3.4 shows several such combinational components. We now describe each briefly.

A multiplexor, sometimes called a selector, allows only one of its data inputs Im to pass through to the output O. Thus, a multiplexor acts much like a railroad switch, allowing only one of multiple input tracks to connect to a single outputtrack. If there are m data inputs, then there are $\log_2(m)$ select lines S, and we call this an m-by-1 multiplexor (m data inputs, one data output). The binary value of S determines which data input passes through; 00...00 means I0 may pass, 00...01 means I1 may pass, 00...10 means I2 may pass, and so on. For

example, an 8x1 multiplexor has 8 data inputs and thus 3 select lines. If those three select lines have values of 110, then I6 will pass through to the output. So if I6 is 1, then the output would be 1; if I6 is 0, then the output would be 0. We commonly use a more complex device called an n-bit multiplexor, in which each data input, as well as the output, consists of n lines. Suppose the previous example used a 4-bit 8x1 multiplexor. Thus, if I6 is 0110, then the output would be 0110. Note that n does not affect the number of select lines.

A decoder converts its binary input I into a one-hot output O. "One-hot" meansthat exactly one of the output lines can be 1 at a given time. Thus, if there are n outputs, then there must be $\log_2(n)$ inputs. We call this a $\log_2(n)$xn decoder. For example, a 3x8 decoder has 3 inputs and 8 outputs. If the input is 000, then the output O0 will be 1. If the input is 001, then the output O1 would be 1, and so on. Acommon feature on a decoder is an extra input called enable. When enable is 0, all outputs are 0. When enable is 1, the decoder functions as before.

An adder adds two n-bit binary inputs A and B, generating an n-bit output sum along with an output carry. For example, a 4-bit adder would have a 4-bit A input, a4-bit B input, a 4-bit sum output, and a 1-bit carry output. If A is 1010 and B is 1001, then sum would be 0011 and carry would be 1.

A comparator compares two n-bit binary inputs A and B, generating outputs that indicate whether A is less than, equal to, or greater than B. If A is 1010 and Bis 1001, then less would be 0, equal would be 0, and greater would be 1.

An ALU (arithmetic-logic unit) can perform a variety of arithmetic and logic functions on its n-bit inputs A and B. The select lines S choose the current function;

if there are m possible functions, then there must be at least $\log_2(m)$ select lines.
Common functions include addition, subtraction, AND, and OR.

STANDARD SINGLE-PURPOSE PROCESSORS: PERIPHERALS

Introduction

A single-purpose processor is a digital system intended to solve a specific computation task. The processor may be a standard one, intended for use in a wide variety of applications in which the same task must be performed. Themanufacturer of such an off-the-shelf processor

sells the device in large quantities. On the other hand, the processor may be a custom one, built by a designer to implement a task specific to a particular application. An embedded system designerchoosing to use a standard single- purpose, rather than a general-purpose, processorto implement part of a system's functionality may achieve several benefits.

First, performance may be fast, since the processor is customized for theparticular task at hand. Not only might the task execute in fewer clock cycles, but also those cycles themselves may be shorter. Fewer clock cycles may result from many data path components operating in parallel, from data path components passing data directly to one another without the need for intermediate registers (chaining), or from elimination of program memory fetches. Shorter cycles may result from simpler functional units, less multiplexors, or simpler control logic. For standard single-purpose processors, manufacturers may spread NRE cost over many units. Thus, the processor's clock cycle may be further reduced by the use of custom IC technology, leading-edge IC's, and expert designers, just as is the case with general-purpose processors.

Second, size may be small. A single-purpose processor does not require a program memory. Also, since it does not need to support a large instruction set, it may have a simpler data path and controller.

Third, a standard single-purpose processor may have low unit cost, due to the manufacturer spreading NRE cost over many units. Likewise, NRE cost may be low, since the embedded system designer need not design a standard single-purpose processor, and may not even need to program it.

Timers, counters, and watchdog timers

A timer is a device that generates a signal pulse at specified time intervals. A time interval is a "real-time" measure of time, such as 3 milliseconds. These devices are extremely useful in systems in which a particular action, such as sampling an input signal or generating an output signal, must be performed every X time units.

Internally, a simple timer may consist of a register, counter, and an extremely simple controller. The register holds a count value representing the number of clock cycles that equals the desired real-time value. This number can be computedusing the simple formula:

Number of clock cycles = Desired real-time value / Clock cycle

For example, to obtain a duration of 3 milliseconds from a clock cycle of 10 nanoseconds (100 MHz), we must count ($3x10^{-6}$ s / $10x10^{-9}$ s/cycle) = 300 cycles. The counter is initially loaded with the count value, and then counts down on every clock cycle until 0 is reached, at which point an output signal is generated, the count value is reloaded, and the process repeats itself.

To use a timer, we must configure it (write to its registers), and respond to its output signal. When we use a timer in conjunction with a general-purpose processor, we typically respond to the timer signal by assigning it to an interrupt, so we include the desired action in an interrupt service routine. Many microcontrollers that include built-in timers will have special interrupts just for itstimers, distinct from external interrupts.

Note that we could use a general-purpose processor to implement a timer. Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on adedicated general-purpose processor is obviously quite inefficient in terms of size.One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timerfunctionality to a special-purpose processor becomes evident.

A counter is nearly identical to a timer, except that instead of counting clock cycles (pulses on the clock signal), a counter counts pulses on some otherinput signal.

A watchdog timer can be thought of as having the inverse functionality than that of a regular timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, *we* must generate a signal for the timer every X time units. If we fail to generate this signal in time, then the timer generates a signal indicating that we failed. We often connect this signal to the reset or interrupt signal of a general-purpose processor. Thus, a watchdog timer provides a mechanism ofensuring that our software is working properly; every so often in the software, we include a statement that generates a signal to the watchdog timer (in particular, that resets the timer). If something undesired happens in the software (e.g., we enter an undesired infinite

loop, we wait for an input signal that never arrives, a part fails, etc.), the watchdog generates a signal that we can use to restart or test parts of the system. Using an interrupt service routine, we may record informationas to the number of failures and the causes of each, so that a service technician may later evaluate this information to determine if a particular part requires replacement. Note that an embedded system often must recover from failures whenever possible, as the user may not have the means to reboot the system in thesame manner that he/she might reboot a desktop system.

UART

A UART (Universal Asynchronous Receiver/Transmitter) receives serial dataand stores it as parallel data (usually one byte), and takes parallel data and transmits it as serial data.

Such serial communication is beneficial when we need to communicate bytes of data between devices separated by long distances, or when we simply have fewavailable I/O pins. We must be aware that we must set the transmission and reception rate, called the baud rate, which indicates the frequency that the signal changes. Common rates include 2400, 4800, 9600, and 19.2k. We must also be aware that an extra bit may be added to each data word, called parity, to detect transmission errors -- the parity bit is set to high or low to indicate if the word has an even or odd number of bits.

Internally, a simple UART may possess a baud-rate configuration register, and two independently operating processors, one for receiving and the other for transmitting. The transmitter may possess a register, often called a transmit buffer, that holds data to be sent. This register is a shift register, so the data can betransmitted one bit at a time by shifting at the appropriate rate. Likewise, the receiver receives data into a shift register and then this data can be read in parallel. Note that in order to shift at the appropriate rate based on the configuration register, a UART requires a timer.

To use a UART, we must configure its baud rate by writing to the configuration register, and then we must write data to the transmit register and/or read data from the received register. Unfortunately, configuring the baud rate is usually not as simple as writing the desired rate (e.g., 4800) to a register. For example, to configure the UART of an 8051, we must use the following equation:

$$Baudrate = (2^{s\,\mathrm{mod}} / 32) * oscfreq / (12 * (256 - TH1)))$$

smod corresponds to 2 bits in a special-function register,

oscfreq is the frequency of the oscillator, and

TH1 is an 8-bit rate register ofa built-in timer.

Note that we could use a general-purpose processor to implement a UART completely in software. If we used a dedicated general-processor, the implementation would be inefficient in terms of size. We could alternatively integrate the transmit and receive functionality with our main program. This would require creating a routine to send data serially over an I/O port, making use of a timer to control the rate. It would also require using an interrupt service routine to capture serial data coming from another I/O port whenever such data begins arriving. However, as with the timer functionality, adding send and receive functionality can detract from time for other computations.

Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special- purpose processor becomes evident.

Pulse width modulator

A pulse-width modulator (PWM) generates an output signal that repeatedly switches between high and low. We control the duration of the high value and of the low value by indicating the desired period, and the desired duty cycle, which is the percentage of time the signal is high compared to the signal's period. A square wave has a duty cycle of 50%. The pulse's width corresponds to the pulse's timehigh.

Again, PWM functionality could be implemented on a dedicated general- purpose processor, or integrated with another program's functionality, but the single-purpose processor approach has the benefits of efficiency and simplicity.

One common use of a PWM is to control the average current or voltage inputto a device. For example, a DC motor rotates when power is applied, and thispower can be turned on and off by setting an input high or low. To control the speed, we can adjust the

input voltage, but this requires a conversion of our high/low digital signals to an analog signal. Fortunately, we can also adjust the speed simply by modifying the duty cycle of the motors on/off input, an approach which adjusts the average voltage. This approach works because a DC motor does not come to an immediate stop when power is turned off, but rather it coasts, much like a bicycle coasts when we stop pedaling. Increasing the duty cycle increases the motor speed, and decreasing the duty cycle decreases the speed. This duty cycle adjustment principle applies to the control other types of electric devices, such as dimmer lights.

Another use of a PWM is to encode control commands in a single signal foruse by another device. For example, we may control a radio-controlled car by sending pulses of different widths. Perhaps a 1 ms width corresponds to a turn left command, a 4 ms width to turn right, and 8 ms to forward.

LCD controller

An LCD (Liquid crystal display) is a low-cost, low-power device capable of displaying text and images. LCDs are extremely common in embedded systems, since such systems often do not have video monitor's standard for desktop systems.LCDs can be found in numerous common devices like watches, fax and copy machines, and calculators.

The basic principle of one type of LCD (reflective) works as follows. First, incoming light passes through a polarizing plate. Next, that polarized light encounters liquid crystal material. If we excite a region of this material, we cause the material's molecules to align, which in turn causes the polarized light to pass through the material. Otherwise, the light does not pass through. Finally, light that has passed through hits a mirror and reflects back, so the excited region appears to light up. Another type of LCD (absorption) works similarly, but uses a blacksurface instead of a mirror. The surface below the excited region absorbs light, thus appearing darker than the other regions.

One of the simplest LCDs is 7-segment LCD. Each of the 7 segments can be activated to display any digit character or one of several letters and symbols. Such an LCD may have 7 inputs, each corresponding to a segment, or it may have only4 inputs to represent the numbers 0 through 9. An LCD driver converts these inputsto the electrical signals necessary to excite the appropriate LCD segments.

A dot-matrix LCD consists of a matrix of dots that can display alphanumeric characters (letters and digits) as well as other symbols. A common dot-matrix LCD has 5 columns and 8

rows of dots for one character. An LCD driver converts input data into the appropriate electrical signals necessary to excite the appropriate LCD bits.

Each type of LCD may be able to display multiple characters. In addition, each character may be displayed in normal or inverted fashion. The LCD may permit a character to be blinking (cycling through normal and inverted display) or may permit display of a cursor (such as a blinking underscore) indicating the "current" character. This functionality would be difficult for us to implement using software. Thus, we use an *LCD controller* to provide us with a simple interface, perhaps 8data inputs and one enable input. To send a byte to the LCD, we provide a value to the 8 inputs and pulse the enable. This byte may be a control word, which instructs the LCD controller to initialize the LCD, clear the display, select the position of the cursor, brighten the display, and so on. Alternatively, this byte may be a data word, such as an ASCII character, instructing the LCD to display the character at the currently-selected display position.

Keypad controller

A *keypad* consists of a set of buttons that may be pressed to provide input to an embedded system. Again, keypads are extremely common in embedded systems, since such systems may lack the keyboard that comes standard with desktop systems.

A simple keypad has buttons arranged in an N-column by M-row grid. The device has N outputs, each output corresponding to a column, and another M outputs, each output corresponding to a row. When we press a button, one column output and one row output go high, uniquely identifying the pressed button. To readsuch a keypad from software, we must scan the column and row outputs.

The scanning may instead be performed by a *keypad controller* (actually, sucha device decodes rather than controls, but we'll call it a controller for consistency with the other peripherals discussed). A simple form of such a controller scans the column and row outputs of the keypad. When the controller detects a button press,it stores a code corresponding to that button into a register and sets an output high, indicating that a button has been pressed. Our software may poll this output every 100 milliseconds or so, and read the register when the output is high. Alternatively, this output can generate an interrupt on our general-purpose processor, eliminating the need for polling.

Stepper motor controller

A *stepper motor* is an electric motor that rotates a fixed number of degrees whenever we apply a "step" signal. In contrast, a regular electric motor rotates continuously whenever power is applied, coasting to a stop when power is removed. We specify a stepper motor either by the number of degrees in a single step, such as 1.8 degree, or by the number of steps required to move 360 degree, such as 200 steps. Stepper motors obviously abound in embedded systems with moving parts, such as disk drives, printers, photocopy and fax machines, robots, camcorders, VCRs, etc.

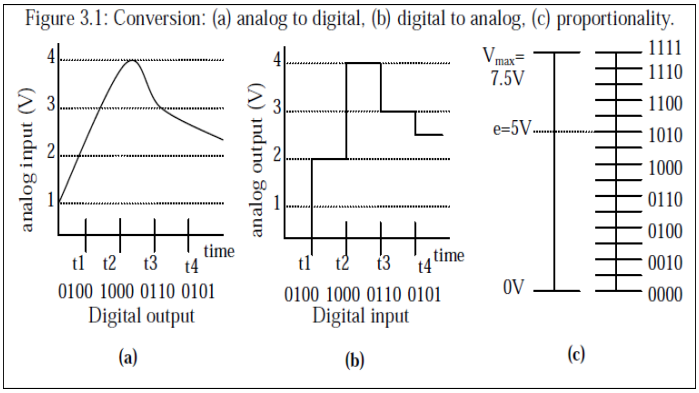
Internally, a stepper motor typically has four coils. To rotate the motor one step, we pass current through one or two of the coils; the particular coils depend on the present orientation of the motor. Thus, rotating the motor 360 degree requires applying current to the coils in a specified sequence. Applying the sequence in reverse causes reversed rotation.

In some cases, the stepper motor comes with four inputs corresponding to the four coils, and with documentation that includes a table indicating the proper input sequence. To control the motor from software, we must maintain this table in software, and write a step routine that applies high values to the inputs based on the table values that follow the previously-applied values.

In other cases, the stepper motor comes with a built-in controller (i.e., a special- purpose processor) implementing this sequence. Thus, we merely create a pulse on an input signal of the motor, causing the controller to generate the appropriate high signals to the coils that will cause the motor to rotate one step.

Analog-digital converters

An analog-to-digital converter (ADC, A/D or A2D) converts an analog signal to a digital signal, and a digital-to-analog converter (DAC, D/A or D2A) does the opposite. Such conversions are necessary because, while embedded systems deal with digital values, an embedded system’s surroundings typically involve many analog signals. Analog refers to continuously-valued signal, such as temperature or speed represented by a voltage between 0 and 100, with infinite possible values in between. "Digital" refers to discretely-valued signals, such as integers, and in computing systems, these signals are encoded in binary. By converting between analog and digital signals, we can use digital processors in an analog environment.



For example, consider the analog signal of Figure 3.1(a). The analog input voltage varies over time from 1 to 4 Volts. We sample the signal at successive timeunits, and encode the current voltage into a 4-bit binary number. Conversely, consider Figure 3.1(b). We want to generate an analog output voltage for the givenbinary numbers over time. We generate the analog signal shown.

We can compute the digital values from the analog values, and vice-versa, using the following ratio:

$$\frac{e}{V} = \frac{d}{}$$

V_{max} is the maximum voltage that the analog signal can assume, n is the number of bits available for the digital encoding, d is the present digital encoding, and e is the present analog voltage. This proportionality of the voltage and digital encoding is shown graphically in Figure 3.1(c).

In our example of Figure 3.1, suppose V_{max} is 7.5V. Then for e = 5V, we have the following ratio: 5/7.5 = d/15, resulting in d = 1010 (ten), as shown in Figure 3.1(c). The *resolution* of a DAC or ADC is defined as $V_{max}/(2^n-1)$, representing the number of volts between successive digital encodings. The above discussion assumes a minimum voltage of 0V.

Internally, DACs possess simpler designs than ADCs. A DAC has n inputs for the digital encoding d, a V_{max} analog input, and an analog output e. A fairly straightforward circuit (involving resistors and an op-amp) can be used to convert d to e.

ADCs, on the other hand, require designs that are more complex, for the following reason. Given a V_{max} analog input and an analog input *e*, how does the converter know what binary value to assign in order to satisfy the above ratio? Unlike DACs, there is no simple analog circuit to compute *d* from *e*. Instead, an ADC may itself contain a DAC also connected to V_{max}. The ADC "guesses" an encoding *d*, and then evaluates its guess by inputting *d* into the DAC, and comparing the generated analog output *e'* with the original analog input *e* (using an analog comparator). If the two sufficiently match, then the ADC has found a proper encoding. So now the question remains: how do we guess the correct encoding?

This problem is analogous to the common computer-programming problem of finding an item in a list. One approach is sequential search, or "counting-up" in analog- digital terminology. In this approach, we start with an encoding of 0, then 1, then 2, etc., until we find a match. Unfortunately, while simple, this approach in the worst case (for high voltage values) requires 2^n comparisons, so it may be quite slow.

A faster solution uses what programmers call binary search, or "successive approximation" in analog-digital terminology. We start with an encoding corresponding half of the maximum. We then compare the resulting analog value with the original; if the resulting value is greater (less) than the original, we set the new encoding to halfway between this one and the maximum (minimum). Wecontinue this process, dividing the possible encoding range in half at each step, until the compared voltages are equal. This technique requires at most *n* comparisons. However, it requires a more complex converter.

Because ADCs must guess the correct encoding, they require some time. Thus,in addition to the analog input and digital output, they include an input "start" that starts the conversion, and an output "done" to indicate that the conversion iscomplete.

Real-time clocks

Much like a digital wristwatch, a real-time clock (RTC) keeps the time and datein an embedded system. Read-time clocks are typically composed of a crystal- controlled oscillator, numerous cascaded counters, and a battery backup. The crystal-controlled oscillator generates a very consistent high-frequency digital pulse that feed the cascaded counters. The first counter, typically, counts these pulses up to the oscillator frequency, which corresponds to exactly one second. At this point,it generates a pulse that feeds the next counter. This counter counts up to 59, at which point it generates a pulse feeding the minute counter. The hour, date,

month and year counters work in similar fashion. In addition, real-time clocks adjust for leap years. The rechargeable back-up battery is used to keep the real-time clock running while the system is powered off.

From the micro-controller's point of view, the content of these counters can be set to a desired value, (this corresponds to setting the clock), and retrieved. Communication between the micro-controller and a real-time clock is accomplished

through a serial bus, such as I^2C. It should be noted that, given a timer peripheral, it is possible to implement a real-time clock in software running on a processor. In fact, many systems use this approach to maintain the time. However, the drawback ofsuch systems is that when the processor is shut down or reset, the time is lost.

Part A

1.Compare synchronous communication and iso-synchronous communication.
2.Extend a) SPI b) SCI
3.Define software timer.
4.What is meant by UART?
5.Outline the states of timer?
6.What is meant by status flag?
7.Identify the use of RTC in embedded system?
8.What is RT level combinational components?
9.What is ASIP in embedded system?
10.What is the function of pulse width modulator?
11.Identify the common types of displays used in embedded system?
12.What are the common LCD screens used in embedded systems?
13.Compare ADC and DAC in embedded system?

Part B

1.Explain in detail about custom single purpose processor with example.
2.Discuss the RT level design of embedded system
3.Elaborate in detail about ASIP design in embedded system.
4.Explain in detail about Real Time Clock in embedded system.
5.Design stepper motor control interface with embedded system

TEXT/ REFEENCE BOOKS

1.David E.Simon, "An Embedded Software Primer", Pearson Education,2001
2. Frank Vahid and Tony Gwargie, "Embedded System Design", John Wiley & Sons,2002
3. Steve Heath, "Embedded System Design", Elsevier, Second Edition,2004.

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

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT- 4 MEMORY AND INTERFACING

UNIT 4 MEMORY AND INTERFACING

Memory write ability and storage performance Memory types composing memory Advance RAM interfacing communication basic Microprocessor interfacing I/O addressing Interrupts Direct memory access Arbitration multilevel bus architecture Serial protocol Parallel protocols Wireless protocols Digital camera example.

4.1Write Ability and Storage Performance

There are two important specifications for the Memory as far as Real Time EmbeddedSystems are concerned.

1. Write Ability
2. Storage Performance

1. Write ability

It is the manner and speed that a particular memory can be written

- Ranges of write ability
  - High end
    - processor writes to memory simply and quickly e.g., RAM
  - Middle range
    - processor writes to memory, but slower e.g., FLASH, EEPROM(Electrically Erasable and Programmable Read Only Memory)
  - Lower range
    - special equipment, "programmer", must be used to write to memory e.g.,EPROM, OTP ROM (One Time Programmable Read Only Memory)
  - Low end
    - bits stored only during fabrication e.g., Mask-programmed ROM
- In-system programmable memory
  - Can be written to by a processor in the embedded system using the memory
  - Memories in high end and middle range of write ability

2. Storage permanence

It is the ability to hold the stored bits.
Range of storage permanence

- High end
  - essentially never loses bits
  - e.g., mask-programmed ROM
- Middle range
  - holds bits days, months, or years after memory's power source turned off
  - e.g., NVRAM
- Lower range
  - holds bits as long as power supplied to memory

- e.g., SRAM

– Low end
  - begins to lose bits almost immediately after written
  - e.g., DRAM

Nonvolatile memory
  - Holds bits after power is no longer supplied
  - High end and middle range of storage permanence

4.2 Memory and its types

Any embedded system's functionality consists of three aspects: processing, storage, and communication. Processing is the transformation of data, storage is the retention of data for later use, and communication is the transfer of data. Each of these aspects must be implemented. We useprocessors to implement processing, memories to implement storage, and buses to implement communication. The earlier chapters described common processor types: general-purpose processors, standard single-purpose processors, and custom single-purpose processors. This chapterdescribes memories.

A memory stores large numbers of bits. These bits exist as m words of n bits each, for a total of m*nbits. We refer to a memory as an m x n ("m-by-n") memory. $Log_2(m)$ address input signals are necessary to identify a particular word. Stated another way, if a memory has k address inputs, it canhave up to 2^k words. n signals are necessary to output (and possibly input) a selected word. To reada memory means to retrieve the word of a particular address, while to write a memory means to storea word in a particular address. Some memories can only be read from (ROM), while others can be both read from and written to (RAM). There isn't much demand for a memory that can only be written to (what purpose would such a memory serve?). Most memories have an enable input; whenthis enable is low, the address is ignored, and no data is written to or read from the memory.

Read-only memory -- ROM

ROM, or read-only memory, is a memory that can be read from, but not typically written to, duringexecution of an embedded system. Of course, there must be a mechanism for setting the bits in thememory (otherwise, of what use would the read data serve?), but we call this "programming," not writing. Such programming is usually done off-line, i.e., when the memory is not actively serving as a memory in an embedded system. We usually program a ROM before inserting it into the embedded system. Figure 4.1(b) provides a block diagram of a ROM.

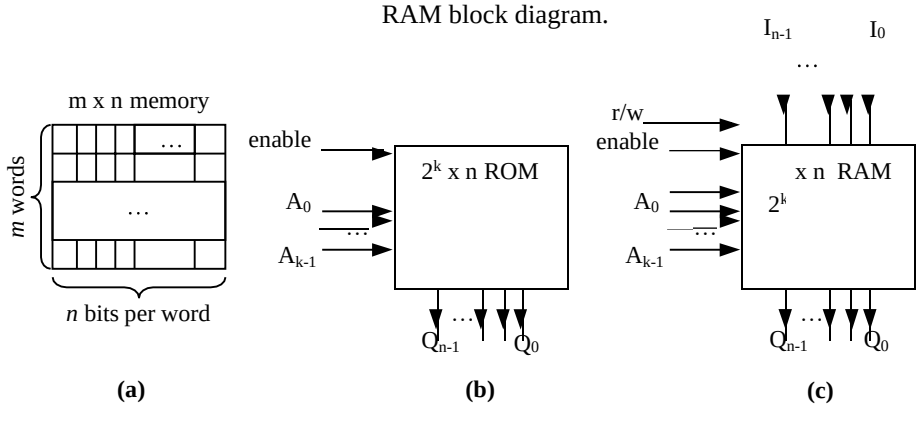
We can use ROM for various purposes. One use is to store a software program for a general-purpose processor. We may write each program instruction to one ROM word. For some processors,

we write each instruction to several ROM words. For other processors, we may pack several instructions into a single ROM word. A related use is to store constant data, like large lookup tablesof strings or numbers.

Another common use is to implement a combinational circuit. We can implement any combinational function of k variables by using a 2^kx 1 ROM, and we can implement n functions of the same k variables using a 2^kx n ROM. We simply program the ROM to implement the truth tablefor the functions, as shown in Figure 4.2.

Figure 4.3 provides a symbolic view of the internal design of an 8x4 ROM. To the right of the 3x8decoder in the figure is a grid of lines, with word lines running horizontally and data lines vertically;lines that cross without a circle in the figure are *not* connected. Thus, word lines only connect to data lines via the programmable connection lines shown. The figure shows all connection lines in place except for two connections in word 2. To see how this device acts as a read-only memory, consider an input address of "010." The decoder will thus set word 2's line to 1. Because the lines connecting this word line with data lines 2 and 0 do not exist, the ROM output will read "1010." Note that if the ROM enable input is 0, then no word is read. Also note that each data line is shownas a wired-OR, meaning that the wire itself acts to logically OR all the connections to it.

Figure 4.1: Basic memories: (a) words and bits per word, (b) ROM block diagram, (c) RAM block diagram.



How do we program the programmable connections? The answer depends on the type of ROMbeing used. In a mask-programmed ROM, the connection is made when the chip is being fabricated(by creating an appropriate set of masks). Such ROM types are typically only used

in high-volumesystems, and only after a final design has been determined.

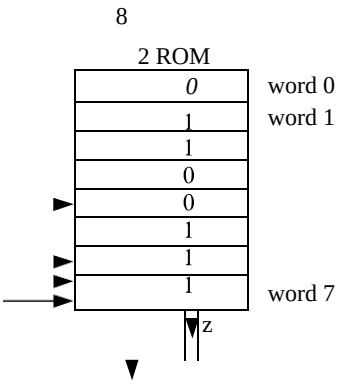
Most other systems use user-programmable ROM devices, or PROM, which can be programmed by the chip's user, well after the chip has been manufactured. These devices are bettersuited to prototyping and to low-volume applications. To program a PROM device, the user provides a file indicating the desired ROM contents. A piece of equipment called a ROM programmer (note: the programmer is a piece of equipment, not a person who writes software) thenconfigures each programmable connection according to the file. A basic PROM uses a fuse for eachprogrammable connection. The ROM programmer blows fuses by passing a large current wherevera connection should not exist. However, once a fuse is blown, the connection can never be re- established. For this reason, basic PROM is often referred to as one-time-programmable device, orOTP.

Another type of PROM is an *erasable* PROM, or *EPROM.* This device uses a MOS transistor as its programmable component. The transistor has a "floating gate," meaning its gate is not connected. An EPROM programmer injects electrons into the floating gate, using higher than normal voltage (usually 12V to 25V) that causes electrons to "tunnel" into the gate. When that highvoltage is removed, the electrons cannot escape, and hence

Figure 4.2: Implementing combinational functions with a ROM: (a) truth table, (b) ROM contents.

Truth table (ROM contents)

Inputs (address)			Outputs	
a	b	c	y	z
0	0	0	*0*	*0*
0	0	1	0	1
0	1	0	0	1
0	1	1	1	
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1



(a) **(b)**

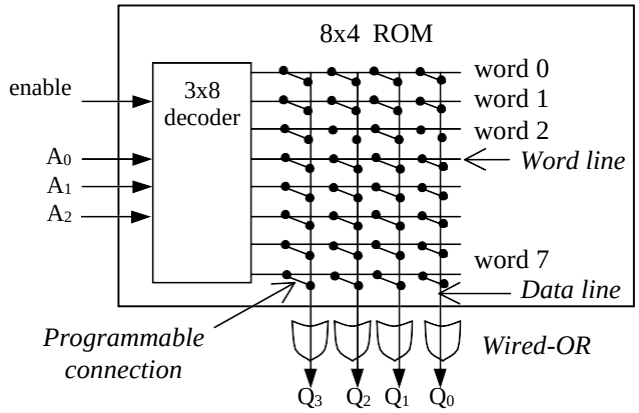
the gate has been charged and programming has occurred. Standard EPROMs are guaranteedto hold their programs for at least 10 years. To erase the program, the electrons must be excited enough to escape from the gate. Ultra-violet (UV) light is used to fulfil this role of erasing. The device must be placed under a UV eraser for a period of time, typically ranging from 5 to 30 minutes, after which the device can be programmed again. In order for the UV light to reach the chip, EPROM's come with a small quartz window in the package through which the chip can be seen. For this reason, EPROM is often referred to as a *windowed* ROM device.

Electrically-erasable PROM, or *EEPROM*, is designed to eliminate the time- consuming andsometimes impossible requirement of exposing an EPROM to UV light toerase the ROM. An EEPROM is not only programmed electronically, but is also erased electronically. These devices are typically more expensive the EPROM's, but far more convenient to use. EEPROM's are oftencalled "E-squared's" for short. *Flash* memory is a type of EEPROM in which reprogramming canbe done to certain regions of the memory, rather than the entire memory at once.

Which device should be used during development? The answer depends on cost and convenience. For example, OTP's are typically quite inexpensive, so they are quite practical unless frequent reprogramming is expected. In that case, windowed devices are typically cheaper than E- squared's. However, if one can not (or does not want to) deal with the time required for UV erasing,or if one can not move the device to a UV eraser (e.g., if it's being used in a microcontroller emulator), then E-squared's may be used.

For final implementation in a product, masked-ROM may be best for high-volume production,since its high up-front cost can be amortized over the large number of products. OTP has the advantage of low cost as well as resistance to undesired program changes caused by noise. Windowed parts if used in production should have their windows covered by a sticker to prevent undesired changes of the memory.

Figure 4.3: ROM internals



Read-write memory -- RAM

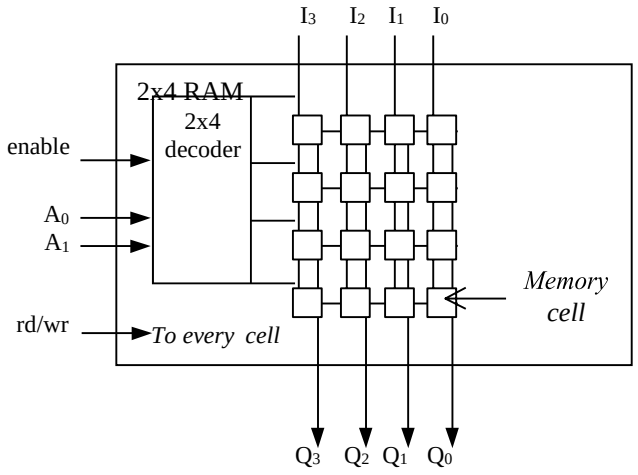
RAM, or random-access memory, is a memory that can be both read and written. In contrast to ROM, a RAM's content is not "programmed" before being inserted into an embedded system. Instead, the RAM contains no data when inserted in the embedded system; the system writes datato and then reads data from the RAM during its execution. Figure 1(c) provides a block diagram ofa RAM.

A RAM's internal structure is somewhat more complex than a ROM's, as shown in Figure 4.4, which illustrates a 4x4 RAM (note: RAMs typically have thousands of words, not just 4 as inthe figure). Each word consists of a number of memory cells, each storing one bit. In the figure, each input data connects to every cell in its column. Likewise, each output data line connects to every cell in its column, with the output of a memory cell being OR'ed with the output data line from above. Each word enable line from the decoder connects to every cell its its row. The read/write input (rd/wr) is assumed to be connected to every cell. The memory cell must possess logic such that it stores the input data bit when rd/wr indicates write and the row is enabled, and such that it outputs this bitwhen rd/wr indicates read and the row is enabled.

There are two basic types of RAM, static and dynamic. Static RAM is faster but bigger thandynamic RAM. *Static* RAM, or *SRAM*, uses a memory cell consisting of a flip-flop to store a bit.Each bit thus requires about 6 transistors. This RAM type is called static because it will hold itsdata as long as power is supplied, in contrast to dynamic RAM. Static RAM is typically used forhigh-performance parts of a system (e.g., cache).

Dynamic RAM, or *DRAM*, uses a memory cell consisting of a MOS transistor and capacitor tostore a bit. Each bit thus requires only 1 transistor, resulting in more compact memory than SRAM. However, the charge stored in the capacitor leaks gradually, leading to discharge and eventually to loss of data. To prevent loss of data, each cell must regularly have its charge "refreshed." A typical DRAM cell minimum refresh rate is once

Figure 4.4: RAM internals



every 15.625 microseconds. Because of the way DRAMs are designed, reading a DRAM word refreshes that word's cells. In particular, accessing a DRAM word result in the word's data being stored in a buffer and then being written back to the word's cells. DRAMs tend to be

slower to accessthan SRAMs.

Many RAM variations exist. Pseudo-Static RAMs, or PSRAMs, are DRAMs with a refresh controllerbuilt-in. Thus, since the RAM user need not worry about refreshing, the device appears to behave much like an SRAM. However, in contrast to true SRAM, a PSRAM may be busy refreshing itself when accessed, which could slow access time and add some system complexity. Nevertheless, PSRAM is a popular low-cost alternative to SRAM in many embedded systems.

Non-volatile RAM, or *NVRAM*, is another RAM variation. Non-volatile storage is storage that can hold its data even after power is no longer being supplied. Note that all forms of ROM are non- volatile, while normal forms of RAM (static or dynamic) are volatile. One type of NVRAM containsa static RAM along with its own permanently connected battery. A second type contains a static RAMand its own (perhaps flash) EEPROM. This type stores RAM data into the EEPROM just before power is turned off (or whenever instructed to store the data), and reloads that data from EEPROM into RAM after power is turned back on. NVRAM is very popular in embedded systems. For example,a digital camera must digitize, store and compress an image in a fraction of a second when the camera's button is pressed, requiring writes to a fast RAM (as opposed to programming of a slower EEPROM). But it also must store that image so that the image is saved even when the camera's power is shut off, requiring EEPROM. Using NVRAM accomplishes both these goals, since the datais originally and quickly stored in RAM, and then later copied to EEPROM, which may even take a few seconds.

Note that the distinction we made between ROM and RAM, namely that ROM is programmed before insertion into an embedded system while RAM is written by the embedded system, does not hold in every case. As in the digital camera example above, EEPROM may be programmed by the embedded system during execution, though such programming is typically infrequent due to its time-consuming nature.

A common is question is: where does the term "random-access" come from in random-access memory? RAM should really be called read-write memory, to contrast it from read-only memory. However, when RAM was first introduced, it was in stark contrast to the then common sequentially-accessed memory media, like magnetic tapes or drums. These media required that the particular location to be accessed be positioned under an access device (e.g., a head). To access another locationnot immediately adjacent to the current location on the media, one would have sequence through a number of other locations, e.g., for a tape, one

would have to rewind or fast-forward the tape. In contrast, with RAM, any "random" memory location could be accessed in the same amount of time as any other location, regardless of the previously read location. This random-access feature was thekey distinguishing feature of this memory type at the time of its introduction, and the name has stuckeven today.

4.3 Composing memories

An embedded system designer is often faced with the situation of needing a particular-sized memory (ROM or RAM), but having readily available memories of a different size. For example, thedesigner may need a 2k x 8 ROM, but may have 4k x 16 ROMs readily available. Alternatively, the designer may need a 4k x 16 ROM, but may have 2k x 8 ROMs available for use.

The case where the available memory is larger than needed is easy to deal with. We simply use the needed lower words in the memory, thus ignoring unneeded higher words and their high-orderaddress bits, and we use the lower data input/output lines, thus ignoring unneeded higher data lines. (Of course, we could use the higher data lines and ignore the lower lines instead).

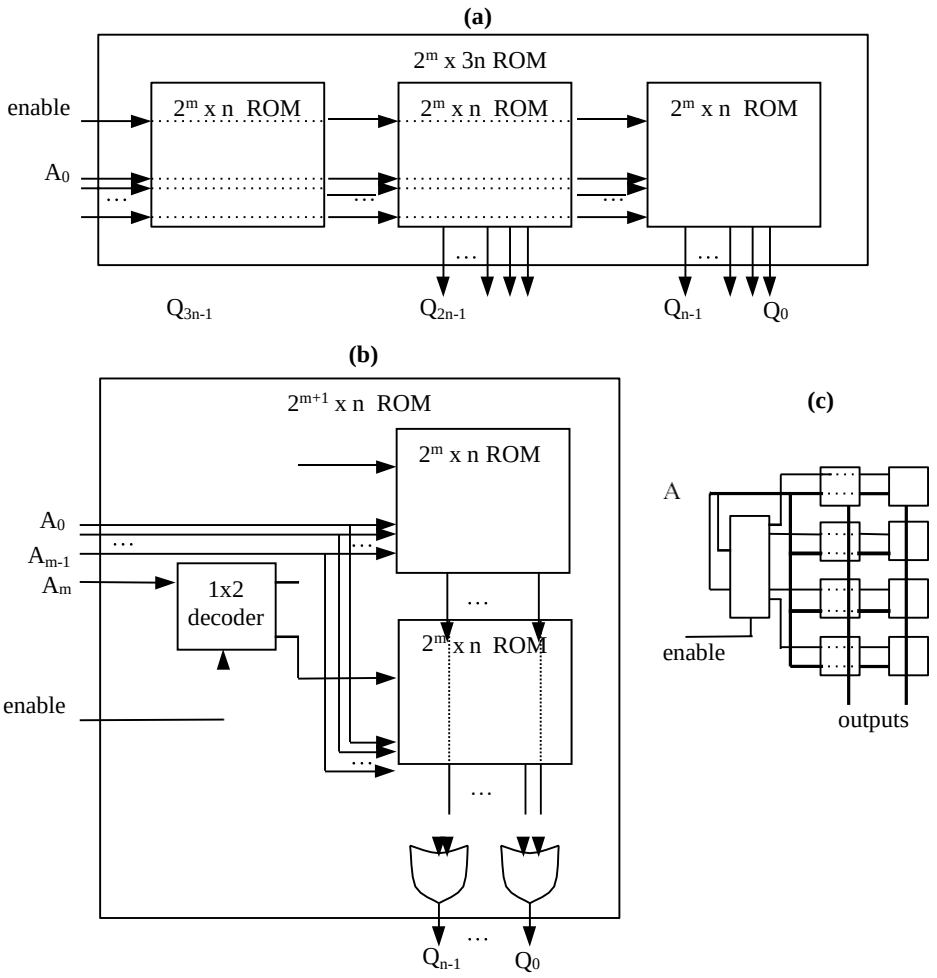
The case where the available memory is smaller than needed requires more design effort. In thiscase, we must compose several smaller memories to behave as the larger memory we need. Supposethe available memories have the correct number of words, but each word is not wide enough. In thiscase, we can simply connect the available memories side-by-side. For example, Figure 4.5(a) illustrates the situation of needing a ROM three-times wider than that available. We connect three ROMs side-by-side, sharing the same address and enable lines among them, and concatenating the data lines to form the desired word width.

Suppose instead that the available memories have the correct word width, but not enough words.In this case, we can connect the available memories top-to-bottom. For example, Figure 4.5(b) illustrates the situation of needing a ROM with twice as many words, and hence needing one extra address line, than that available. We connect the ROMs top-to-bottom, OR'ing the corresponding data lines of each. We use the extra high-order address line to select the higher or lower ROM (usinga 1x2 decoder), and the remaining address lines to offset into the selected ROM. Since only one ROM will ever be enabled at a time, the OR'ing of the data lines never actually involves more than one 1.

If we instead needed four times as many words, and hence two extra address lines, we would insteaduse four ROMs. A 2x4 decoder having the two high-order address lines as input would select whichof the four ROMs to access.

Finally, suppose the available memories have a smaller word with as well as fewer words thannecessary. We then combine the above two techniques, first creating the number of columns of memories necessary to achieve the needed word width, and then creating the number of rows of memories necessary, along with a decoder, to achieve the needed number of words. The approach is illustrated in Figure 4.5(c).

Figure 4.5: Composing memories into larger ones.



4.4 Interfacing

Buses implement communication among processors or among processors and memories. Communication is the transfer of data among those components. For example, a general-purpose processor reading or writing a memory is a common form of communication. A general-purpose processor reading or writing a peripheral's register is another common form.

A bus consists of wires connecting two or more processors or memories. Figure 6.1(a) shows the wires of a simple bus connecting a processor with a memory. Note that each wire may be uni-directional, as are *rd/wr*, *enable*, and *addr*, or bi-directional, as is *data*. Also note that a

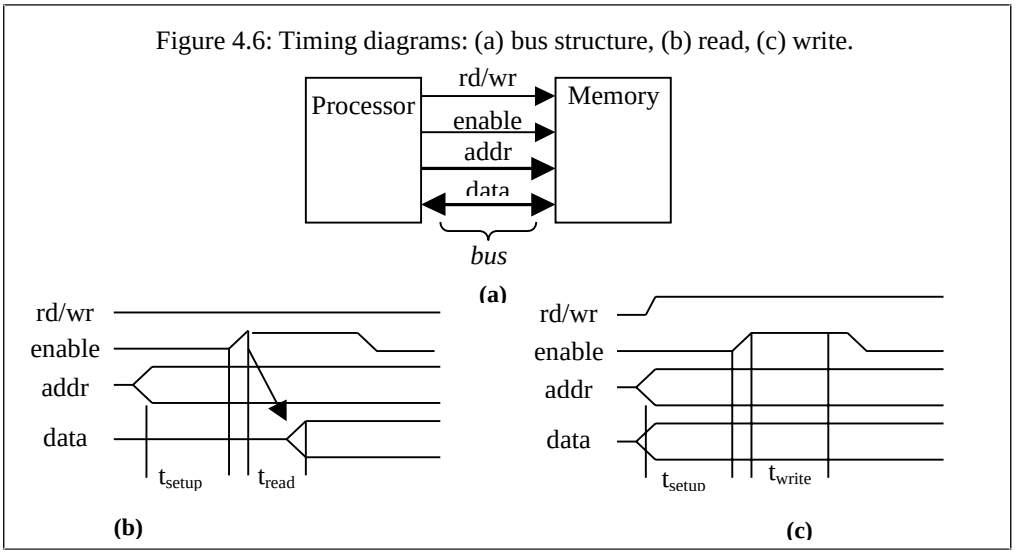
set of wires with the same function is typically drawn as a thick line (or a line with a small angled line drawn through it). *addr* and *data* each represent a set of wires; the *addr* wires transmit an address, while the *data* wires transmit data. The bus connects to "pins" of a processor (or memory). A pin is the actual conducting device (i.e., metal) on the periphery of a processor through which a signal is input to or output from the processor. When a processor is packaged as its own IC, there are actual pins extending from the package, designed to be plugged into a socket on a printed-circuit board. Today, however, a processor commonly co-exists on a single IC with other processors and memories.Such a processor does not have any actual pins on its periphery, but rather "pads" of metal in the IC.In fact, even for a processor packaged in its own IC, alternative packaging-techniques may use something other than pins for connections, such as small metallic balls. For consistency, though, weshall use the term pin in this chapter regardless of the packaging situation.

A bus must have an associated *protocol* describing the rules for transferring data over those wires.We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

Interfacing with a general-purpose processor is extremely common. We describe three issues relating to such interfacing: addressing, interrupts, and direct memory access. When multiple processors attempt to access a single bus or memory simultaneously, resource contention exists.

Timing diagrams

Figure 4.6: Timing diagrams: (a) bus structure, (b) read, (c) write.



The most common method for describing a hardware protocol is a timing diagram. Consider the example processor-memory bus of Figure 4.6 (a). Figure 4.6 (b) uses a timing diagramto describe the protocol for reading the memory over the bus. In the diagram, time

proceeds to the right along the x-axis. The diagram shows that the processor must set the *rd/wr* line low for a readto occur. The diagram also shows, using two vertical lines, that the processor must place the address on *addr* for at least t_{setup} time before setting the *enable* line high. The diagram shows that the high *enable* line triggers the memory to put data on the *data* wires after a time t_{read}. Note that a timing diagram represents control lines, like *rd/wr* and *enable*, as either being high or low, whileit represents data lines, like *addr* and *data*, as being either invalid (a single horizontal line) or valid(two horizontal lines); the value of data lines is not normally relevant when describing a protocol.

In the above protocol, the control line *enable* is active high, meaning that a 1 on the enable line triggers the data transfer. In many protocols, control lines are instead active low, meaning that a 0on the line triggers the transfer. Such a control line is typically written with a bar above it, a singlequote after it (e.g., *enable'*), or an underscore l after it(e.g., *enable_l*). To be general, we will use the term "assert" to mean setting a control lineto its active value (i.e., to 1 for an active high line, to 0 for an active low line), and the term "deassert" to mean setting the control line to its inactive value.

Hardware protocol basics

Concepts

The protocol described above was a simple one. Hardware protocols can be much more complex. However, we can understand them better by defining some basic protocol

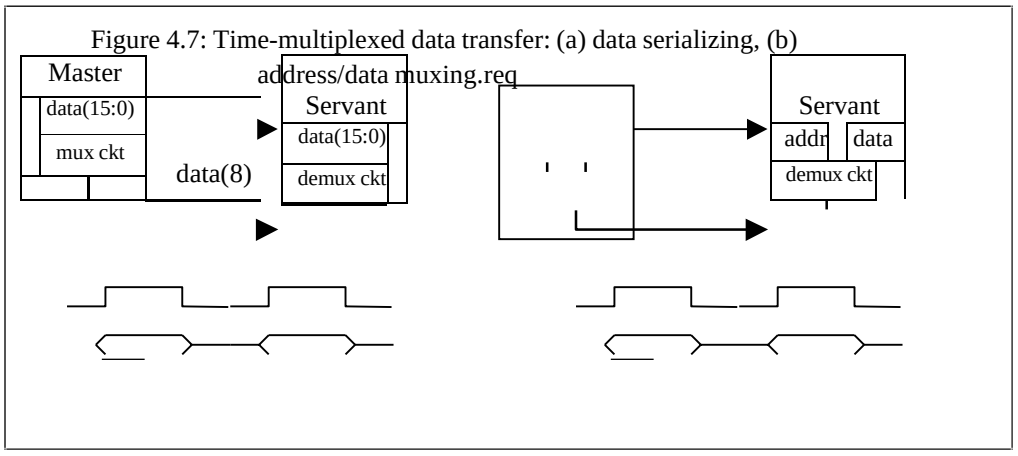


Figure 4.7: Time-multiplexed data transfer: (a) data serializing, (b) address/data muxing.req

An *actor* is a processor or memory involved in the data transfer. A protocol typically involvestwo actors: a master and a servant. A *master* initiates the data transfer. A *servant* (usually called a slave) responds to the initiation request. In the example of Figure 4.7, the processor is the master and the memory is the servant, i.e., the memory cannot initiate a data transfer. The servant could also be another processor. Masters are usually general-purpose

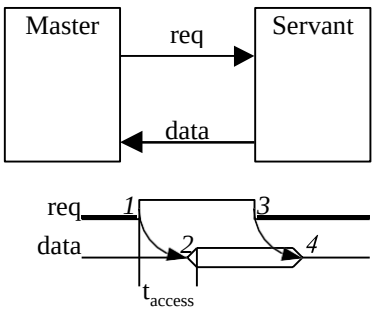
processors, while servants are usually peripherals and memories.

Data direction denotes the direction that the transferred data moves between the actors. We indicate this direction by denoting each actor as either receiving or sending data. Note that actor types are independent of the direction of the data transfer. In particular, a master may either be the receiver of data, as in Figure 4.6(b), or the sender of data, as shown in Figure 4.6(c).

Addresses represent a special type of data used to indicate where regular data should go to or come from. A protocol often includes both an address and regular data, as did the memory access protocol in Figure 4.6, where the address specified where the data should be read from or written toin the memory. An address is also necessary when a general- purpose processor communicates withmultiple peripherals over a single bus; the address not only specifies a particular peripheral, but alsomay specify a particular register within that peripheral.

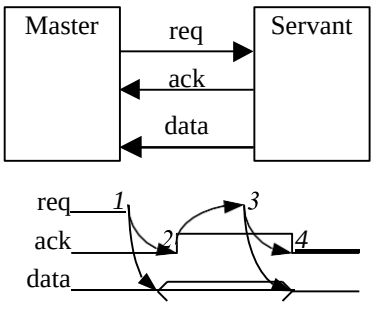
Another protocol concept is *time multiplexing*. To multiplex means to share a single set of wiresfor multiple pieces of data. In time multiplexing, the multiple pieces of data are sent one at a time over the shared wires. For example, Figure 4.7(a) shows a master sending 16 bits of data over an 8-bit bus using a strobe protocol and time-multiplexed data. The master first sends the high-order byte,then the low-order byte. The servant must receive the bytes and then demultiplex the data. This serializing of data can be done to any extent, even down to a 1-bit bus, in order to reduce the numberof wires. As another example, Figure 4.7(b) shows a master sending both an address and data to a servant (probably a memory). In this case, rather than using separate sets of lines for

Figure 4.8: Two protocol control methods: (a) strobe, (b) handshake.



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

(a)



1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

(b)

address and data, as was done in Figure 4.6, we can time multiplex the address and data over a shared set of lines *addr/data*.

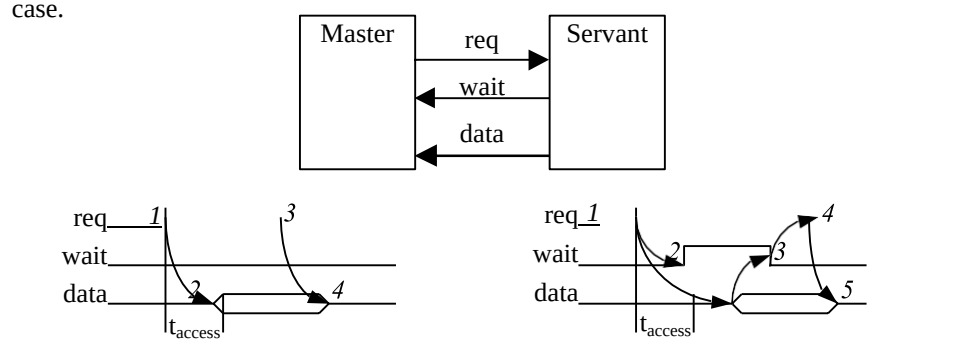
Control methods are schemes for initiating and ending the transfer. Two of the most common methods are strobe and handshake. In a *strobe* protocol, the master uses onecontrol line, often called the *request* line, to initiate the data transfer, and the transfer is considered to be complete after some fixed time interval after the initiation. For example, Figure 4.8(a) shows a strobe protocol with a master wanting to receive data from a servant.The master first asserts the request line to initiate a transfer. The servant then has time t_{access}

,to put the data on the data bus. After this time, the master reads the data bus, believing the data to be valid. The master than deasserts the request line, so that the servant can stop putting the data on the data bus, and both actors are then ready for the next transfer. An analogy is a demanding boss who tells an employee "I want that report (the data) on my desk (the data bus) in one hour (t_{access})," and merely expects the report to be on the desk in one hour.

The second common control method is a *handshake* protocol, in which the master usesa request line to initiate the transfer, and the servant uses an *acknowledge* line to inform the master when the data is ready. For example, Figure 4.8 (b) shows a handshake protocol witha receiving master. The master first asserts the request line to initiate the transfer. The servant takes however much time is necessary to put the data on the data bus, and then asserts the acknowledge line to inform the master that the data is valid. The master reads thedata bus and then deasserts the request line so that the servant can stop putting data on the data bus. The servant deasserts the acknowledge line, and both actors are then ready for thenext transfer. In our boss-employee analogy, a handshake protocol corresponds to a more tolerant boss who tells an employee "I want that report on my desk soon; let me know when

it's ready." A handshake protocol can adjust to a servant (or

Figure 4.9: A strobe/handshake compromise: (a) fast-response case, (b) slow-response case.



1. Master asserts req to receive data
***2.** Servant puts data on bus* **within time t_{access}** *(wait line is unused)*
3. Master receives data and deasserts request
4. Servant ready for next request

(a)

1. Master asserts req to receive data
2.** Servant can't put data within* **t_{access}, *asserts wait
3. Servant finally puts data and ***deasserts wait***
ack
4. Master receives data and deasserts request
5. Servant ready for next request

(b)

servants) with varying response times, unlike a strobe protocol. However, when responsetime is known, a handshake protocol may be slower than a strobe protocol, since it requires the master to detect the acknowledgement before getting the data, possibly requiring an extra clockcycle if the master is synchronizing the bus control signals. A handshake also requires an extra line for acknowledge.

To achieve both the speed of a strobe protocol and the varying response time tolerance of a handshake protocol, a compromise protocol is often used, as illustrated in Figure

4.9. In the case, when the servant can put the data on the bus within time t_{access}, the protocol is identical to a strobe protocol, as shown in Figure 4.9(a). However, if the servant cannot put the data on the bus in time, it instead tells the master to wait longer, by asserting a line we've labeled*wait*. When the servant has finally put the data on the bus, it deasserts the wait line, thus informing the master that the data is ready. The master receives the data and deasserts the requestline. Thus, the handshake only occurs if it is necessary. In our boss-employee analogy, the bosstells the employee "I want that report on my desk in an hour; if you can't finish by then, let me know that and then let me know when it's ready."

Example: A simple bus protocol Interfacing with a general-purpose processor

Perhaps the most common communication situation in embedded systems is the input and output (I/O) of data to and from a general-purpose processor, as it communicates with its peripherals and memories. I/O is relative to the processor: input means data comes into the processor, while output means data goes out of the processor.

I/O addressing

A microprocessor may have tens or hundreds of pins, many of which are control pins, such as a pin for clock input and another input pin for resetting the microprocessor. Many of the other pins are used to communicate data to and from the microprocessor, which we call processor I/O. There are two common methods for using pins to support I/O: ports, and system buses.

A *port* is a set of pins that can be read and written just like any register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named P0. A C-language programmer may write to P0 using an instruction like: P0 = 255, which would set all 8 pins to 1's. In this case, the C compiler manual would have defined P0 as a special variable that would automatically be mapped to the register P0 during compilation. Conversely, the programmer might read the value of a port P1 being

written by some other device, by saying something like a=P1. In some microprocessors, each bit of a port can be configured as input or output by writing to a configuration register for the port. For example, P0 might have an associated configuration register called CP0. To set the high-order four bits to input and the low- order four bits to output, we might say: CP0 = 15. This writes 00001111 to the CP0 register, where a 0 means input and a 1 means output. Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say: x = P0.2, giving x the value of the number 2 connection of port P0. Port- based I/O is also called *parallel I/O*.

In contrast to a port, a *system bus* is a set of pins consisting of address pins, data pins, and control pins (for strobing or handshaking). The microprocessor uses the bus to access memory as well as peripherals. We normally consider the access to the peripherals as I/O, but don't normally consider the access to memory as I/O, since the memory is considered more as a part of the microprocessor. A microprocessor may use one of two methods for communication over a system bus: standard I/O or memory-mapped I/O.

In *memory-mapped I/O*, peripherals occupy specific addresses in the existing address space. For example, consider a bus with a 16-bit address. The lower 32K addresses may correspondto memory addresses, while the upper 32K may correspond to I/O addresses.

In *standard I/O* (also known as I/O-mapped I/O), the bus includes an additional pin, which we label M/IO, to indicate whether the access is to memory or to a peripheral (i.e., an I/O device).For example, when M/IO is 0, the address on the address bus corresponds to a memory address. When M/IO is 1, the address corresponds to a peripheral.

An advantage of memory-mapped I/O is that the microprocessor need not include special instructions for communicating with peripherals. The microprocessor's assembly instructions involving memory, such as MOV or ADD, will also work for peripherals. For example, a microprocessor may have an *ADD A, B* instruction that adds the data at address *B* to the data at address *A* and stores the result in *A*. *A* and *B* may correspond to memory locations, or registers in peripherals. In contrast, if the microprocessor uses standard I/O, the microprocessor requires special instructions for reading and writing peripherals. These instructions are often called *IN* and *OUT*. Thus, to perform the same addition of locations *A* and *B* corresponding to peripherals,the following instructions would be necessary:

IN R0, AIN R1, BADD R0,
R1 OUT A, R0

4.5 Interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine** (ISR) or **Interrupt Handler**. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

Hardware Interrupt

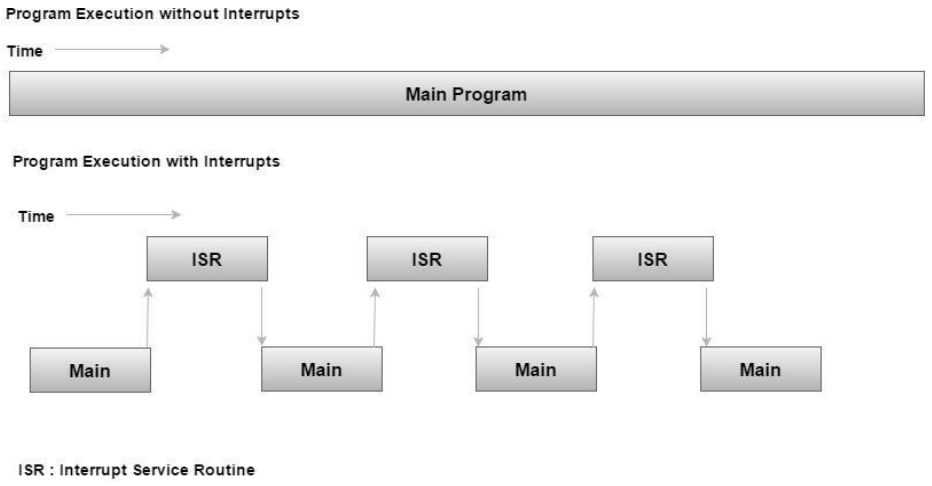
A hardware interrupt is an electronic alerting signal sent to the processor from an external device,like a disk controller or an external peripheral. For example, when we press a key on the keyboardor move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

Software Interrupt

A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if theprocessor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message.Software interrupt instructions work similar to subroutine calls.

Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or **interrupt handler**. Whenan interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, thereis a fixed location in memory that holds the address of its interrupt service routine, ISR. The tableof memory locations set aside to hold the addresses of ISRs is called as



the Interrupt Vector Table.

Interrupt Vector Table

There are six interrupts including RESET in 8051.

Interrupts	ROM Location (Hex)	Pin
Interrupts	ROM Location (HEX)	
Serial COM (RI and TI)	0023	
Timer 1 interrupts(TF1)	001B	
External HW interrupt 1 (INT1)	0013	P3.3 (13)
External HW interrupt 0 (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
Reset	0000	9

- When the reset pin is activated, the 8051 jumps to the address location 0000. This is power-up reset.
- Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations are 000BH and 001BH respectively in the interrupt vector table.

- Two interrupts are set aside for hardware external interrupts. Pin no. 12 and Pin no. 13 in Port 3 are for the external hardware interrupts INT0 and INT1, respectively. Memory locations are 0003H and 0013H respectively in the interrupt vector table.
- Serial communication has a single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps.

- The microcontroller closes the currently executing instruction and saves the address of thenext instruction (PC) on the stack.
- It also saves the current status of all the interrupts internally (i.e., not on the stack).
- It jumps to the memory location of the interrupt vector table that holds the address of the interrupts service routine.
- The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).
- Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by poppingthe top bytes of the stack into the PC. Then, it start to execute from that address.

Enabling and Disabling an Interrupt

Upon Reset, all the interrupts are disabled even if they are activated. The interrupts must be enabledusing software in order for the microcontroller to respond to those interrupts.

IE (interrupt enable) register is responsible for enabling and disabling the interrupt. IE is a bit addressable register.

Interrupt Enable Register

EA	-	ET2	ES	ET1	EX1	ET0	EX0

- **EA** − Global enable/disable.
- **-** − Undefined.
- **ET2** − Enable Timer 2 interrupt.
- **ES** − Enable Serial port interrupt.
- **ET1** − Enable Timer 1 interrupt.
- **EX1** − Enable External 1 interrupt.
- **ET0** − Enable Timer 0 interrupt.
- **EX0** − Enable External 0 interrupt.

To enable an interrupt, we take the following steps

Bit D7 of the IE register (EA) must be high to allow the rest of register to take effect.

- If EA = 1, interrupts will be enabled and will be responded to, if their corresponding bits inIE are high. If EA = 0, no interrupts will respond, even if their associated pins in the IE register are high.

-	-	-	-	PT1	PX1	PT0	PX0

-	IP.7	Not Implemented.
-	IP.6	Not Implemented.
-	IP.5	Not Implemented.
-	IP.4	Not Implemented.
PT1	IP.3	Defines the Timer 1 interrupt priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level.
PT0	IP.1	Defines the Timer 0 interrupt priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level.

Interrupt Priority in 8051

We can alter the interrupt priority by assigning the higher priority to any one of the interrupts.

Thisis accomplished by programming a register called **IP** (interrupt priority).
The following figure shows the bits of IP register. Upon reset, the IP register contains all 0's. Togive a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

Interrupt inside Interrupt

What happens if the 8051 is executing an ISR that belongs to an interrupt and another one gets active? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is knownas **interrupt inside interrupt**. In 8051, a low-priority interrupt can be interrupted by a high- priority interrupt, but not by any another low-priority interrupt.

Triggering an Interrupt by Software

There are times when we need to test an ISR by way of simulation. This can be done with the simple instructions to set the interrupt high and thereby cause the 8051 to jump to the interrupt vector table. For example, set the IE bit as 1 for timer 1. An instruction **SETB TF1** will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table.

4.6 Direct memory access (DMA)

Direct memory access (DMA) is a means of having a peripheral device control a processor's memory bus directly. DMA permits the peripheral, such as a UART, to transfer data directly to or from memory without having each byte (or word) handled by the processor. Thus DMA enables more efficient use of interrupts, increases data throughput, and potentially reduces hardware costs by eliminating the need for peripheral-specific FIFO buffers.

Dealing direct

In a typical DMA transfer, some event (such as an incoming data-available signal from a UART) notifies a separate device called the *DMA controller* that data needs to be transferred to memory. The DMA controller then asserts a *DMA request* signal to the CPU, asking its permission to use thebus. The CPU completes its current bus activity, stops driving the bus, and returns a *DMAacknowledge* signal to the DMA controller. The DMA controller then reads and writes one or morememory bytes, driving the address, data, and control signals as if it were itself the CPU. (The CPU'saddress, data, and control outputs are tri stated while the DMA controller has control of the bus.) When the transfer is complete, the DMA controller stops driving the bus and de-asserts the DMA request signal. The CPU can then remove its DMA acknowledge signal and resume control of the bus.
Each DMA cycle will typically result in at least two bus cycles: either a peripheral read

followed by a memory write or a memory read followed by a peripheral write, depending on the transfer baseaddresses. The DMA controller itself does no processing on this data. It just transfers the bytes as instructed in its configuration registers.
It's possible to do a flyby transfer that performs the read and write in a single bus cycle. However, though supported on the ISA bus and its embedded cousin PC/104, flyby transfers are not typical.

Processors that support DMA provide one or more input signals that the bus requester can assert togain control of the bus and one or more output signals that the processor asserts to indicate it has relinquished the bus. A typical output signal might be named HLDA (short for HoLD Acknowledge).

When designing with DMA, address buffers must be disabled during DMA so the bus requester candrive them without bus contention. To avoid bus contention, the bus buffer used by the DMA devicemust not drive the address bus until after HLDA goes active to indicate that the CPU has stopped driving the bus signals, and it must stop driving the bus before the CPU drives HLDA inactive. Thesystem design may also need pullup resistors or terminators on control signals (such as read and write strobes) so the control signals don't float to the active state during the brief period when neitherthe processor nor the DMA controller is driving them.
DMA controllers require initialization by software. Typical setup parameters include the base address of the source area, the base address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete.

It's typically possible to have the DMA controller automatically increment one or both addresses after each byte (word) transfer, so that the next transfer will be from the next memory location. Transfers between peripherals and memory often require that the peripheral address not be incremented after each transfer. When the address is not incremented, each data byte will be transferred to or from the same memory location.

DMA or burst

DMA operations can be performed in either burst or single-cycle mode. Some DMA controllerssupport both. In burst mode, the DMA controller keeps control of the bus until all the data buffered by the requesting device has been transferred to memory (or when the output device buffer is full, if writing to a peripheral).

In single-cycle mode, the DMA controller gives up the bus after each transfer. This minimizes the amount of time that the DMA controller keeps the processor off of the memory bus, but it requiresthat the bus request/acknowledge sequence be performed for every transfer. This overhead can result in a drop in overall system throughput if a lot of data needs to be transferred.

In most designs, you would use single cycle mode if your system cannot tolerate more than a few cycles of added interrupt latency. Likewise, if the peripheral devices can buffer very large amountsof data, causing the DMA controller to tie up the bus for an excessive amount of time, single-cyclemode is preferable.

Note that some DMA controllers have larger address registers than length registers. For instance, aDMA controller with a 32-bit address register and a 16-bit length register can access a 4GB memoryspace, but can only transfer 64KB per block. If your application requires DMA transfers of larger amounts of data, software intervention is required after each block.

Get on the bus

The simplest way to use DMA is to select a processor with an internal DMA controller. This eliminates the need for external bus buffers and ensures that the timing is handled correctly. Also, an internal DMA controller can transfer data to on-chip memory and peripherals, which is something that an external DMA controller cannot do. Because the handshake is handled on-chip, the overhead of entering and exiting DMA mode is often much faster than when an external controller is used.

If an external DMA controller or processor is used, be sure that the hardware handles the transitionbetween transfers correctly. To avoid the problem of bus contention, ensure that bus requests are inhibited if the bus is not free. This prevents the DMA controller from requesting the bus before theprocessor has reacquired it after a transfer. DMA is not as mysterious as it sometimes seems. DMAtransfers can provide real advantages when the system is properly designed.

4.7 Arbitration

For example, multiple peripherals might share a single microprocessor that services their interrupt requests. As another example, multiple peripherals might share a single DMA controller that services their DMA requests. In such situations, two or more peripherals may request service simultaneously. We therefore must have some method to arbitrate among these contending requests,i.e., to decide which one of the contending peripherals gets service, and thus which peripherals need to wait.

Multi-level bus architectures

A microprocessor-based embedded system will have numerous types of communications that must take place, varying in their frequencies and speed requirements. The most frequent and high- speed communications will likely be between the microprocessor and its memories. Less frequent communications, requiring less speed, will be between the microprocessor and its peripherals, like a UART. We could try to implement a single high-speed bus for all the communications, but this approach has several disadvantages. First, it requires each peripheral to have a high-speed bus interface. Since a peripheral may not need such high-speed communication, having such an interfacemay result in extra gates, power consumption and cost. Second, since a high- speed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, havingtoo many peripherals on the bus may result in a slower bus.

Therefore, we often design systems with two levels of buses: a high-speed processorlocal busand a lower-speed peripheral bus, as illustrated in Figure 4.10. The processor local bus typically connects the microprocessor, cache, memory controllers, certain high- speed co-processors, and ishighly processor specific. It is usually wide, as wide as a memory word.

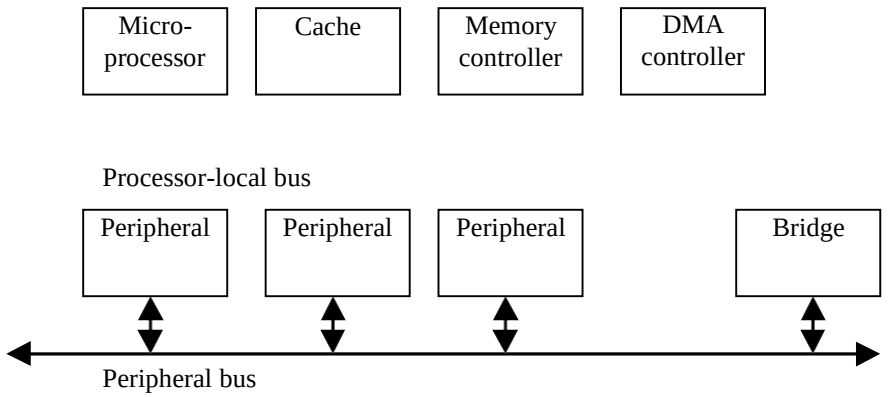
The peripheral bus connects those processors that do not have fast processor local bus access as a top priority, but rather emphasize portability, low power, or low gate count. The peripheral busis typically an industry standard bus, such as ISA or PCI, thus supporting portability of the peripherals. It is often narrower and/or slower than a processor local bus, thus requiring fewer gatesand less power for interfacing.

A *bridge* connects the two buses. A bridge is a single-purpose processor that converts communication on one bus to communication on another bus. For example, the microprocessor maygenerate a read on the processor local bus with an address corresponding to a peripheral. The bridgedetects that the address corresponds to a peripheral, and thus it then generates a read on the peripheral bus. After receiving the data, the bridge sends that data to the microprocessor. The microprocessor thus need not even know that a bridge exists -- it receives the data, albeit a few cycles later, as if the peripheral were on the processor local bus.

A three-level bus hierarchy is also possible, as proposed by the VSI Alliance. The first level is the processor local bus, the second level a system bus, and the third level a peripheral bus. The system bus would be a high-speed bus, but would offload much of the traffic from

the processor local bus. It may be beneficial in complex systems with numerous co-processors.

Figure 4.10: A two-level bus architecture.



4.8 Communication Protocols

4.8.1 Serial Protocols

Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called *communication protocols*. Luckily for us, there are only a few communication protocols we need to know when building most electronics projects. The basics of the three most common protocols: SPI, I2C and UART.

SPI, I2C, and UART are quite a bit slower than protocols like USB, Ethernet, Bluetooth,and Wi-Fi, but they're a lot simpler and use less hardware and system resources. SPI, I2C, andUART are ideal for communication between microcontrollers and between microcontrollers and sensors where large amounts of high speed data don't need to be transferred.

Data Communication Types:

(1) PARALLEL

(2) SERIAL: (I) ASYNCHRONOUS (II) SYNCHRONOUSParallel Communication:

- In parallel communication, all the bits of data are transmitted simultaneously onseparate communication lines.
- Used for shorter distance.
- In order to transmit n bit, n wires or lines are used.
- More costly.
- Faster than serial transmission.
- Data can be transmitted in less time.

Example: printers and hard disk

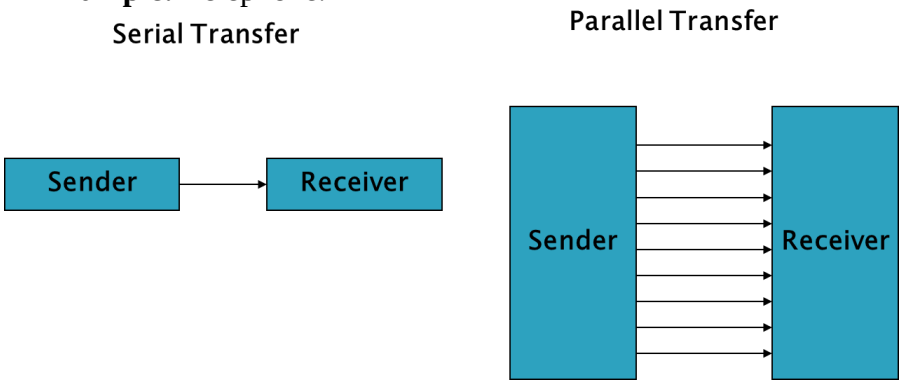
Serial Communication Basics:

- In serial communication the data bits are transmitted serially one by one

  i.e. bit by bit on single communication line
- It requires only one communication line rather than n lines to transmitdata from sender to receiver.

- Thus all the bits of data are transmitted on single lines in serial fashion.
- Less costly.
- Long distance transmission.

Example: Telephone.



Serial communication uses two methods:

- Asynchronous.
- Synchronous.Asynchronous:
- Transfers single byte at a time.
- No need of clock signal

❖ Example: UART (universal asynchronous receiver transmitter)Synchronous:

- Transfers a block of data (characters) at a time.
- Requires clock signal

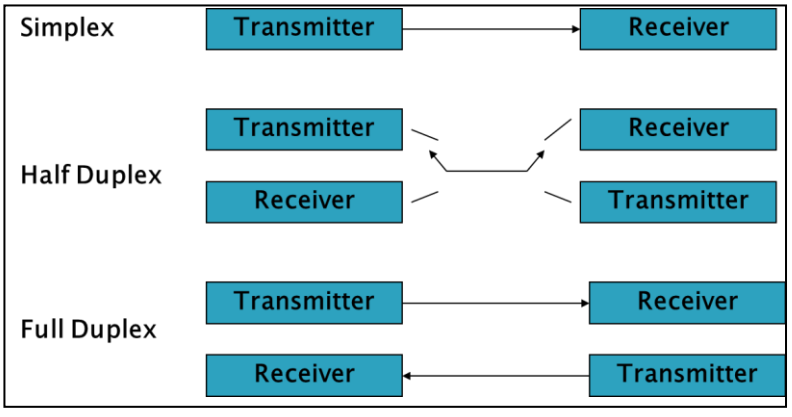
❖ Example: SPI (serial peripheral interface),
I2C (inter integrated circuit).

Data Transmission: In data transmission if the data can be transmitted and received, it is a duplex transmission.

Simplex: Data is transmitted in only one direction i.e. from TX to RX only one TXand one RX only

Half duplex: Data is transmitted in two directions but only one way at a time i.e. twoTX's, two RX's and one line

Full duplex: Data is transmitted both ways at the same time i.e. two TX's, two RX'sand two lines



A **Protocol** is a set of rules agreed by both the sender and receiver on

- How the data is packed
- How many bits constitute a character
- When the data begins and ends

Table: Various Serial Communication Protocols

Serial Protocol	**Synchronous /Asynchronous**	**Type**	**Duplex**	**Data transfer rate (kbps)**
UART	Asynchronous	peer-to-peer	Full-duplex	20
I2C	Synchronous	multi-master	Half-duplex	3400
SPI	Synchronous	multi-master	Full-duplex	>1,000
MICROWIRE	Synchronous	master/slave	Full-duplex	> 625
1-WIRE	Asynchronous	master/slave	Half-duplex	16

Baud Rate Concepts:

Data transfer rate in serial communication is measured in terms of bits per second (bps). This is also called as Baud Rate. Baud Rate and bps can be used inter changeably with respect to UART.

Ex: The total number of bits gets transferred during 10 pages of text, each with 100 ×25 characters with 8 bits per character and 1 start & stop bit is:
For each character a total number of bits are 10. The total number of bits is:
100 × 25 × 10 = 25,000 bits per page. For 10 pages of data it is required to transmit 2, 50,000
bits. Generally baud rates of SCI are 1200, 2400, 4800, 9600, 19,200 etc. To transfer 2, 50,000bits at a baud rate of 9600, we need: 250000/9600 = 26.04 seconds (27 seconds).

Synchronous/Asynchronous Interfaces (like UART, SPI, I2C, and USB):

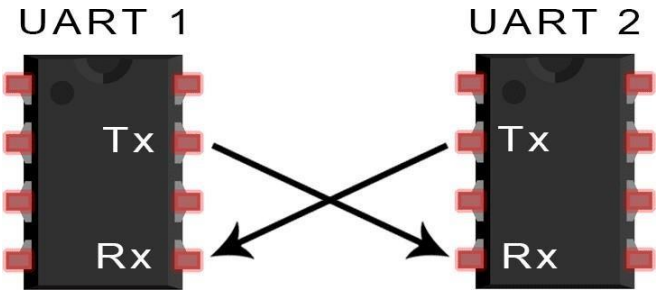
Serial communication protocols can be categorized as Synchronous and Asynchronous protocols. In synchronous communication, data is transmission and receiving is a continuous stream at a constant rate. Synchronous communication requires the clock of transmitting device and receiving device synchronized. In most of the systems, like ADC, audio codes, potentiometers, transmission and reception of data occurs with same frequency. Examples of synchronous communication are: I2C, SPI etc. In the case of asynchronous communication, the transmission of data requires no clock signal and data transfer occurs intermittently rather than steady stream. Handshake signals between the transmitter and receiver are important in asynchronous communications. Examples of asynchronous communication are Universal Asynchronous Receiver Transmitter (UART), CAN etc.

Synchronous and asynchronous communication protocols are well-defined standards and can be implemented in either hardware or software. In the early days of embedded systems, Software implementation of I^2C and SPI was common as well as a tedious work and used to take long programs. Gradually, most the microcontrollers started incorporating the standard communication protocols as hardware cores. This development in early 90‟s made job of the embedded software development easy for communication protocols.

Microcontroller of our interest TM4C123 supports UART, CAN, SPI, I^2C and USB protocols. The five (UART, CAN, SPI, I^2C and USB) above mentioned communication protocols are available in most of the modern day microcontrollers. Before studying the implementation and programming details of these protocols in TM4C123, it is required to understand basic standards, features and applications. In the following sections, we discuss fundamentals of the above mentioned communication protocols.

UART COMMUNICATION

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:



UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

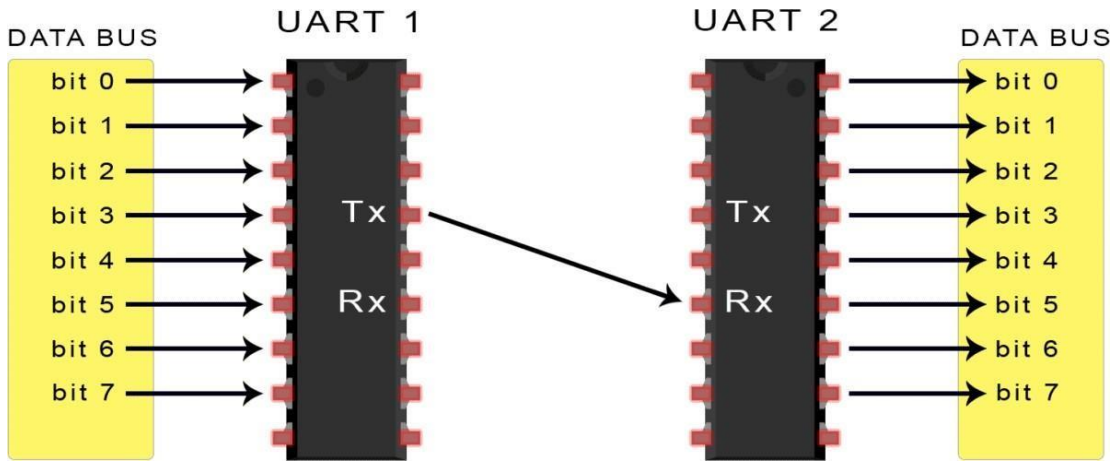
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baudrate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.

Both UARTs must be configured to transmit and receive the same data packet structure.

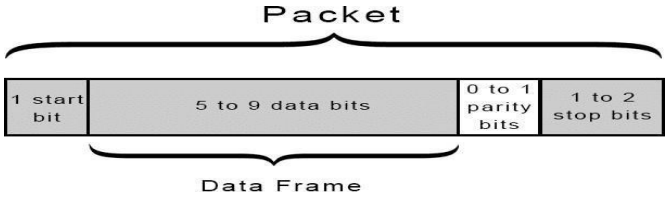
Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

HOW UART WORKS

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



START BIT

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

DATA FRAME

The data frame contains the actual data being transferred. It can be 5 bits to 9 bits long if a parity bit is used. If no parity bit is used, the data frame can be 8 bits long. In most cases, the data is sent with the least significant bit first.

PARITY

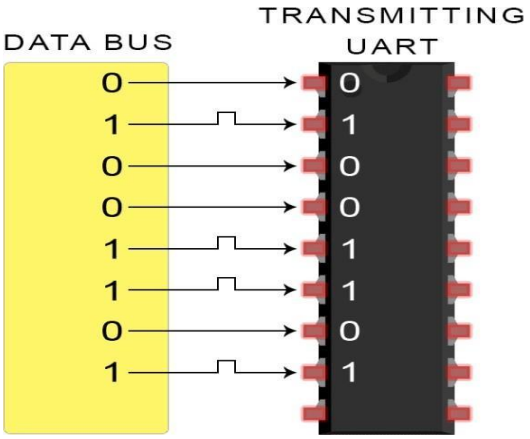
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a1, and the total is even, the UART knows that bits in the data frame have changed.

STOP BITS

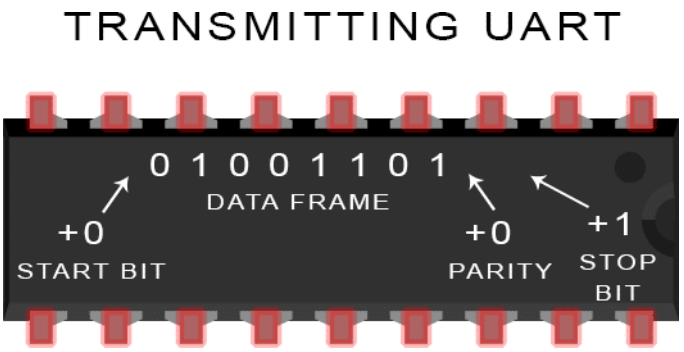
The Stop Bit, as the name suggests, marks the end of the data packet. It is usually two bits long but often only on bit is used. In order to end the transmission, the UART maintains the data line at high voltage (1).

STEPS OF UART TRANSMISSION

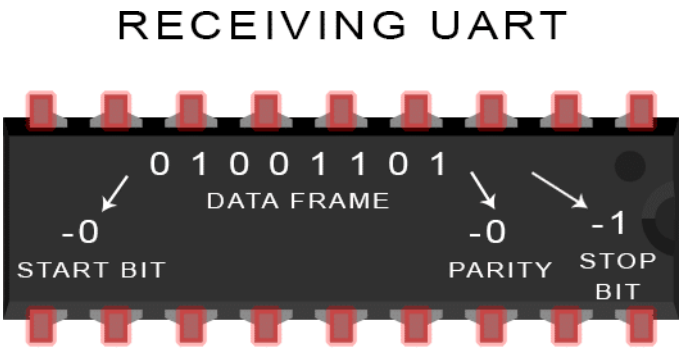
1. The transmitting UART receives data in parallel from the data bus:



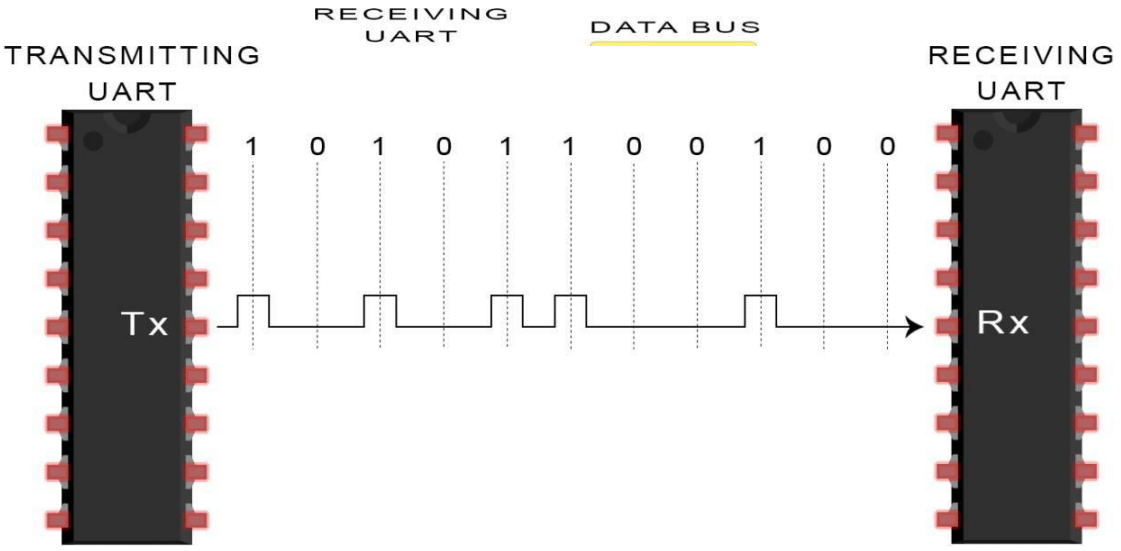
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



3. The entire packet is sent serially from the transmitting UART to the receiving UART.The receiving UART samples the data line at the pre-configured baud rate:



4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts

6. the serial data back into parallel and transfers it to the data bus on the receiving end:

ADVANTAGES AND DISADVANTAGES OF UARTS

No communication protocol is perfect, but UARTs are pretty good at what they do. Here

are some pros and cons to help you decide whether or not they fit the needs of your project:

ADVANTAGES

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

DISADVANTAGES

- The size of the data frame is limited to a maximum of 9 bits
- Doesn't support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

UART or Universal Asynchronous Receiver Transmitter is a dedicated hardware associated with serial communication. The hardware for UART can be a circuit integrated on the microcontroller or a dedicated IC. This is contrast to SPI or I2C, which are just communication protocols.

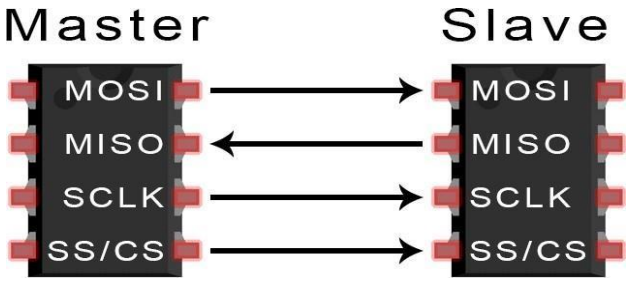
UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.

SPI COMMUNICATION PROTOCOL

SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below).



MOSI (Master Output/Slave Input) – Line for the master to send data to the slave. **MISO (Master Input/Slave Output)** – Line for the slave to send data to the master **SCLK (Clock)** – Line for the clock signal.

SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

Wires Used	4
Maximum Speed	Up to 10 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	Theoretically unlimited*

*In practice, the number of slaves is limited by the load capacitance of the system, which reduces the ability of the master to accurately switch between voltage levels.

HOW SPI WORKS

THE CLOCK

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always

initiated by the master since the master configures and generates the clock signal.

Any communication protocol where devices share a clock signal is known as *synchronous*. SPI is a synchronous communication protocol. There are also *asynchronous* methods that don't use a clock signal. For example, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

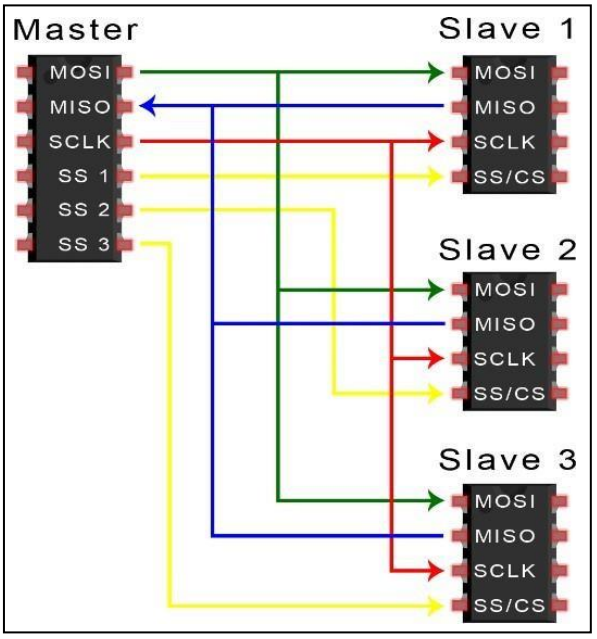
The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.

SLAVE SELECT

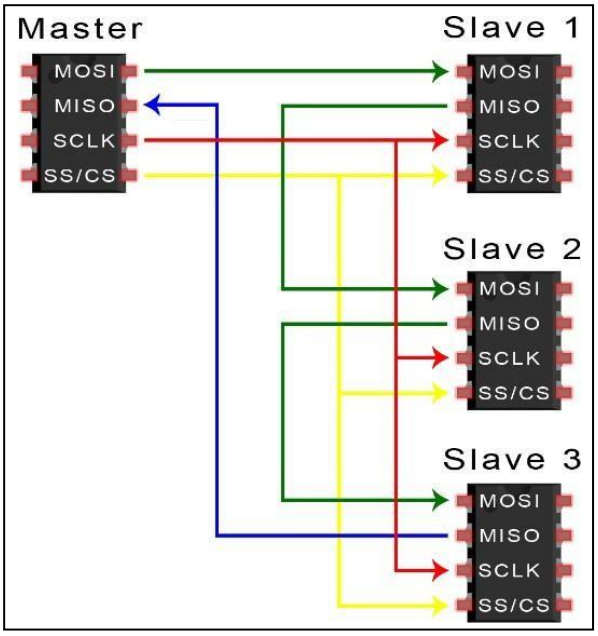
The master can choose which slave it wants to talk to by setting the slave's CS/SS line toa low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

MULTIPLE SLAVES

SPI can be set up to operate with a single master and a single slave, and it can be set up with multiple slaves controlled by a single master. There are two ways to connect multiple slaves to the master. If the master has multiple slave select pins, the slaves can be wired in parallel like this:



If only one slave select pin is available, the slaves can be daisy-chained like this:



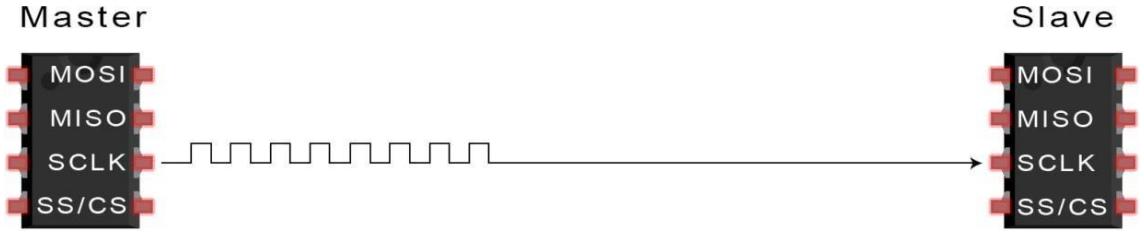
MOSI AND MISO

The master sends data to the slave bit by bit, in serial through the MOSI line. The slavereceives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

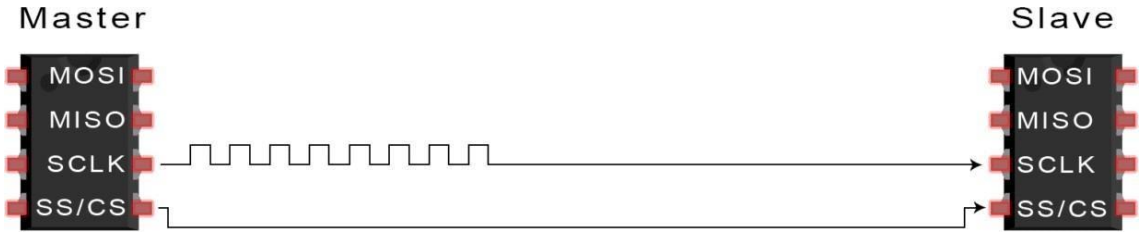
The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

STEPS OF SPI DATA TRANSMISSION

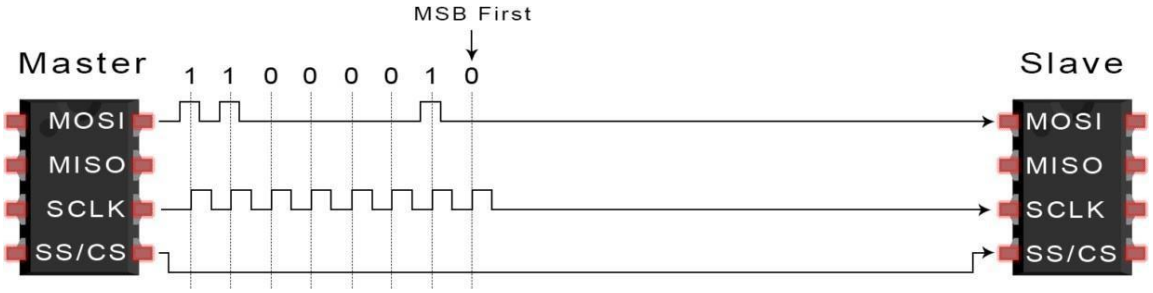
1. The master outputs the clock signal:



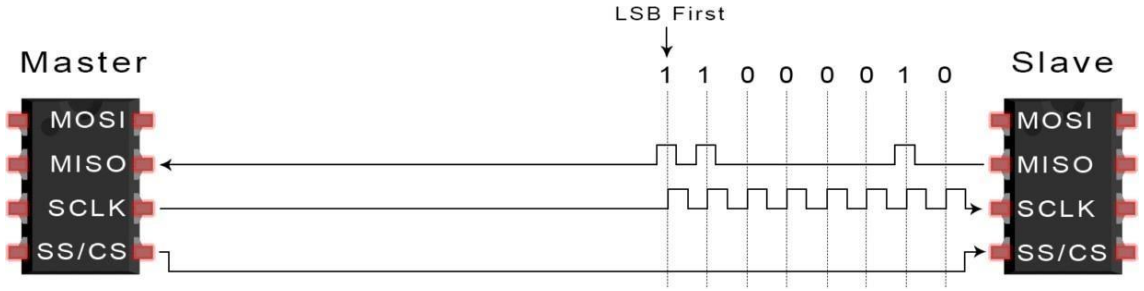
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along theMISOline. The master reads the bits as they are received:



ADVANTAGES AND DISADVANTAGES OF SPI

There are some advantages and disadvantages to using SPI, and if given the choice between different communication protocols, you should know when to use SPI according to the requirements of your project:

ADVANTAGES

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

DISADVANTAGES

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

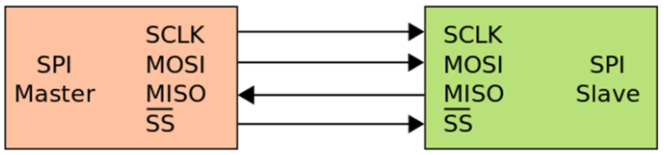


Fig: SPI Master connected to a single slave

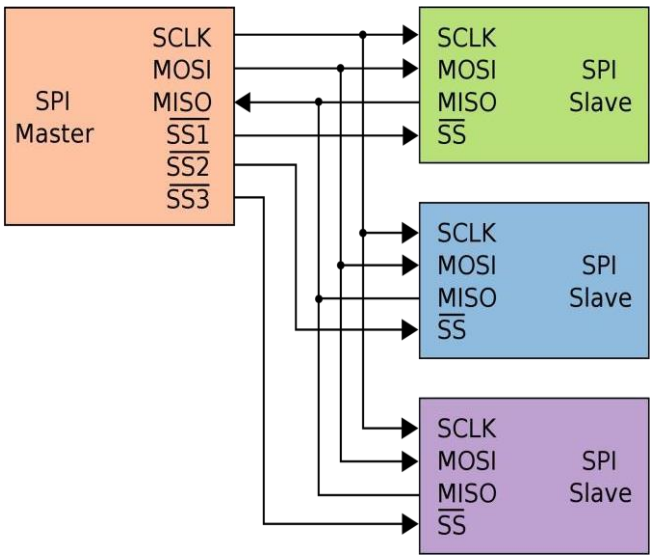


Fig: SPI master connected to multiple slaves

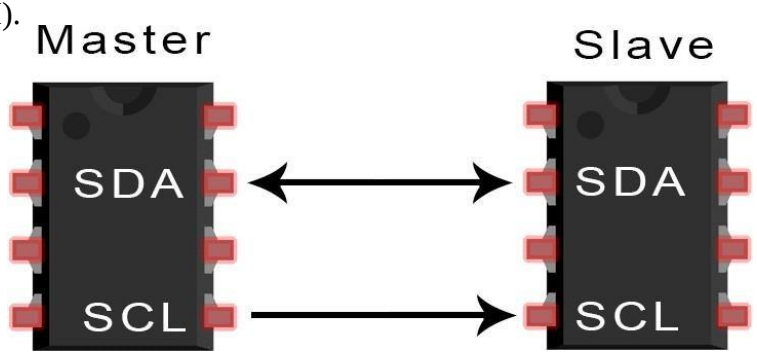
I2C COMMUNICATION PROTOCOL

Inter IC (i2c) (IIC) is important serial communication protocol in modern electronic systems. Philips invented this protocol in 1986. The objective of reducing the cost of production of television remote control motivated Philips to invent this protocol. IIC is a serialbus interface, can be implemented in software, but most of the microcontrollers support IIC by incorporating it as hard IP (Intellectual Property). IIC can be used to interface microcontroller with RTC, EEPROM and different variety of sensors. IIC is used to interface chips on motherboard, generally between a processor chip and any peripheral which supports IIC. IIC is very reliable wireline communication protocol for an on board or short distances. I2C is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller

logging data to a single memory card or displaying text to a single LCD.

IIC protocol uses two wires for data transfer between devices: Serial Data Line (SDA) and Serial Clock Line (SCL). The reduction in number of pins in comparison with parallel data transfer is evident. This reduces the cost of production, package size and power consumption. IIC is also best suited protocol for battery operated devices. IIC is also referred as two wire serial interface (TWI).



SDA (Serial Data) – The line for the master and slave to send and receive data.

SCL (Serial Clock) – The line that carries the clock signal.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire(the SDA line).

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

Wires Used	2
Maximum Speed	Standard mode= 100 kbps
	Fast mode= 400 kbps
	High speed mode= 3.4 Mbps
	Ultra fast mode= 5 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	Unlimited
Max # of Slaves	1008

GENERAL ELECTRICAL CHARACTERISTICS OF I2C

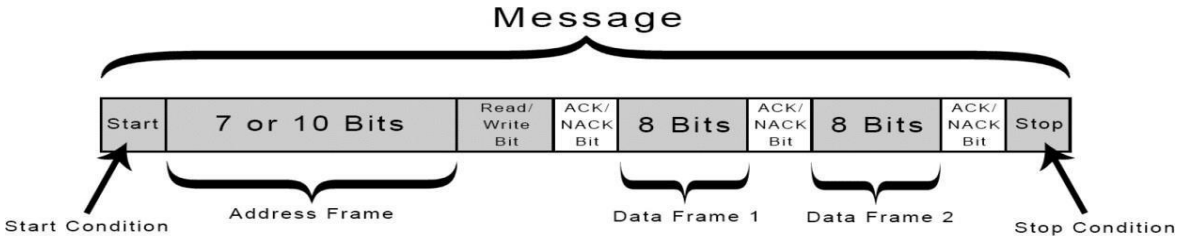
To implement I2C (For TIVA series microcontrollers or for most of the microcontrollers) a 4.7kilo ohm pull-up resistor for each line is needed. This is required to implement wired-AND logic in IIC.

More than 100 devices can be connected to I2C bus theoretically. It is better to restrict

to 15 devices for better performance of the network. Each device is called as node. Nodes which generates clock are called Master nodes and devices which work based on the clock generated by master node are called Slave nodes. Generally, master nodes initiate and terminate the transmission. The four possible modes of operation are: master transmitter, master receiver, slave transmitter and slave receiver.

HOW I2C WORKS

With I2C, data is transferred in *messages.* Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:



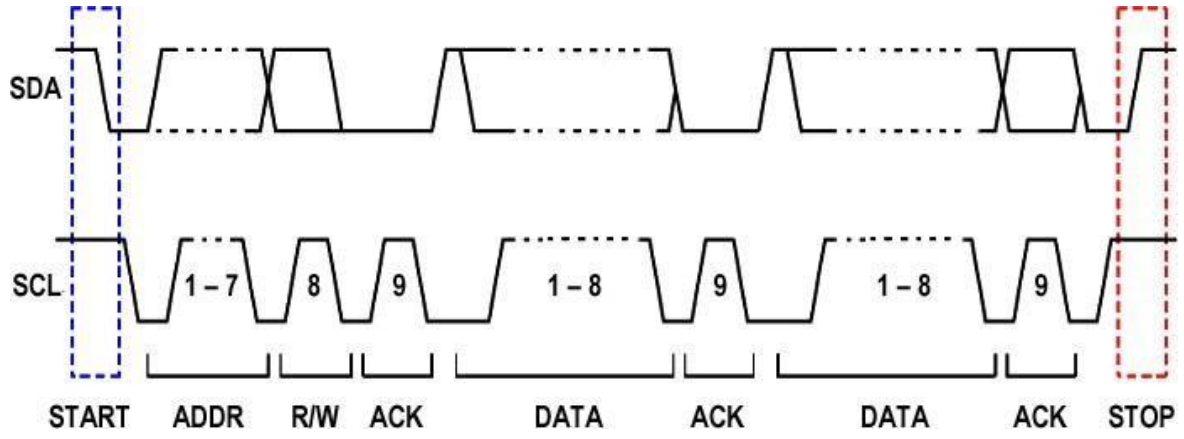
Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

Stop Condition: The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

Address Frame: A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

ACK/NACK Bit: Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.



ADDRESSING

I2C doesn't have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

READ/WRITE BIT

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

THE DATA FRAME

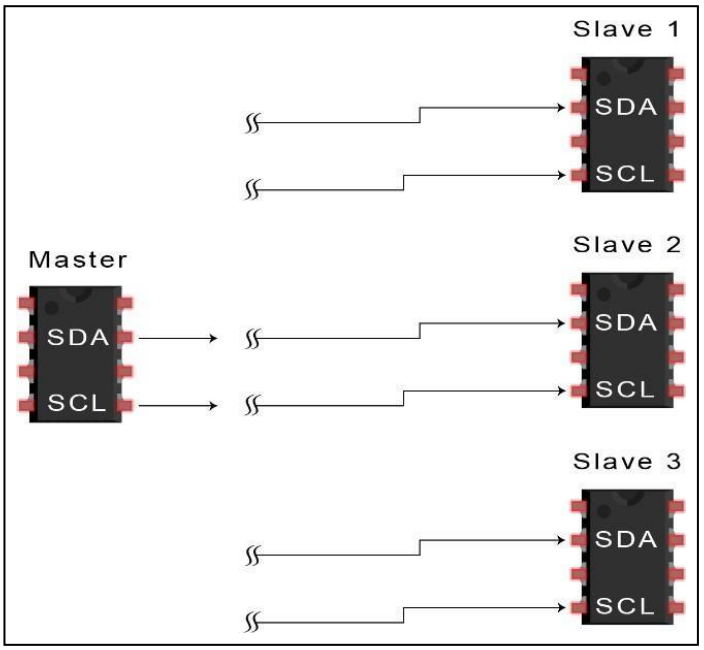
After the master detects the ACK bit from the slave, the first data frame is ready tobe sent.

The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent.

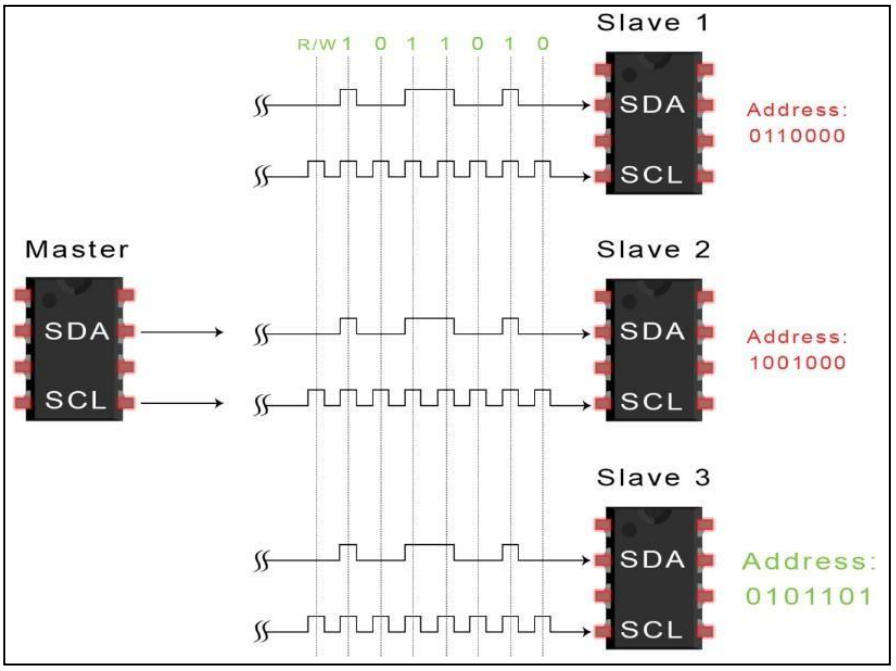
After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

STEPS OF I2C DATA TRANSMISSION

1. The master sends the start condition to every connected slave by switching the SDA linefrom a high voltage level to a low voltage level *before* switching the SCL line from high to low:

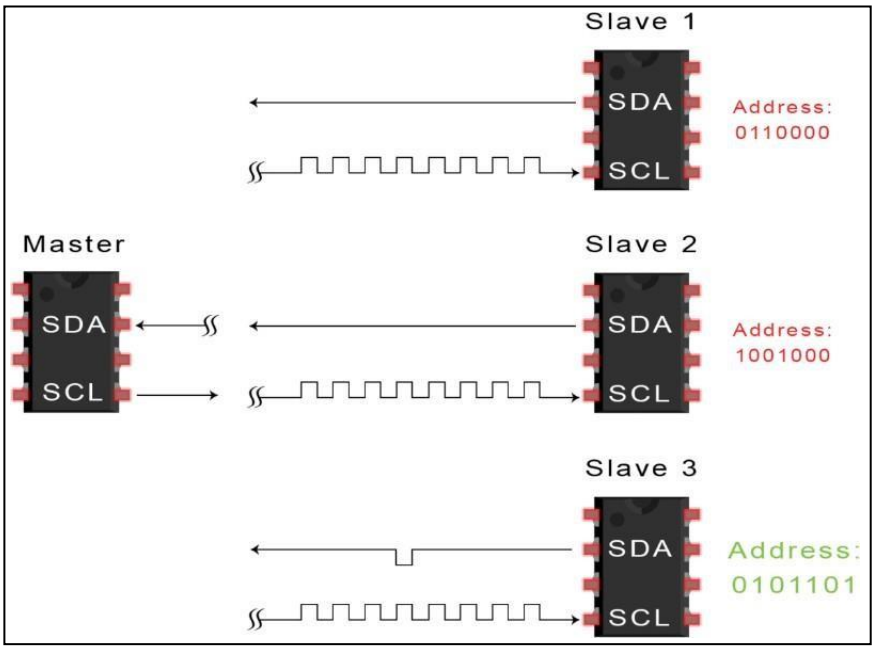


2. The master sends each slave the 7 or 10 bit address of the slave it wants to

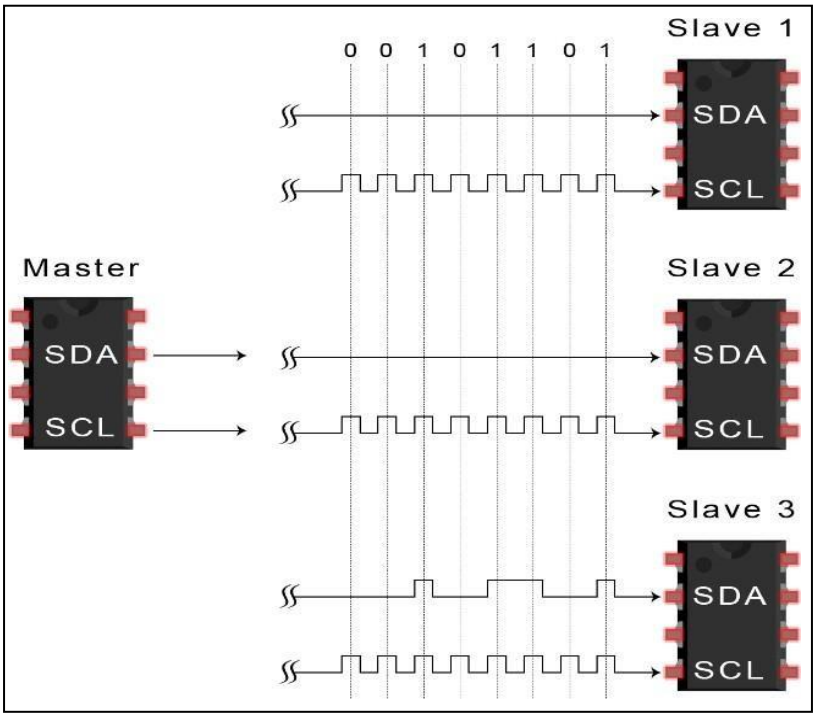


Communicate with, along with the read/write bit:

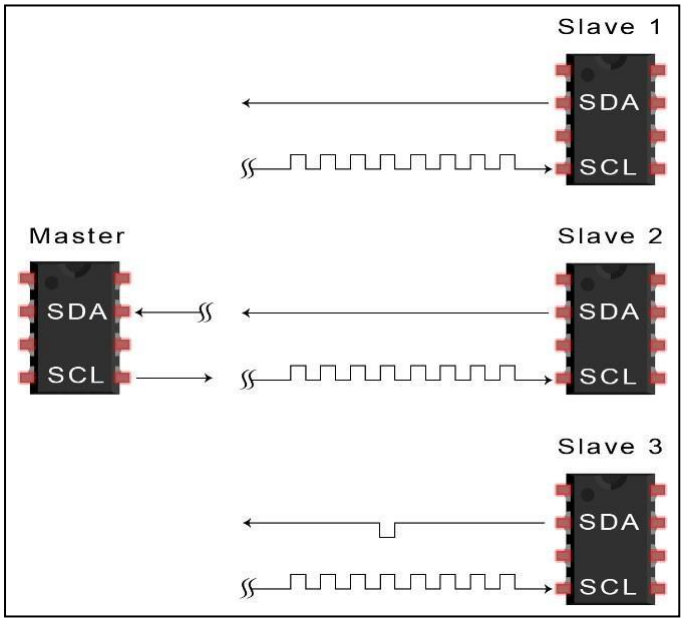
3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



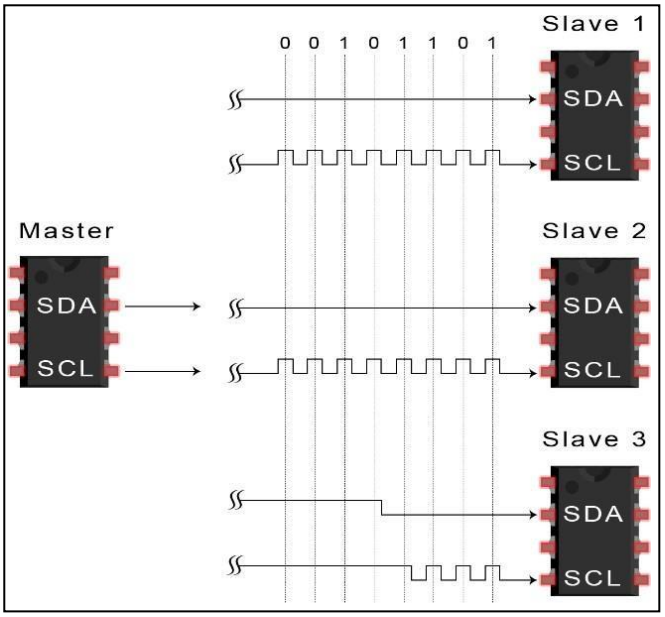
4. The master sends or receives the data frame:



5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:



6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:



SINGLE MASTER WITH MULTIPLE SLAVES

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 (2^7) unique address are available. Using 10 bit addresses is uncommon, but provides 1,024 (2^{10}) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K/10K Ohm pull-up resistors connecting the SDA and SCLlines to Vcc:

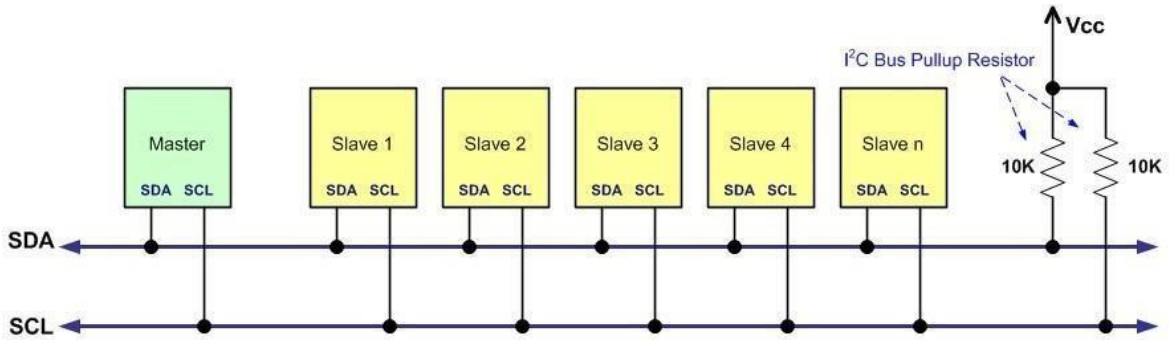
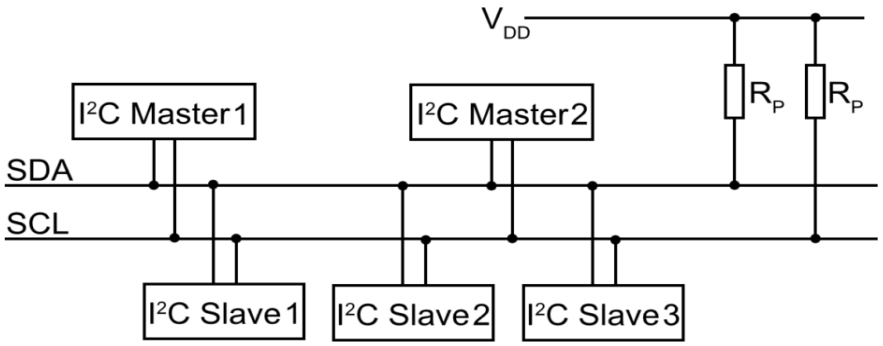


Figure: I2C Bus Connection

MULTIPLE MASTERS WITH MULTIPLE SLAVES

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



ADVANTAGES AND DISADVANTAGES OF I2C

There is a lot to I2C that might make it sound complicated compared to other protocols,but there are some good reasons why you may or may not want to use I2C to connect to a particular device:

ADVANTAGES

- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs
- Well known and widely used protocol

DISADVANTAGES

- Slower data transfer rate than SPI
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI

4.8.2 Parallel Protocols

Examples of Parallel Communication Protocols are **ISA, MCA,EISA,VESA,PCI**.**ISA Bus**

This is the most common type of early expansion bus, which was designed for use in the original IBM PC. The IBM PC-XT used an 8-bit bus design. This means that the data transfers take place in 8 bit chunks (i.e. one byte at a time) acrossthe bus. The ISA bus ran at a clock speed of 4.77 MHz.

For the 80286-based IBM PC-AT, an improved bus design, which could transfer 16-bits of data at a time, was announced. The 16-bit version of the ISA busis sometimes known as the AT bus. (AT-Advanced Technology)

The improved AT bus also provided a total of 24 address lines, which allowed 16MB of memory to be addressed. The AT bus was backward compatible with its 8-bit predecessor and allowed 8-bit cards to be used in 16-bit expansion slots.

When it first appeared the 8-bit ISA bus ran at a speed of 4.77MHZ – the same speed as the processor. It was improved over the years and eventually the ATbus ran at a clock speed of 8MHz.

MCA (Micro Channel Architecture)

This bus was developed by IBM as a replacement for ISA when they designed the PS/2 PC which was launched in 1987.

The bus offered a number of technical improvements over the ISA bus. For instance, the MCA runs at a faster speed of 10MHz and can support either 16-bit or32- bit data. It also supports bus mastering - a technology that placed a mini- processor on each expansion card. These mini-processors controlled much of the data transfer allowing the CPU to perform other tasks.

One advantage of MCA was that the plug-in cards were software configurable i.e. they required minimal intervention by the user when configuring.

The MCA expansion bus did not support ISA cards and IBM decided to charge other manufacturers royalties for use of the technology. This made itunpopular and it is now an obsolete technology.

EISA (Extended Industry Standard Architecture)

It was developed by a group of manufactures as an alternative to MCA. It was designed to use a 32-bit data path and provided 32 address lines giving access to 4GBof memory.

Like the MCA, EISA offered a disk-based setup for the cards, but it still ranat 8MHz in order for it to be compatible with ISA.

The EISA expansion slots are twice as deep as an ISA slot. If an ISA card isplaced in an EISA slot it will use only the top row of connectors, whereas a full EISA card uses both rows. It offered bus mastering.

EISA cards were relatively expensive and were normally found on high-end workstations and network servers.

VESA Bus

Also known as the Local bus or the VESA-Local bus. VESA (Video Electronics Standards Association) was invented to help standardize PCs video specifications, thus solving the problem of proprietary technology where different manufacturers were attempting to develop their own buses.

The VL Bus provides 32-bit data path and can run at 25 or 33MHZ. It ran atthe same clock frequency as the host CPU. But this became a problem as processorspeeds increased because, the faster the peripherals are required to run, the more expensive they are to manufacture.

It was difficult to implement the VL-Bus on newer chips such as the 486s and the new Pentiums and so eventually the VL-Bus was superseded by PCI.

VESA slots have extra set of connectors and therefore the cards are larger.

The VESA design was backward compatible with the older ISA cards.

Peripheral Component Interconnect (PCI)

Peripheral Component Interconnect (PCI) is one of the latest developments in bus architecture and is the current standard for PC expansion cards. It wasdeveloped by Intel and launched as the expansion bus for the Pentium processor in 1993. It is a local bus like VESA i.e. it connects the CPU, memory and peripherals to wider, faster data pathway.

PCI supports both 32-bit and 64-bit data width; therefore it is compatible with 486s and Pentiums. The bus data width is equal to the processor, for example,a 32 bit processor would have a 32 bit PCI bus, and operates at 33MHz.

PCI was used in developing Plug and Play (PnP) and all PCI cards supportPnP i.e. the user can plug a new card into the computer, power it on and it will “self identify” and “self specify” and start working without manual configuration using jumpers.

Unlike VESA, PCI supports bus mastering that is, the bus has some processing capability and therefore the CPU spends less time processing data. MostPCI cards are designed for 5v, but there are also 3v and dual-voltage cards, Keyingslots are used to differentiate 3v and 5v cards and slots to ensure that a 3v card is notslotted into a 5v socket and vice versa.

4.8.3 Wireless Protocols

IoT (Internet of Things) has power to make the complete system automatic. There are various **IOT communication protocols** which are used in communication between devices inthe IoT network. **The wireless communication protocol** is a standard set of rules withreference to which various electronic devices communicate with each other wirelessly.

Since there are many wireless communication protocols available to use for your product, it becomes difficult for the product designers to choose the correct one but once the scope of IoT application is decided it would become easier to select the right protocol. Here we are briefly explaining some **protocols used in IOT** with their features and applications.

Wi-Fi

Wi-Fi (Wireless Fidelity) is the most popular **IOT communication protocols** for wireless local area network (WLAN) that utilizes the IEEE 802.11 standard through 2.4 GHz UHF and5 GHz ISM frequencies. Wi-Fi provides Internet access to devices that are within the range ofabout 20 - 40 meters from the source. It has a data rate up to 600 Mbps maximum, depending on channel frequency used and the number of antennas. In embedded systems, ESP series controllers from Espressif are popular for building IoT based Applications.

ESP32 and **ESP8266** are the most commonly use wi-fi modules for embedded applications. In terms of using the Wi-Fi protocol for IOT, there are some pros & cons to be considered. The infrastructure or device cost for Wi-Fi is low & deployment is easy but the power consumptionis high and the Wi-Fi range is quite moderate. So, the Wi-Fi may not be the best choice for alltypes of IOT applications but it can be used for applications like Home Automation.

There are many development boards available that allow people to build IOT applications usingWi-Fi. The most popular ones are the Raspberry Pi and Node MCU. These boards allow peopleto build

IOT prototypes and also can be used for small real-time applications. Likewise is the Marvell Avastar 88W8997 SoC, which follows the Wi-Fi's IEEE 802.11n standard. The chip has applications like wearables, wireless audio & smart home.

Bluetooth

Bluetooth is a technology used for exchanging data wirelessly over short distances and preferred over various **IOT network protocols**. It uses short-wavelength UHF radio waves offrequency ranging from 2.4 to 2.485 GHz in the ISM band. The Bluetooth technology has 3 different versions based on its applications:

- **Bluetooth:** The Bluetooth that is used in devices for communication has many applications in IOT/M2M devices nowadays. It is a technology using which two devicescan communicate and share data wirelessly. It operates at 2.4GHz ISM band and the data is split in packets before sending and then is shared using any one of the designated79 channels operating at 1 MHz of bandwidth.
- **BLE (Bluetooth 4.0, Bluetooth Low Energy):** The BLE has a single main difference from Bluetooth that it consumes low power. With that, it makes the product of low cost& more long-lasting than Bluetooth.
- **iBeacon:** It is a simplified communication technique used by Apple and is completelybased on Bluetooth technology. The Bluetooth 4.0 transmits an ID called UUID for each user and makes it each to communicate between iPhone users.

Bluetooth has many applications, such as in telephones, tablets, media players, robotics systems, etc. The range of Bluetooth technology is between 50 – 150 meters and the data is being shared at a maximum data rate of 1 Mbps.

After launching the BLE protocol, there have been many new applications developed using Bluetooth in the field of IOT. They fall under the category of low-cost consumer products and Smart-Building applications. Like Wi-Fi, **Bluetooth also has a module Bluetooth HC-05 thatcan be interfaced with development boards like Arduino or Raspberry Pi to build DIY projects**. When it comes to Real-time applications, Marvell's Avastar 88W8977 comes with Bluetooth v4.2 and has features like high speed, mesh networking for IOT. Another product, M5600 is a wireless pressure transducer with a Bluetooth v4.0 embedded in it.

Zigbee

ZigBee is another **IOT wireless protocols** has features similar to the Bluetooth technology.But it follows the IEEE 802.15.4 standard and is a high-level communication protocol. It has some advantages similar to Bluetooth i.e. low-power consumption, robustness, high security, and high scalability.

Zigbee offers a range of about 10 – 100 meters maximum and data rate to transfer data between

communicated devices is around 250 Kbps. It has a large number of applications in technologies like M2M & IOT.

Having limitations in regards to data rate, range, and power consumption, Zigbee is only appropriate for Small-Scale Wireless applications. Though having some limitations**,** it providesa 128-bit AES encryption and is giving a big hand in making secure communication for Home automation & small Industrial applications. Zigbee too has its DIY modulenamed **XBee** & **XBee Pro** which can be interfaced with Arduino or Raspberry Pi boards to make simple projects or application prototypes.

The company Develco has made products using Zigbee technologies like Sensors, gateways, meter interfaces, smart plugs, smart relays, etc which all work on the Zigbee wireless Mesh network, consuming low power and free from external interferences. Another company, Datanet has Zigbee based products which are used in real-time applications already, like the DNL910 & DNL920.

RFID

Radio-frequency identification (RFID) is a technology that uses electromagnetic fields toidentify objects or tags which contains some stored information. The range of RFID variesfrom about 10cm to 200m maximum and such a long difference makes the two range have names like short-range distance and long-range distance. Since the range has a huge difference,the frequency at which the RFID operates has a huge difference too i.e. it starts from KHz andranges till GHz or can be said as frequency ranges from Low frequency (LF) to Microwave depending upon the application and distance of communication.

RFID has RC522 Arduino & Raspberry Pi compatible module that can be used to build an IOT based RFID application or application prototypes like attendance system.

Cellular

The cellular network has been in use since the last 2 decades and comprises of GSM/GPRS/EDGE(2G)/UMTS or HSPA(3G)/LTE(4G) communication protocols. This protocol is generally used for long-distance communications. The data can be sent of large size and with high speeds compared to other technologies.

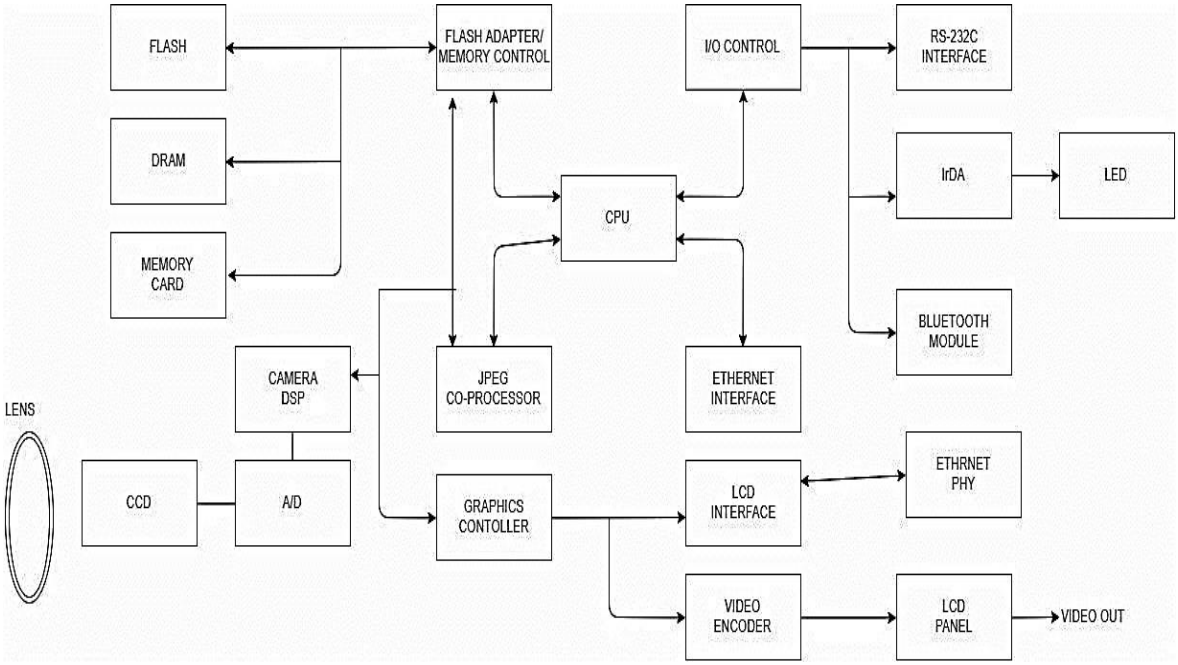
The operating frequencies range from 900 – 2100 MHz with a distance coverage of 35km to 200km and the data rates i.e. the speed of transferring data is from 35 Kbps to 10 Mbps. A company Quectel has cellular IOT products like EC21, EC23, EG91 and many more LTE standard products working on 4G. UMTS/HSPDA UC15, UC20, UC15 Mini & UC20 Mini are the 3G based IOT module launched by the same company.

4.9 Digital camera Example:

A digital camera is an example of sophisticated embedded system. It consists of a lot of components including the DSP processors. The fig(a) shows one possible block diagram of a digital camera.

Digital camera includes various types of memories like DRAM, memory card, flash memory with controller etc.

The CPU is the main processor is also connected with the various other processors. The host processor just controls the various operations and the complicated operations are performed by these task processors.



- The JPEG co-processor is mainly meant to compress and decompose image into JPEG format.

- The camera DSP processes the images taken by CCD camera after it is converted to digital form. A graphics processor is also connected to do graphics processing for displaying the images and videos from the memory either on the LCD panel through the LCD controller interface or to the video out after video encoding.

- The IrDA interface is provided for remote controlling of the camera through the infrared remote. An Ethernet interface is given for Ethernet connection. There are other interfaces like RS232 and Bluetooth for advance communication support.

Part A

1.What is meant by writing skills and Storage permanence of memory?
2.What type of memory technology is generally used to store program variables in an embedded system?
3.Interpret about Composing memory
4.Compare serial and parallel communication interface
5.List out the examples of Wireless protocol?
6.Identify the functions of DMA
7.List the types of memory
8.How does a digital camera use an embedded system?

Part B

1.Explain in detail about operations of DMA
2.Examine in detail about Microprocessor interfacing I/O addressing
3.Explain in detail about bus arbitration mechanism in embedded system
4.Discuss about serial and parallel protocols used in embedded system

TEXT/ REFEENCE BOOKS

1.David E.Simon, "An Embedded Software Primer", Pearson Education,2001
2. Frank Vahid and Tony Gwargie, " Embedded System Design", John Wiley & Sons,2002
3. Steve Heath, "Embedded System Design", Elsevier, Second Edition,2004.

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

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT- 5 PROCESS MODELS AND HARDWARE SOFTWARE CO-DESIGN

UNIT 5 PROCESS MODELS AND HARDWARE SOFTWARE CO-DESIGN

Modes of operation Finite state machine HCFSL and state charts language state machine models Concurrent process model Concurrent process Communication among process Synchronization among process Implementation – Data Flow mode

5.1Mode of Operation

We implement a system's processing behavior with processors. But to accomplish this, we must have first described that processing behavior. One method we've discussed for describing processing behavior uses assembly language. Another, more powerful method usesa high-level programming language like C. Both these methods use what is known as a sequential program computation model, in which a set of instructions executes sequentially. A high-level programming language provides more advanced constructs for sequencing among the instructions than does an assembly language, and the instructions are more complex,but nevertheless, the sequential execution model (one statement at a time) is the same.

However, embedded system processing behavior is becoming very complex, requiring more advanced computation models to describe that behavior. The increasing complexity results from increasing IC capacity: the more we can put on an IC, the more functionality we want to put into our embedded system. Thus, while embedded systems previously encompassed applications like washing machines and small games requiring perhaps hundredsof lines of code, today they also extend to fairly sophisticated applications like television set- top boxes and digital cameras requiring perhaps hundreds of thousands of lines.

Trying to describe the behavior of such systems can be extremely difficult. The desired behavior is often not even fully understood initially. Therefore, designers must spend much time and effort simply understanding and describing the desired behavior ofa system, and some studies have found that most system bugs come from mistakes made describing the desired behavior rather than from mistakes in implementing that behavior. The common method today of using an English (or some other natural language) description of desired behavior provides a reasonable first step, but is not nearly sufficient, because English is not precise. Trying to describe a system precisely in English can be an arduous and often futile endeavor -- just look at any legal document forany example of attempting to be precise in a natural language.

A computation model assists the designer to understand and describe the behavior by providing a means to compose the behavior from simpler objects. A *computation model* provides a set of objects, rules for composing those objects, and execution semantics of the composed objects. For example, the sequential program model providesa set of statements,

rules for putting statements one after another, and semantics stating how the statements are executed one at a time. Unfortunately, this model is often not enough. Several other models are therefore also used to describe embedded system behavior. These include the communicating process model, which supports description ofmultiple sequential programs running concurrently. Another model is the state machine model, used commonly for control-dominated systems. A *control-dominated* system is one whose behavior consists mostly of monitoring control inputs and reacting by setting control outputs. Yet another model is the dataflow model, used for data-dominated systems. A *data-dominated* system's behaviorconsists mostly of transforming streams of input data into streams of output data, such as a system for filtering noise out of an audio signal as part of a cell phone. An extremely complexsystem may be best described using an object-oriented model, which provides an elegant meansfor breaking the complex system into simpler, well-defined objects.

A model is an abstract notion, and therefore we use *languages* to capture the model ina concrete form. For example, the sequential program model can be captured in a variety of languages, such as C, C++, Pascal, Java, Basic, Ada, VHDL, and Verilog. Furthermore, a single language can capture a variety of models. Languages typically are textual, but may also be graphical. For example, graphical languages have been proposed for sequential programming (though they have not been widely adopted).

5.1 State machine model

In a state machine model, we describe system behavior as a set of possible states; the system can only be in one of these states at a given time. We also describe the possible transitions from one state to another depending on input values. Finally, we describe the actions that occur when in a state or when transitioning between states.

For example, Figure 5.1 shows a state machine description of the Unit Control part of our elevator example. The initial state, Idle, sets upand downto 0 and open to 1. The state machine stays in state Idle until the requested floor differs from the current floor. If the requested floor is greater, then the machine transitions to state Going Up, which sets up to 1,whereas if the requested floor is less, then the machine transitions to state Going Down, whichsets down to 1. The machine stays in either state until the current floor equals the requested floor, after which the machine transitions to state Door Open, which sets open to 1. We assume the system includes a timer, so we startthe timer while transitioning to Door Open,.

We stay in this state until the timer says 10 seconds have passed, after which we transition back to the Idle state.

5.1.1 Finite-state machines: FSM

We have described state machines somewhat informally, but now provide a more formal definition. We start by defining the well-known finite-state machine computation model, or FSM, and then we'll define extensions to that model to obtain a more useful model for embedded system design. An FSM is a 6-tuple, <S, I, O, F, H, s_0>, where:

S is a set of states {s_0, s_1, … , s_l},I is a set of inputs {i_0, i_1, …, i_m},
O is a set of outputs {o_0, o_1, …, o_n},
F is a next-state function (i.e., transitions), mapping states and inputs to states (S XIS),
H is an output function, mapping current states to outputs (SO), ands_0 is an initial state.

The above is a Moore-type FSM above, which associates outputs with states. A second type of FSM is a Mealy-type FSM, which associates outputs with transitions, i.e., H maps S x I O. You might remember that Moore outputs are associated with states by noting thatthe name Moore has two o's in it, which look like states in a state diagram. Many tools that support FSM's support combinations of the two types, meaning we can associate outputs withstates, transitions, or both.

We can use some shorthand notations to simplify FSM descriptions. First, there may bemany system outputs, so rather than explicitly assigning every output in every state, we can say that any outputs not assigned in a state are implicitly assigned 0. Second, we often use an FSM to describe a single-purpose processor (i.e., hardware). Most hardware is synchronous, meaning that register updates are synchronized to clock pulses, e.g., registers are only updated on the rising (or falling) edge of a clock. Such an FSM would have every transition condition AND'ed with the clock edge (e.g., clock'rising and x and y). To avoid having to add this clock edge to every transition condition, we can simply say that the FSM is synchronous, meaning that every transition condition is implicitly AND'ed with the clockedge.

5.1.2 Finite-state machines with data paths: FSMD

When using an FSM for embedded system design, the inputs and outputs represent Boolean data types, and the functions therefore represent Boolean functions with Boolean operations. This model is sufficient for purely control systems that do not input or output data. However, when we must deal with data, two new features would be helpful: more complex data types (such as integers or floating point numbers), and variables to store data. Gajski refers to an FSM model extended to support more complex data types and variables as an FSM with data path, or FSMD. Most other authors refer to this model as anextended FSM, but there are many kinds of extensions and therefore we prefer the more precise nameof FSMD. One possible FSMD model definition is as follows:

<S, I, O, V, F, H, s_0> where:

S is a set of states {s_0, s_1, … , s_l},I is a set of inputs {i_0, i_1, …, i_m},
O is a set of outputs {o_0, o_1, …, o_n}, V is a set of variables {v_0, v_1, …, v_n},

F is a next-state function, mapping states and inputs and variables to states(S XIX V S),
H is an action function, mapping current states to outputs and variables (S O
1 V), and
s_0 is an initial state.

In an FSMD, the inputs, outputs and variables may represent various data types (perhaps as complex as the data types allowed in a typical programming language), and the functions F and H therefore may include arithmetic operations, such as addition, rather than just Boolean operations as in an FSM. We now call H an action function rather than an output function, since it describes not just outputs, but also variable updates. Note that the above definition is for a Moore-type FSMD, and it could easily be modified for a Mealy type or a combination of the two types. During execution of the model, the complete system state consists not only of the current state s_i, but also the values of all variables. Our earlier state machine description of Unit Control was an FSMD, since it had inputs whose data types wereintegers, and had arithmetic operations (comparisons)in its transition conditions.

5.1.3 Describing a system as a state machine

Describing a system's behavior as a state machine (in particular, as an FSMD) consistsof several steps:

1. List all possible states, giving each a descriptive name.
2. Declare all variables.
3. For each state, list the possible transitions, withassociated conditions, to other states.
4. For each state and/or transition, list the associated actions
5. For each state, ensure that exiting transition conditions are exclusive(no two conditions could be true simultaneously) and complete (oneof the conditions is true at any time).

If the transitions leaving a state are not exclusive, then we have a non- deterministic state machine. When the machine executes and reaches a state with more than one transition that could be taken, then one of those transitions is taken, but we don't know which one that would be. The non-determinism prevents having to over-specify behavior in some cases, and may result in don't-cares that may reduce hardware size, but we won'tfocus on non-deterministic state machines in this book.

If the transitions leaving a state are not complete, then that usually means that we stay inthat state until one of the conditions becomes true. This way of reducing the number of explicit transitions should probably be avoided when first learning to use state machines.

5.1.4 Comparing the state machine and sequential program models

Many would agree that the state machine model excels over the sequential program model for describing a control-based system like the elevator controller. The state machine model is designed such that it encourages a designer to think of all possible states of the system, and to think of all possible transitions among states based on possible inputconditions. The sequential program model, in contrast, is designed to transform data througha series of instructions that may be iterated and conditionally executed. Each encourages a different way of thinking of a system's behavior.

A common point of confusion is the distinction between state machine and sequential program models versus the distinction between graphical and textual languages. In particular,

a state machine description excels in many cases, not because of its graphical representation,but rather because it provides a more natural means of computing for those cases; it can be captured textually and still provide the same advantage. For example, while in Figure 8.2 wedescribed the elevator's Unit Controlas a state machine captured in a graphical state-machinelanguage, called a state diagram, we could have instead captured the state machine in a textualstate-machine language. One textual language would be a state table, in which we list each state as an entry in a table. Each state's row would list the state's actions. Each row would also list all possible input conditions, and the next state for each such condition. Conversely,while in Figure 5.1 we described the elevator's Unit Control as a sequential program capturedusing a textual sequential programming language (in this case C), we could have instead captured the sequential program using a graphical sequential programming language, such asa flowchart.

Figure 5.1: Capturing the elevator's Unit Control state machine in a sequentialprogramming language (in this case C).

```
#define  IDLE          0
#define GOINGUP        1
#define GOINGDN        2
#define DOOROPEN       3
void UnitControl()
{
  int state = IDLE;
  while (1) {
    switch (state) {
      IDLE:       up=0; down=0; open=1; timer_start=0;
        if  (req==floor)  {state = IDLE;}
        if  (req > floor)    {state = GOINGUP;}
        if  (req  <  floor)  {state  =  GOINGDN;}
        break;
      GOINGUP: up=1; down=0; open=0; timer_start=0;
        if  (req > floor)  {state = GOINGUP;}
        if  (!(req>floor))  {state  =  DOOROPEN;}
        break;
      GOINGDN: up=1; down=0; open=0; timer_start=0;
        if  (req > floor)  {state = GOINGDN;}
        if  (!(req>floor))  {state  =  DOOROPEN;}
        break;
      DOOROPEN: up=0; down=0; open=1; timer_start=1;
        if (timer < 10) {state = DOOROPEN;}
        if (!(timer<10)){state = IDLE;}
        break;
    }
  }
}
```

5.1.5 Capturing a state machine model in a sequential programming language

As elegant as the state machine model is for capturing control-dominated systems, the fact remains that the most popular embedded system development tools use sequential programming languages like C, C++, Java, Ada, VHDL or Verilog. These tools are typically complex and expensive, supporting tasks like compilation, synthesis, simulation, interactive debugging, and/or in-circuit emulation. Unfortunately, sequential programming languages do not directly support the capture of state machines, i.e., they don't possess specific constructs corresponding to states or transitions. Fortunately, we can still describe our system using a state machine model while capturing the model in a sequential program language, by using one of two approaches.

In a front-end tool approach, we install an additional tool that supports a state machine language. These tools typically define graphical and perhaps textual state machine languages, and include nice graphic interfaces for drawing and displaying states as circles and transitions as directed arcs. They may support graphical simulation of the state machine, highlighting the current state and active transition. Such tools automatically generate code in a sequential program language (e.g., C code) with the same functionality as the state machine. This sequential program code can then be input to our main development tool. In many cases, the front-end tool is designed to interface directly with our main development tool, so that we can control and observe simulations occurring in the development tool directly from the front-end tool. The drawback of this approach is that we must support yet another tool, which includes additional licensing costs, version upgrades, training, integration problems with our development environment, and so on.

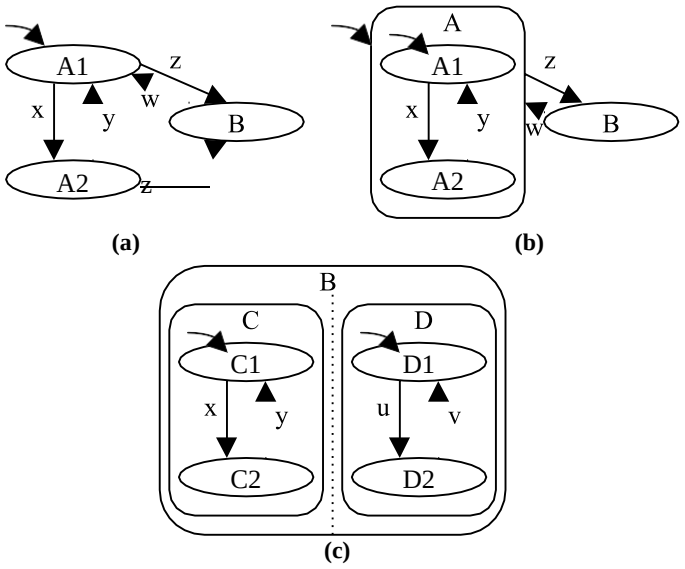
In contrast, we can use a language subset approach. In this approach, we directly capture our state machine model in a sequential program language, by following a strict set of rules for capturing each state machine construct in an equivalent set of sequential program constructs. This approach is by far the most common approach for capturing state machines, both in software languages like C as well as hardware languages like VHDL and Verilog. We now describe how to capture a state machine model in a sequential program language.

Figure 5.2: General template for capturing a state machine in a sequential programming language.

```
#define S0                  0
#define S1                  1
...
#define SN                  N
void StateMachine()
{
  int state = S0; // or whatever is the initial state.
  while (1) {
    switch (state) {
      S0:
        // Insert S0's actions here
        // Insert transitions Ti leaving S0:
        if (T0's condition is true ) {state = T0's next state; // insert T0's actions here.}
        if  (T1's condition is true ) {state = T1's next state; // insert T1's actions here. }
        ...
        if (Tm's condition is true ) {state = Tm's next state; // insert Tm's actions here. }
        break;
      S1:
        // Insert S1's actions here
        // Insert transitions Ti leaving S1
        break;
      ...
      SN:
        // Insert SN's actions here
        // Insert transitions Ti leaving SN
        break;
    }
  }
}
```

We start by capturing our Unit Control state machine in the sequential programminglanguage C, illustrated in Figure 8.3. We enumerate all states, in this case using the #define C construct. We capture the state machine as a subroutine, in which we declare a state variable initialized to the initial state. We then create an infinite loop, containing a single switch statementthat branches to the case corresponding to the value of the state variable. Each state's case startswith the actions in that state, and then the transitions from that state. Each transition is capturedas an if statement that checks if the transition's condition is true and then sets the next state. Figure 5.2 shows a general template for capturing a state machine in C.

Figure 5.3: Adding hierarchy and concurrency to the state machine model: (a) three-state example without hierarchy, (b) same example with hierarchy, (c) concurrency.



5.1.6 Hiererarchical/Concurrent state machines (HCFSM)and State charts

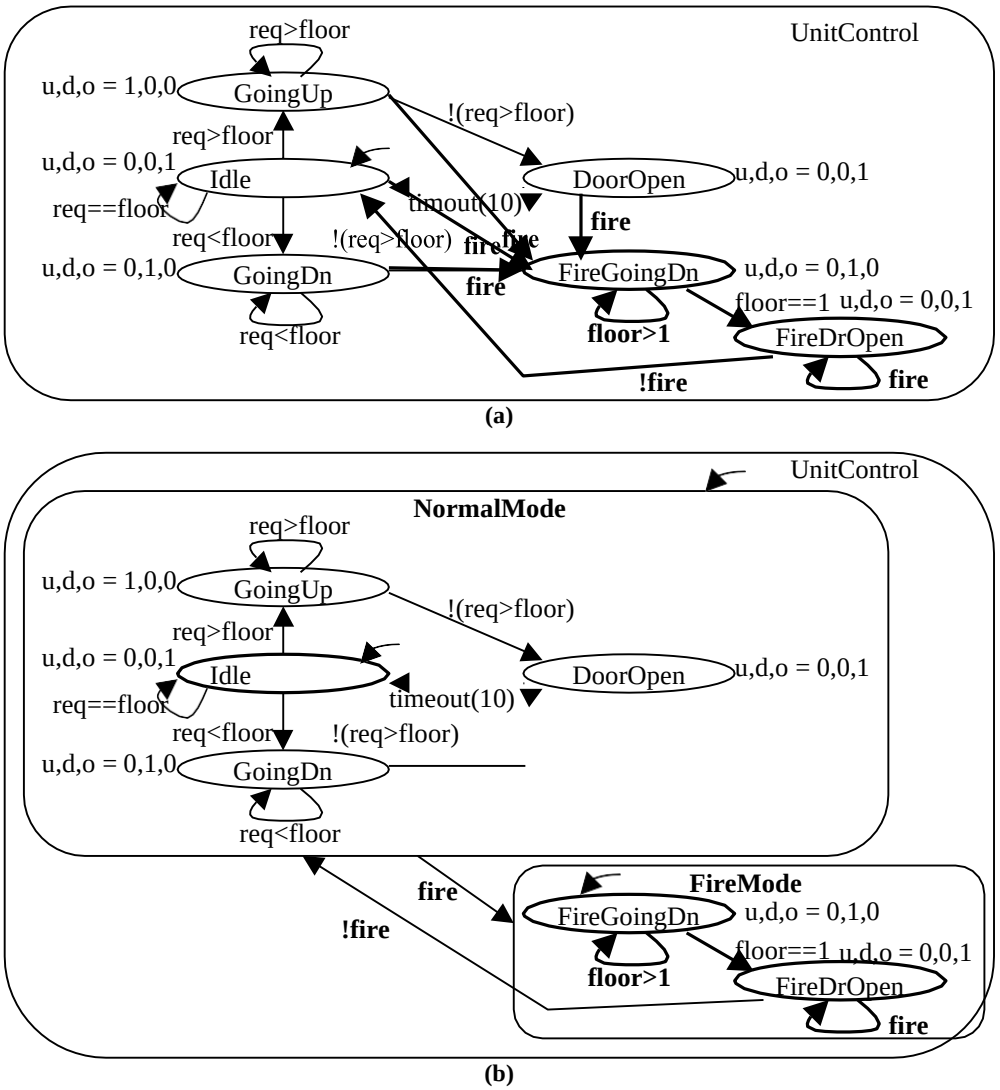
Harel proposed extensions to the state machine model to support hierarchy and concurrency, and developed Statecharts, a graphical state machine language designed to capture that model. We refer to the model as a hierarchical/concurrent FSM, or HCFSM.

The hierarchy extension allows us to decompose a state into another state machine, or conversely stated, to group several states into a new hierarchical state. For example, consider the state machine in Figure 5.3(a), having three states A1 (the initial state), A2, and B. Whenever we are in either A1 or A2 and event z occurs, we transition to state B. We can simplify this state machine by grouping A1 and A2 into a hierarchical state A, as shown in Figure 5.3(b). State A is the initial state, which in turn has an initial state A1. We draw the transition to B on event z as originating from state A, not A1 or A2. The meaning is that regardless of whether we are in A1 or A2, event z causes a transition to state B.

As another hierarchy example, consider our earlier elevator example, and suppose that we want to add a control input fire, along with new behavior that immediately moves the elevator down to the first floor and opens the door when fire is true. As shown in Figure 5.4(a), we can capture this behavior by adding a transition from every state originally in Unit Control to a new state called Fire Going Dn, which moves the elevator to the first floor, followed by a state Fire Dr Open, which holds the door open on the first floor. When fire becomes false, we

go to the Idle state. While this new state machine captures the desired behavior, it is becoming more complex due to many more transitions, and harder to comprehend due to more states. We can use hierarchy to reduce the number of transitions and enhance understandability. As shown inFigure 5.4(b), we can group the original state machine into a hierarchical state called Normal Mode, and group the fire-related states into a state called Fire Mode. This grouping reduces the number of transitions, since instead of four transitions from each original state to the fire-related states, we need only one transition, from Normal Mode to Fire Mode. This grouping also enhances understandability, since it clearly represents two main operating modes, one normal and one in case of fire.

Figure 5.4: The elevator's UnitControl with new behavior for a new input *fire*: (a) without hierarchy, (b) with hierarchy.



The concurrency extension allows us to use hierarchy to decompose a state into two

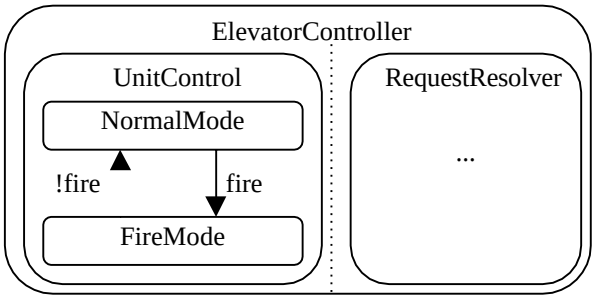
concurrent states, or conversely stated, to group two concurrent states into a new hierarchical state. For example, Figure 5.3 (c), shows a state B decomposed into two concurrent states C and

D. C happens to be decomposed into another state machine, as does D. Figure 5.5 shows the entireElevator Controller behavior captured as a HCFSM with two concurrent states.

Therefore, we see that there are two methods for using hierarchy to decompose a state into substates. OR-decomposition decomposes a state into sequential states, in which only one state is active at a time (either the first state OR the second state OR the third state, etc.). AND- decomposition decomposes a state into concurrent states, all of which are active at a time (the firststate AND the second state AND the third state, etc.).

The State charts language includes numerous additional constructs to improve state machinecapture. A timeout is a transition with a time limit as its condition.

Figure 5.5: Using concurrency in an HCFSM to describe both processes of the ElevatorController.



The transition is automatically taken if the transition source state is active for an amount of time equal to the limit. Note that this would have simplified the Unit Control state machine; rather than starting and checking an external timer, we could simply have created a transition from Door Open to Idle with the condition time out (10). History is a mechanism for remembering the last substate that an OR-decomposed state A was in before transitioning to another state B. Upon re-entering state A, we can start with the remembered substate rather than A's initial state. Thus, the transition leaving A is treated much like an interrupt and B as an interrupt service routine.

5.2 Concurrent process model

In a concurrent process model, we describe system behavior as a set of processes, which communicate with one another. A process refers to a repeating sequential program. While manyembedded systems are most easily thought of as one process, other systems are more easily thought of as having multiple processes running concurrently.

For example, consider the following made-up system. The system allows a user to providetwo numbers X and Y. We then want to write "Hello World" to a display every X seconds, and "How are you" to the display every Y seconds. A very simple way to

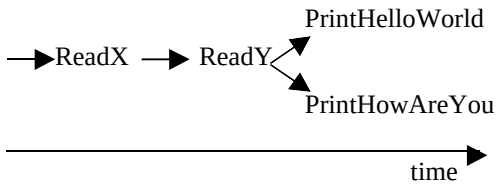
Figure 5.6: A simple concurrent process example: (a) pseudo-code, (b) subroutine execution over time, (c) sample input and output.

```
ConcurrentProcessExample()
  x = ReadX()
  y = ReadY()
  Call concurrently:
    PrintHelloWorld(x) and
    PrintHowAreYou(y)

PrintHelloWorld(x)
  while (1) {
    print "Hello world."
    delay(x);
  }

PrintHowAreYou(x)
  while (1) {
    print "How are you?"
    delay(y);
  }
```

(a)



(b)

Enter X: *1*
Enter Y: *2*
Hello world. *(Time = 1 s)*
Hello world. *(Time = 2 s)* How are you? *(Time = 2 s)* Hello world. *(Time = 3 s)* How are you? *(Time = 4 s)* Hello world. *(Time = 4 s)*
...

(c)

describe this system using concurrent processes is shown in Figure 5.6(a). After reading in X and Y, we call two subroutines concurrently. One subroutine prints "Hello World" every X seconds, the other prints "How are you" every Y seconds. (Note that you can't call two subroutines concurrently in a pure sequential program model, such as the model supported by the basic version of the C language). As shown in Figure 5.6(b), these two subroutines execute simultaneously. Sample output for X=1 and Y=2 is shown in Figure 5.7(c).

To see why concurrent processes were helpful, try describing the same system using a sequential program model (i.e., one process). You'll find yourself exerting effort figuring out how to schedule the two subroutines into one sequential program. Since this example is a trivial one, this extra effort is not a serious problem, but for a complex system, this extra effort can besignificant and can detract from the time you have to focus on the desired system behavior.

Recall that we described our elevator controller using two "blocks." Each block is really aprocess. The controller was simply easier to comprehend if we thought of the two blocks independently.

5.3.1 Concurrent Process

Two concurrent processes communicate using one of two techniques: message passing, or shared data. In the shared data technique, processes read and write variables that

both processes can access, called global variables. For example, in the elevator example above, the Request Resolver process writes to a variable req, which is also read by the Unit Control process.

In message passing, communication occurs using send and receive constructs that are part of the computation model. Specifically, a process P explicitly sends data to another processQ, which must explicitly receive the data. In the elevator example, Request Resolver would include a statement: Send (Unit Control, rr_req). Likewise, Unit Control would include statements of the form: Receive (Request Resolver, uc_req). rr_req and uc_req are variables local to each process.

Message passing may be blocking or non-blocking. In blocking message passing, a sending process must wait until the receiving process receives the data before executing the statement following the send. Thus, the processes synchronize at their send/receive points. In fact, a designer may use a send/receive with no actual message being passed, in order to achieve the synchronization. In non-blocking message passing, the sending process need not wait for the receive to occur before executing more statements. Therefore, a queue is implied in which the sent data must be stored before being received by the receiving process.

5.3.2 Communication among process

- Processes need to communicate data and signals to solve their computation problem
  - Processes that don't communicate are just independent programs solving separate problems
- Basic example: producer/consumer
  - Process A produces data items, Process B consumes them

E.g., A decodes video packets, B display decoded packets on a screen

5.3.3 Synchronization among process

- Sometimes concurrently running processes must synchronize their execution
  - When a process must wait for:
    - another process to compute some value
    - reach a known point in their execution
    - signal some condition
- Recall producer-consumer problem
  - Process A must wait if buffer is full
  - Process B must wait if buffer is empty
  - This is called busy-waiting
    - Process executing loops instead of being blocked
    - CPU time wasted

- More efficient methods
  - Join operation, and blocking send and receive discussed earlier
    - Both block the process so it doesn't waste CPU time
  - Condition variables and monitors

5.3.4 Implementing concurrent processes

The most straightforward method for implementing concurrent processes on processorsis to implement each process on its own processor. This method is common when each process is to be implemented using a single-purpose processor.

However, we often decide that several processes should be implemented using general-purpose processors. While we could conceptually use one general-purpose processor per process, this would likely be very expensive and in most cases is not necessary. It is not necessary because the processes likely do not require 100% of the processor's processing time; instead, many processes may share a single processor's time and still execute at the necessary rates.

One method for sharing a processor among multiple processes is to *manually rewrite* the processes as a single sequential program. For example, consider our Hello World program from earlier. We could rewrite the concurrent process model as a sequential one by replacing the concurrent running of the Print HelloWorld and Print How Are You routines by the following:

```
I = 1;
T = 0;
while (1) {
  Delay(I); T = T + I
  if X modulo T is 0 then callPrintHelloWorldif Y modulo T is 0 then call
  PrintHowAreYou
}
```

We would also modify each routine to have no parameter, no loop and no delay; each would merely print its message. If we wanted to reduce iterations, we could set I to the greatest common divisor of X and Y rather than to 1.

Manually rewriting a model may be practical for simple examples, but extremely difficult for more complex examples. While some automated techniques have evolved to assist with such rewriting of concurrent processes into a sequential program, these techniquesare not very commonly used.

Instead, a second, far more common method for sharing a processor among multiple processes is to rely on a multi-tasking operating system. An operating system is a low- levelprogram that runs on a processor, responsible for scheduling processes, allocating storage, and interfacing

to peripherals, among other things. A real-time operating system (RTOS) isan operating system that allows one to specify constraints on the rate of processes, and that guarantees that these rate constraints will be met. In such an approach, we would describe our concurrent processes using either a language with processes built- in (such as Ada or Java), or a sequential program language (like C or C++) using a library of routines that extends the language to support concurrent processes. POSIX threads were developed for the latter purpose.

A third method for sharing a processor among multiple processes is to convert the processes to a sequential program that includes a process scheduler right in the code. This method results in less overhead since it does not rely on an operating system, but also yieldscode that may be harder to maintain.

5.5 Dataflow Model

In a dataflow model, we describe system behavior as a set of nodes representing transformations, and a set of directed edges representing the flow of data from one node to another. Each node consumes data from its input edges, performs its transformation, and produces data on its output edge. All nodes may execute concurrently.

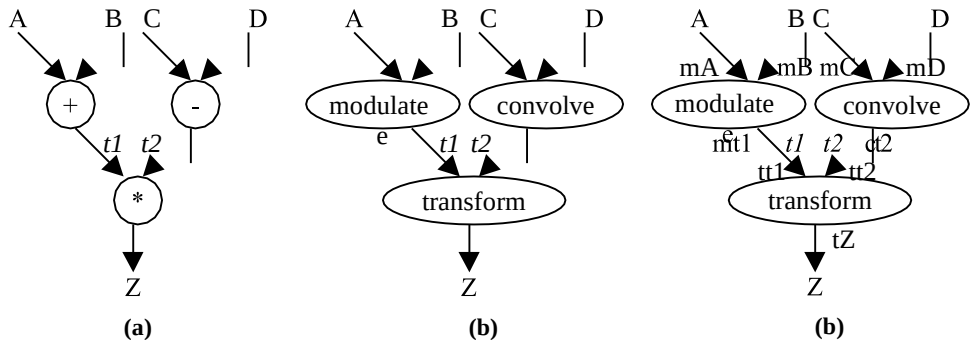
For example, Figure 5.7(a) shows a dataflow model of the computation Z = (A+B)*(C-D). Figure 5.7(b) shows another dataflow model having more complex node transformations. Each edge may or not have data. Data present on an edge is called a token.When all input edges to a node have at least on token, the node may fire. When a node fires,it consumes one token from each input edge, executes its data transformation on the consumed token, and generates a token on its output edge. Note that multiple nodes may fire simultaneously, depending only on the presence of tokens.

Several commercial tools support graphical languages for the capture of dataflowmodels. These tools can automatically translate the model to a concurrent process model for implementation on a microprocessor. We can translate a dataflow model to a concurrent process model by converting each node to a process, and each edge to a channel. This concurrent process model can be implemented either by using a real-time operating system, or by mapping the concurrent processes to a sequential program.

Lee observed that in many digital signal-processing systems, data flows in to and out ofthe system at a fixed rate, and that a node may consume and produce many tokens per firing.He therefore created a variation of dataflow called synchronous dataflow. In this model, we

annotate each input and output edge of a node with the number of tokens that node consumes and produces, respectively, during one firing. The advantage of this model is that, rather than translating to a concurrent process model for implementation, we can instead statically schedule the nodes to produce a sequential program model. This model can be captured in a sequential program language like C, thus running without a real-time operating system and hence executing more efficiently. Much effort has gone into developing algorithms for scheduling the nodes into "single-appearance" schedules, in which the C code only has one statement that calls each node's associated procedure (though this call may be in a loop). Such a schedule allows for procedure in lining, which further improves performance by reducing the overhead of procedure calls, without resulting in an explosion of code size thatwould have occurred had there been many statements that called each node's procedure.

Figure 5.7: Simple dataflow models: (a) nodes representing arithmetic transformations, (b) nodes representing more complex transformations, (c) synchronous dataflow.



5.6 Design Technology

- Design technologies developed to improve productivity
- Focus on technologies advancing hardware/software unified view
- Automation
- Program replaces manual design
- Synthesis
- Reuse
- Predesigned components
- Cores
- General-purpose and single-purpose processors on single IC

–Verification

- Ensuring correctness/completeness of each design step
- Hardware/software co-simulation

5.7 Automation: synthesis

- Early design mostly hardware

- Software complexity increased with advent of general-purpose processor
- Different techniques for software design and hardware design
  - –Caused division of the two fields
- Design tools evolve for higher levels of abstraction
  - –Different rate in each field
- Hardware/software design fields rejoining
  - –Both can start from behavioral description in sequential program model
  - –30 years longer for hardware design to reach this step in the ladder
    - Many more design dimensions
    - Optimization critical

Hardware/software parallel evolution

- Software design evolution
  - –Machine instructions
  - –Assemblers
    - convert assembly programs into machine instructions
  - –Compilers
    - translate sequential programs into assembly
- Hardware design evolution
  - –Interconnected logic gates
  - –Logic synthesis
    - converts logic equations or FSMs into gates
  - –Register-transfer (RT) synthesis
    - converts FSMDs into FSMs, logic equations, predesigned RT components (registers, adders, etc.)
  - –Behavioral synthesis
    - converts sequential programs into FSMDs

5.8 Hardware/software co-simulation

- Variety of simulation approaches exist
  - –From very detailed
    - E.g., gate-level model
  - –To very abstract
    - E.g., instruction-level model
- Simulation tools evolved separately for hardware/software
  - –Recall separate design evolution
  - –Software (GPP)
    - Typically with instruction-set simulator (ISS)

–Hardware (SPP)

• Typically with models in HDL environment

• Integration of GPP/SPP on single IC creating need for merging simulation tools

Integrating GPP/SPP simulations

• Simple/naïve way

–HDL model of microprocessor

• Runs system software

• Much slower than ISS

• Less observable/controllable than ISS

–HDL models of SPPs

–Integrate all models

Hardware-software co-simulator

–ISS for microprocessor

–HDL model for SPPs

–Create communication between simulators

–Simulators run separately except when transferring data

–Faster

–Though, frequent communication between ISS and HDL model slows it down

Advantages/disadvantages of soft/firm cores

• Soft cores

–Can be synthesized to nearly any technology

–Can optimize for particular use

• E.g., delete unused portion of core

–Lower power, smaller designs

–Requires more design effort

–May not work in technology not tested for

–Not as optimized as hard core for same processor

• Firm cores

–Compromise between hard and soft cores

• Some retargetability

• Limited optimization

• Better predictability/ease of use

New challenges to processor providers

- Cores have dramatically changed business model
  - –Pricing models
    - Past
      - –Vendors sold product as IC to designers
      - –Designers must buy any additional copies
        - Could not (economically) copy from original
    - Today
      - –Vendors can sell as IP
      - –Designers can make as many copies as needed
    - Vendor can use different pricing models
      - –Royalty-based model
        - Similar to old IC model
        - Designer pays for each additional model
      - –Fixed price model
        - One price for IP and as many copies as needed
      - –Many other models used

5.9 IP core (intellectual property core)

An IP (intellectual property) core is a block of logic or data that is used in making a field programmable gate array (FPGA) or application-specific integrated circuit (ASIC) for a product. As essential elements of design reuse , IP cores are part of the growing electronic design automation (EDA) industry trend towards repeated use of previously designed components. Ideally, an IP core should be entirely portable - that is, able to easily be inserted into any vendor technology or design methodology. Universal Asynchronous Receiver/Transmitter (UART s), central processing units (CPU s), Ethernet controllers, and PCI interfaces are all examples of IP cores.

IP cores fall into one of three categories: hard cores , firm cores , or soft cores . Hard cores are physical manifestations of the IP design. These are best for plug-and-play applications, and are less portable and flexible than the other two types of cores. Like the hard cores, firm (sometimes called semi-hard) cores also carry placement data but are configurable to various applications. The most flexible of the three, soft cores exist either as a netlist (a list of the logic gate s and

associated interconnections making up an integrated circuit) or hardware description language(HDL) code.

5.10 Design process model

- Describes order that design steps are processed
  - –Behavior description step
  - –Behavior to structure conversion step
  - –Mapping structure to physical implementation step
- Waterfall model
  - –Proceed to next step only after current step completed
- Spiral model
  - –Proceed through 3 steps in order but with less detail
  - –Repeat 3 steps gradually increasing detail
  - –Keep repeating until desired system obtained
  - –Becoming extremely popular (hardware & software development)

Part A

1.Define Finite State Machine model?
2.What is hardware-software co design in embedded system?
3.What are the fundamental issues in hardware and software co design in an embedded system?
4.Interpret about data flow models?
5.What is synthesis in embedded systems?
6.Interpret embedded automation?
What is design technology in embedded system?
List out the functions of IP core in embedded system?
Compare SoC and IP?
10.What are the operations defined by the concurrent process?

Part B

1.Build the steps involved in describing a system's behavior as a state Machine.

2.Elaborate a note on concurrent process model and communication among the process.
3.Explain the Synchronization among process with examples

4.Explain in brief about the following. A) FSM B) Data flow model.

TEXT/ REFEENCE BOOKS

1.David E.Simon, "An Embedded Software Primer", Pearson Education,2001
2. Frank Vahid and Tony Gwargie, "Embedded System Design", John Wiley & Sons,2002
3. Steve Heath, "Embedded System Design", Elsevier, Second Edition,2004.