Embedded System

What is embedded system?

An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is designed for a specific function or within a larger system. Industrial machines, agricultural and process industry devices, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines and toys as well as mobile devices are all possible locations for an embedded system.

An embedded system has three components –

- It has hardware.
- It has application software.
- It has Real Time Operating system (RTOS)

Advantages

- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

Disadvantages

- High development effort
- Larger time to market

Characteristics of an Embedded System

- **Single-functioned** − An embedded system usually performs a specialized operation and does the same repeatedly.
- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay.
- **Microprocessors based** − It must be microprocessor or microcontroller based.
- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- **Connected** − It must have connected peripherals to connect input and output devices.
- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.

Types of Embedded system:

Embedded system can be classified into different types based on performance function requirement and performance of the microcontroller.

- Real time embedded system :

  An embedded system that gives an output within a specified amount of real time is called real time embedded system. That is in additional to a proper output it depends on time constant as well.

- Stand Alone embedded system:

  These are embedded system that can work by themselves. In other words they are self sufficient and don't required a different system to function. Music players, video gaming and microwave ovens are the examples of standalon system.

- Network embedded system:

  Embedded system that are connected to the network & depends on it from their functionally are called network embedded system. Router

  home security system are network embedded system.

- Mobile embedded system

  Embedded system that are mobile in nature are called mobile embedded system. They include generally small system. They are used in

  portable embedded devices like cell phones, mobile, digital camers, mp3 player and personal digital assistant.

Apllication of embedded system:

Here are some application area of embedded system:

- Consumer electronics:

  Mobile phones , video games, printers, home entertainment system, television, digital camera, music player
- Household application:

  Washing machine , diswashes, air conditioner, etc
- Medical equipment:

  BLood pressure monitor, CET scanner, heart beat monitor, ECG machine, etc

- Automobiles

  Anticlock breaking system, air conditionor control, electronic fuel, injections, entertainment systems

- Industrial system

  Robotic machine, control system, smoke detector, fire detector, data collection system, system monitoring, feedback system

- Aerospace

  Navigation system, guided system, global positioning system (GPS)

- Communication

  Router, network hubs, phone, etc

Some Basic terms:

Chips :

Most digital electronic circuits today are built with semi conductor parts called chips. Chips are purchased from manufacturers specializing in building such parts.

Semi-conductor themselves are enclosed in small thin square or rectangle black package made of plastics or ceramics. Each package made of plastics or ceramics. Each package have collection of pins and those pins are response for their own functions. Most digital circuit use just two voltage to do their work

- OVDC somtimes called ground or low input.
- Either 3V or 5V sometimes called VCC or high

The most common mechanism to connect the chips to one another is the printed circuit board.

Telegraph

Telegraph : is one of the example of embedded system that do the long-distance transmission of textual or symbolic (as opposed to verbal or audio) messages without the physical exchange of an object bearing the message. Telegraphy requires that the method used for encoding the message be known to both sender and receiver. Such methods are designed according to the limits of the signaling medium used.

- Telegraph must sort out on the network and provide the clean data stream to the printer.
- Telegraph has to work with a number of different types of a printers without customer configuration.
- Telegraph must respond quite rapidly to certain events.
- Telegraph must keep track of time

Gates :

A very simple part built from a handful of semiconductor transistor is called gates or sometime discrete.

or

" A logic gate is an elementary building block of a digital circuit. Most logic gates have two inputs and one output.

At any given moment , every terminal is in one of two binary conditions low (0) or high(1).

# Types of Gates:

And gate:

An AND gate is one whose output is driven to high of both the input are high and in other conditions the output is driven to low

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR gate:

An OR gate is one which output is high if either or both of the inputs are high. AND whose output is driven to low if only if both input are low.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT/ INVERTER gate:

Inverter are the very simple gate i.e output driven to low if the input is high & vice- versa

INPUT	OUTPUT
1	0
0	1

Bubble:

The bubble or little loop on the invertor symbol is used in other schematic symbols to indicate that an input or an output is inverted that is loop when it should be high & vice versa

A	Y
0	0
1	1

X-OR gate:

AN XOR gate is one whose output is driven high if one if the input is high & other input is low. And if both the input are high or low at that conditions the output is low, then it is called XOR gate.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NAND gate:

NAND gate is not AND gate.

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Few other basic consideration

Cordless Bar-code scanner

A cordless **barcode reader** (or **barcode scanner**) is an embedded system (electronic device) that can read and output printed barcodes to a computer. Like a flatbed scanner, it consists of a light source, a lens and a light sensor translating optical impulses into electrical ones. Additionally, nearly all barcode readers contain *decoder* circuitry analyzing the barcode's image data provided by the sensor and sending the barcode's content to the scanner's output port.

Laser Printer

Laser Printer: Another embedded system is a laser printer. Most laser printers have fairly substantial microprocessor embedded in them to control all aspects of the printing. In particular the microprocessor is responsible for getting the data from various communication ports on the printer for sending when the user presses the certain buttons from the control pannel and for presenting the messages to the users on the control panel display.

Under ground tank monitor system

The under ground tank monitor system is one kind of embedded system that watches the levels of gasoline in the underground tank at a gas station. It’s principle purpose is to detect the leakage before the gas station turns into a toxic waste dump. The system also has a panel of 16 buttons and a 20 character liquid crystal display and a thermal printer. With the help of buttons the user can tell the system to display or print various information such as gasoline level in tank or the overall system status.

Nuclear reactor monitor

Nuclear reactor monitor: is an another example of embedded system that monitors the what is happening inside the nuclear reactor centre where huge amount of nuclear reactions are occurred. In nuclear reactor monitor, there is hypothetical system that controls the nuclear reactor. The embedded code that monitors the two temperature which are always supposed to be equal. If they differ it indicates that there is some defects in the system.

Power and Decoupling

A **power supply** is an electronic device that supplies electric energy to an electrical load. The primary function of a power supply is to convert one form of electrical energy to another and, as a result, power supplies are sometimes referred to as electric power converters.

A **decoupling capacitor** is a capacitor used to decouple one part of an electrical network (circuit) from another. Noise caused by other circuit elements is shunted through the capacitor, reducing the effect it has on the rest of the circuit. An alternative name is **bypass capacitor** as it is used to bypass the power supply or other high impedance component of a circuit.

- A decoupling capacitor can prevent the powered circuit from seeing that signal, thus decoupling it from that aspect of the power supply circuit.
- Another kind of decoupling is stopping a portion of a circuit from being affected by switching that occurs in another portion of the circuit. Switching in sub circuit A may cause fluctuations in the power supply.

Timing Diagrams

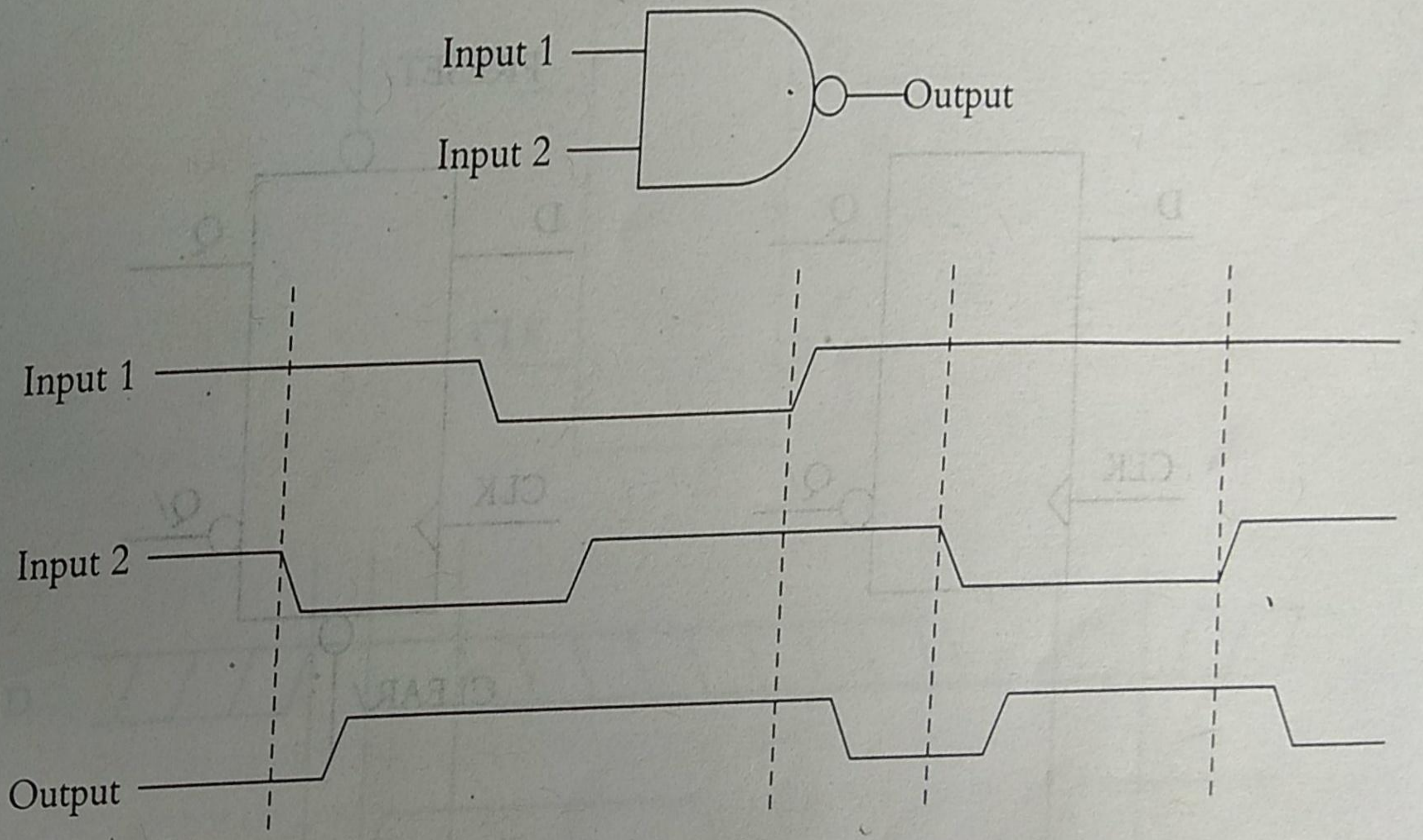
Timing is essential to the success of a microcomputer design. Often it is quite possible to get one system functioning by simply interconnecting the various compo-nents. But it is significantly more difficult to be able to guarantee that many systems will work under the entire range of possible conditions that they may be exposed to.

There are many designs in production right now that have a number of unidentified failures due to the lack of a worst-case analysis of the design. When timing or loading problems show up in a design, they usually appear as intermittent failures or as sensitivity to power supply fluctuations, tempera-ture changes, and so on.

A timing diagram is a graph that shows the passage of time on the horizontal axis and show each of the input and output signals changing and the relationship of the changes to one another. For examples NAND gates are simple that manufacture normally published a time diagram. In the figure you can see that whenever one of the inputs goes low, the output goes high.

Fig:

Figure 2.20 A Simple Timing Diagram for a NAND Gate

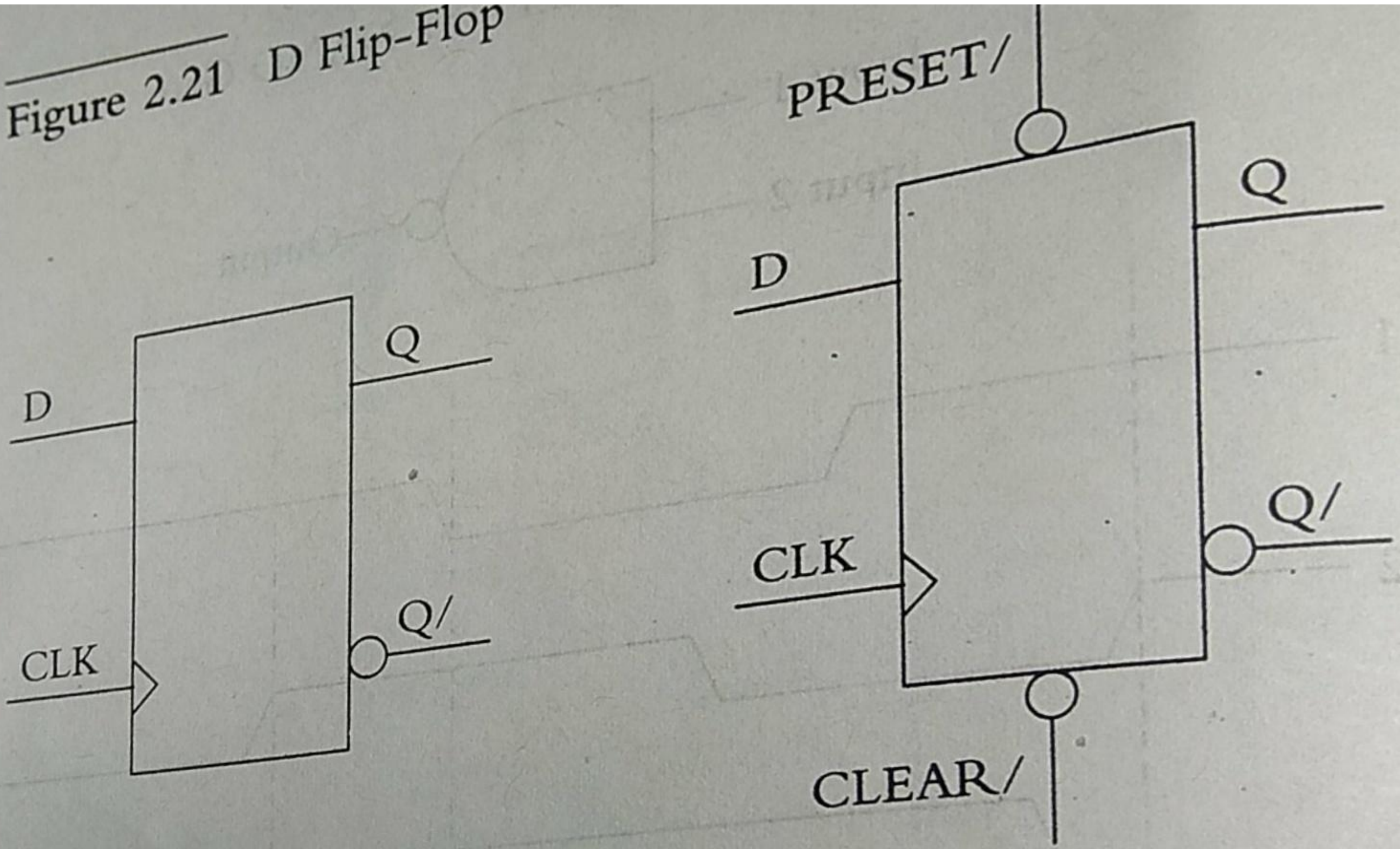


D Flip-Flops

A D **flip-flop** is a circuit that has two stable states and can be used to store state information. A D flip flop is also known as a register, D-flop or a flip-flop or even just a flop. A D flip-flop is essentially a 1-bit memory. The transition of a signal from low to high is called a rising edge. The transition of a signal from high to low is called a falling edge.

The Q output on the D flip-flop takes on the value of the D input at the time that the CLK input transition from low to high, that is, at the CLK signal's rising edge. Then the Q output holds that value until the CLK is driven low again and then high again. Some D flip-flops also have a CLEAR/ signal and a PRESET/ signal. On those parts, asserting the CLREAR/ signal forces the Q signal low, no matter what the CLK and D signals are doing; asserting the PRESET/ signal forces the Q signal high.

Figure 2.21 D Flip-Flop



Clock

Microprocessor based circuit must go on executing instruction even if nothing changes in the outside. To accomplish this, most circuits have a signal called the clock. The purpose of the clock signal is to provide rising and falling edges to make other parts of the circuit do their jobs. The two types of parts used to generate clock signals are **oscillators** and **crystal.**

- Oscillators is a part that generates a clock signal all by itself. Oscillators typically come in metallic packages with four pins: one for VCC, one for ground, one that the clock signal, and one that is there just to make easier to solder.
- A crystal has just two signal connections, and you must build a little circuit around it to get a clock signal out. Many microprocessors have two pins on them for attachment to a circuit containing a crystal.

Memory

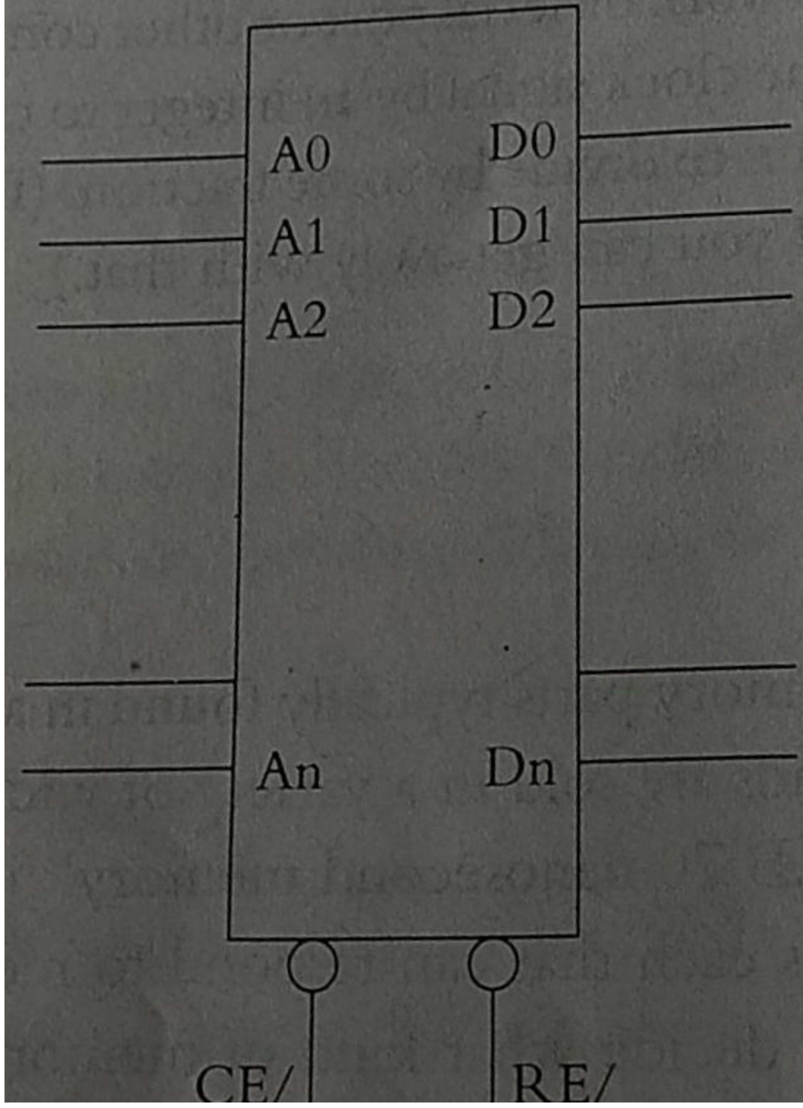
ROM: Almost every computer system needs a memory area in which to store the instruction of its program. This must be a non-volatile memory, that is, one that doesn't forget its data when the power is turned off. In most embedded systems, which do not have a disk drive or other storage medium, the entire program must be in memory that is called ROM.

The characteristics of ROM are the following:

- `The microprocessor can read the program instructions from the ROM quickly.
- The microprocessor can not write new data to the ROM.
- The ROM remembers the data, even if the power is turned off.

Required figure

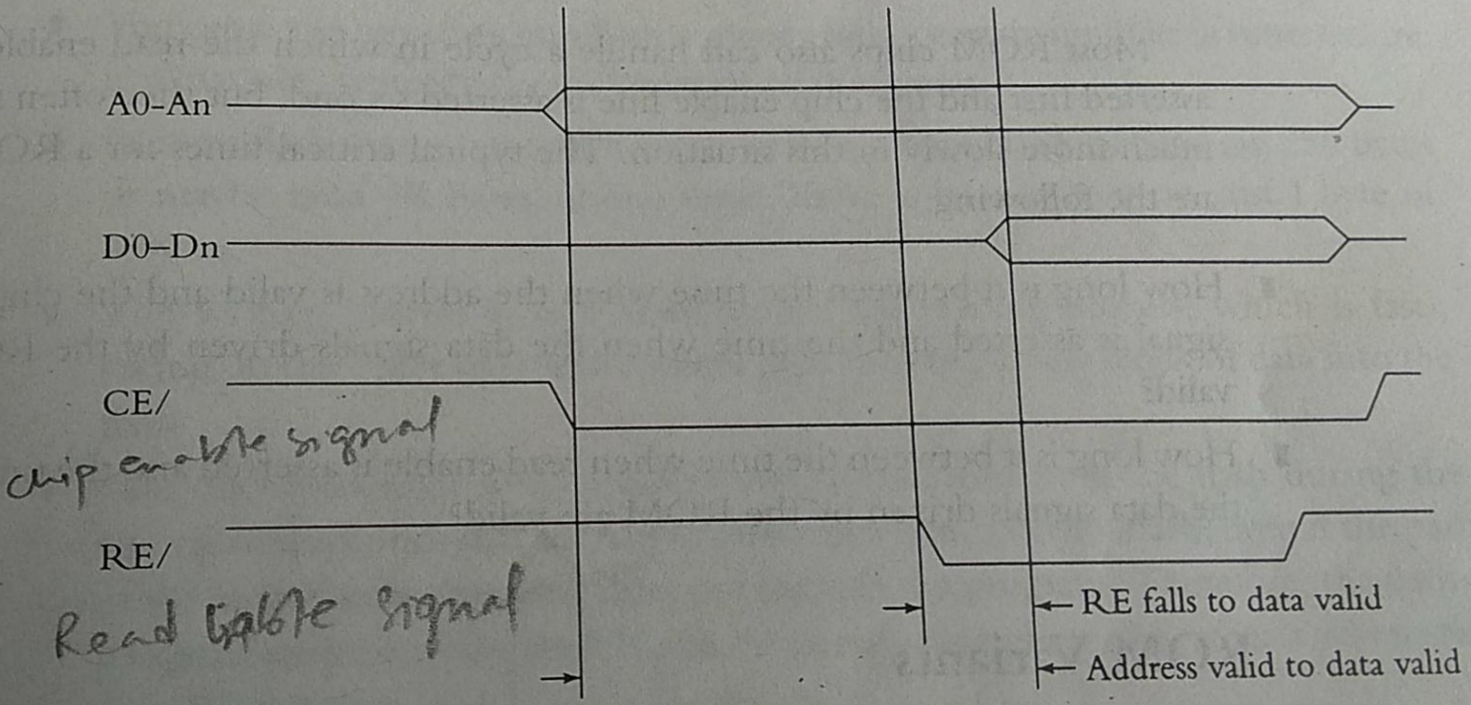
Figure 2.24 Typical ROM Chip Schematic Symbol



In the figure, the signal from A0 to An are the address signal, which indicate the address from which the processor wants to read. The signal from D0 to Dn are the data signals driven by the ROM. The CE/ signal is the chip enable signal, which tells the ROM that the microprocessor wants to activate the ROM. The RE/ signal is the read enable signal, which indicates that the ROM should drive its data on the D0 to Dn signals.

Timing Diagram for ROM

Figure 2.25 Timing Diagram for a Typical ROM



Above figure shows the timing diagram for ROM with parts such as memory chips, which have multiple address or data signals, it is common to show a group of such related signals on a single row of the timing diagram. With such a group of signals, a single line that is neither high nor low indicates that the signals are floating or changing. When the signals take on a particular value, that is shown in the timing diagram with two lines, one high and one low, to indicate that each of the signals has been driven either high or low.

The sequence of events when a microprocessor reads from a ROM is as follows:

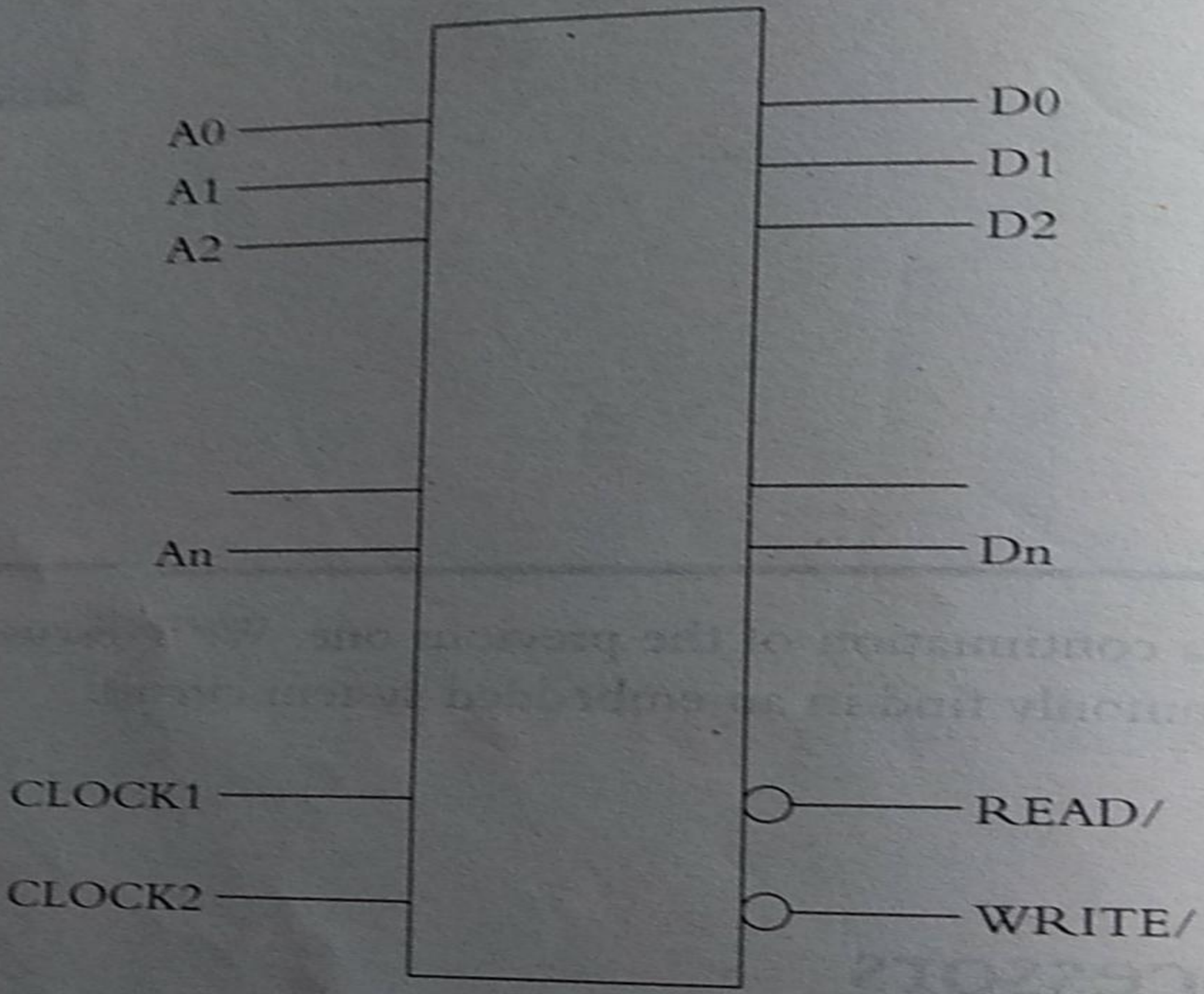
- The microprocessor drives the address lines with the address of the location it wants to fetch from the ROM.
- At about the same time, the chip enable signal is asserted.
- A little while later the microprocessor asserts the read line.
- The ROM drives the data onto the data lines for the microprocessor to read
- When the microprocessor has been seen the data on the data lines it releases the chip enables and read enable lines.

Microprocessor

A **microprocessor** is a system processor which incorporates the functions of a system's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based, programmable electronic device which accepts digital or binary data as input, processes it according to instructions stored in its memory, and provides results as output.

A collection of address signal is used to tell the various other parts of the circuit memory. For example:- the addresses it want to read from or write to. A collection of data signals, it uses to get the data from and send data to another parts in the circuit.

Figure 3.1 A Very Basic Microprocessor



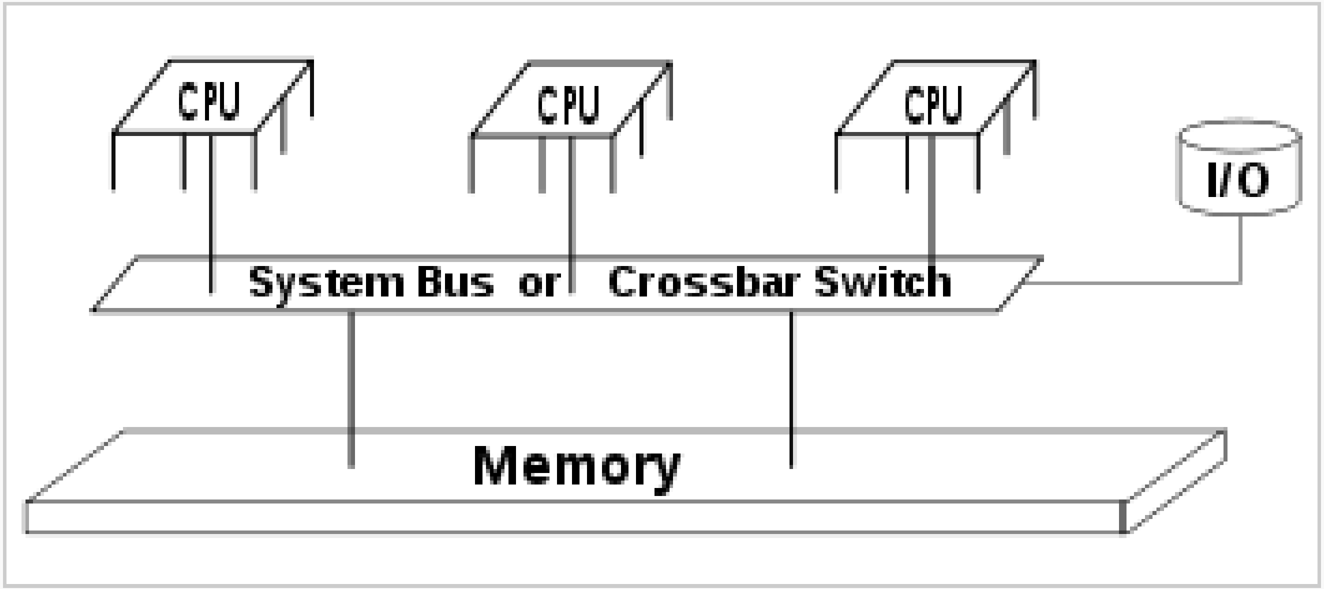
A READ/ line which it pulses low when it wants to get data and a WRITE/ line which it pulses low when it wants to write data out. A clock signal input which paces(speed) all of the work that the microprocessor does and as a consequence pace the work in the rest of the system. Some microprocessor have two clocks inputs to allow the designer to attached the crystal circuit.

Buses

A **system bus** is a single system bus that connects the major components of a system, combining the functions of a data bus to carry information, an address bus to determine where it should be sent, and a control bus to determine its operation.

The address, data and control buses are used for data signals, address signals and control on the microprocessor. The READ/ signal from the microprocessor is connected to the output enable signals on the memory chips.

The write/ signal for microprocessor is connected to the write enable signals on the RAM. Some types of the clock circuit is attached to the clock signals on the microprocessor. The address signals as a group are very often refers to as address bus. Similarly the data signals are refers to as data bus. The combination of two bus the READ & WRITE signals from the processors are referred to as microprocessor bus.



Simple symmetric multiprocessing using a system bus

Bus handshaking

The RAM & ROM will have various timing requirements. The address line must stay stable for a certain period of time and read enable & chip enable lines must be asserted for some period of time then the data will be valid on the bus. The microprocessor is in control of all these signals and it decides when to took for data on the bus. The entire process is called bus cycle. For the circuit to work the signals that the microprocessor produces must confirm to the requirement of the other parts in the circuit. The various mechanism by which this can be accomplish are referred to as bus handshaking.

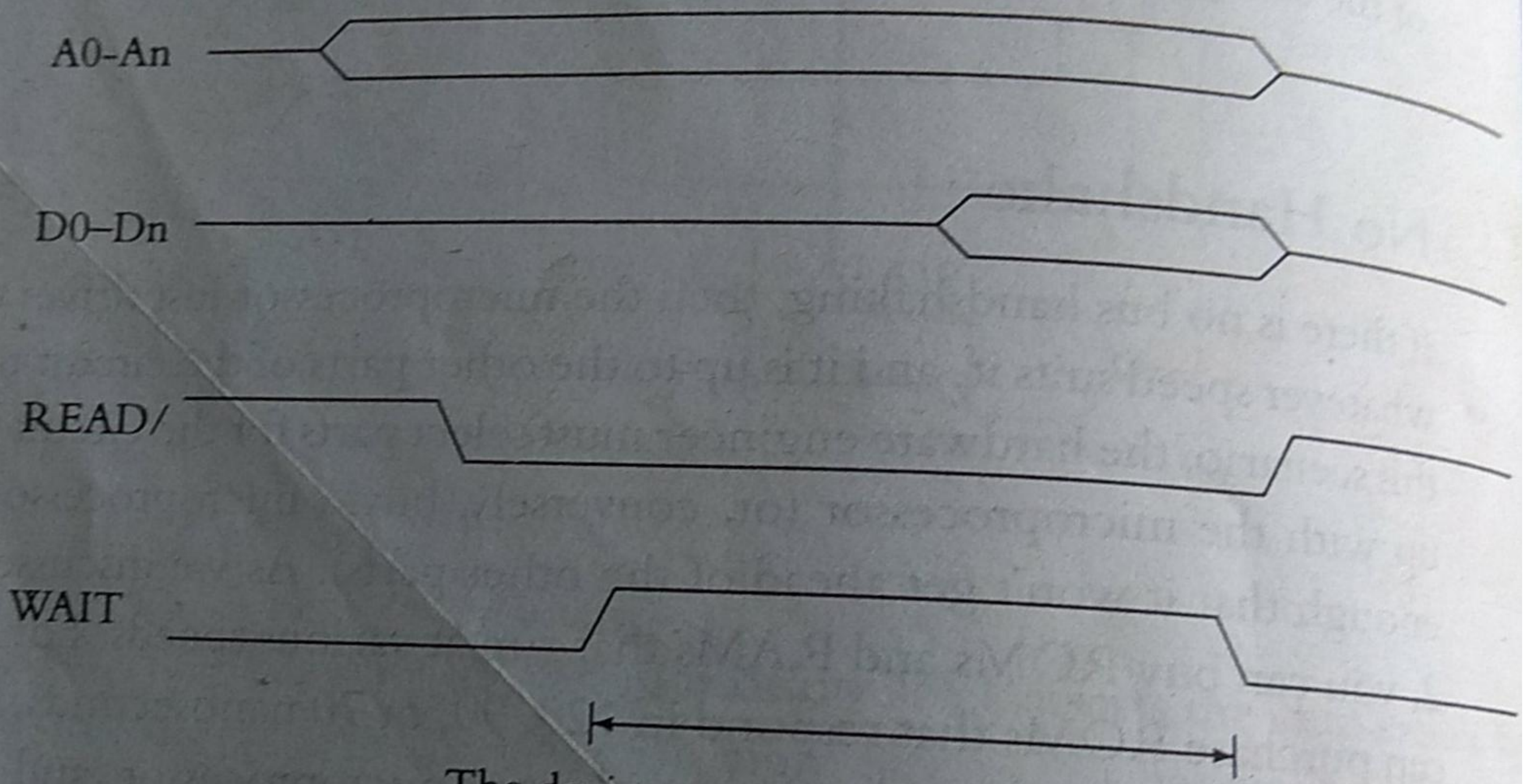
No Bus Handshaking

If there is no bus handshaking then the microprocessor just derive the signals at whatever speed suits it and it is up to the other parts of the circuit to keep up. In this scenario the hardware engineer parts for the circuit that can keep up with the microprocessor . For example we can purchase the ROM’s that respond in 120, 90, 70 nanoseconds depending on how fast they must be keep up with our microprocessor.

Wait signals

- Some microprocessor offers the alternative that they have a wait input signal that the memory can use to extend the bus cycle as needed. The figure shows the microprocessor at normal state and the microprocessor in wait state. As long as wait signal is asserted the microprocessor will wait in definitely for the device to put the data on the bus. The only one disadvantage of wait signal is that ROM’s and RAM’s do not come from the manufacturers with a wait signal correctly

Bus cycle extended by asserting the WAIT signal

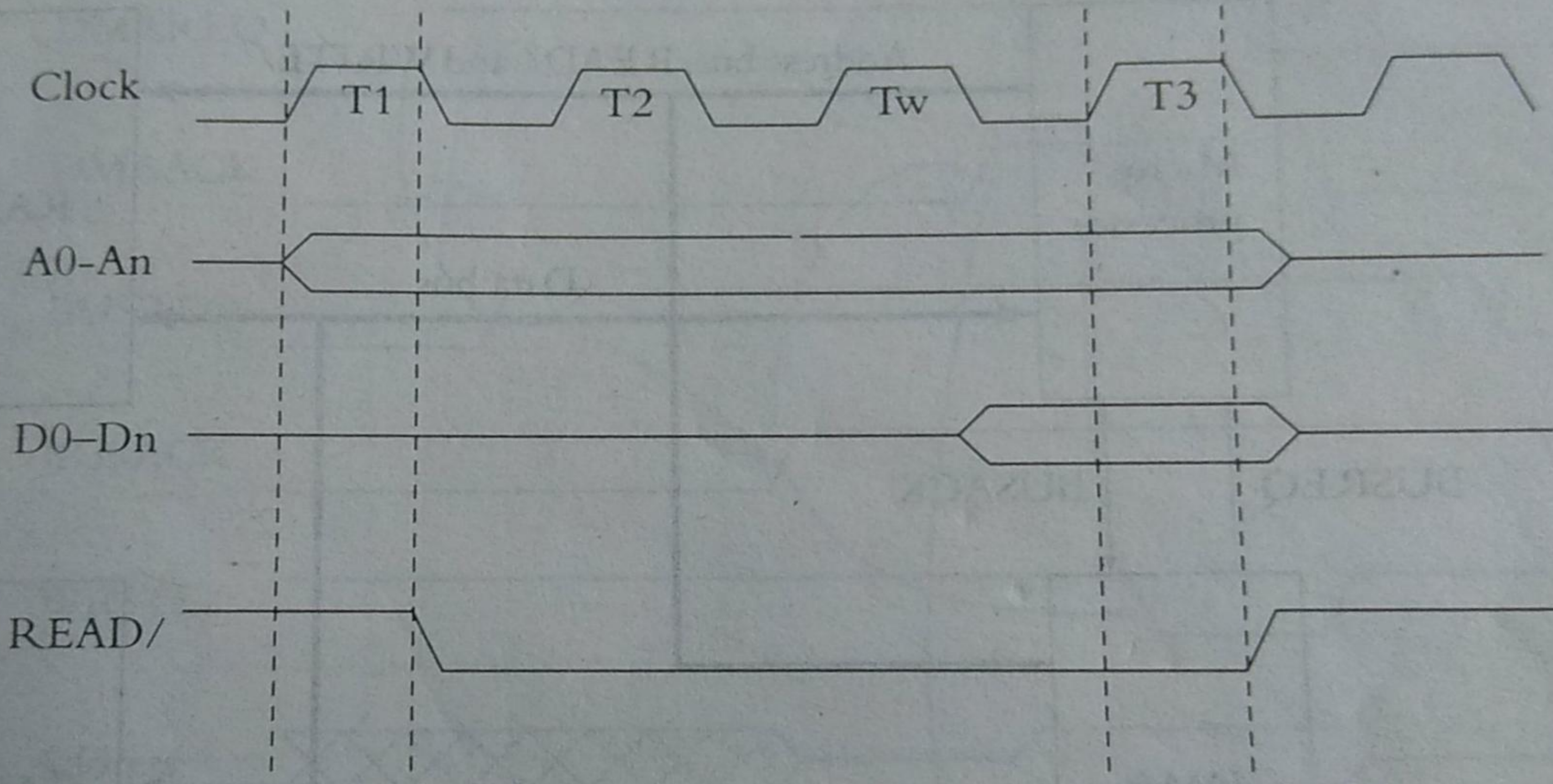


The device can assert the WAIT signal as long as it needs to, and the microprocessor will wait.

Wait State

Some microprocessor offer the alternative for dealing the slow memory devices i.e. wait state. For understanding the wait state we need to first understand how the microprocessor times the signals on the bus in the first place. The microprocessor have clock inputs, address lines, data lines & it uses the clock time for all its activities in a particular interaction with the bus.

Figure 3.7 The Microprocessor Adds a Wait State



Microprocessor drives address bus to start the bus cycle.

Microprocessor drives READ low.

Microprocessor reads the data from the bus.

End of the bus cycle

It outputs the address on the rising edge of T1 i.e. when the clock signal transmission from low to high in the first clock cycle of the bus cycle. It assert the READ/ line at falling edge of T_1. It expect the data to be valid and actually takes the data in just a little after the rising edge of T_3. It desserts the READ/ line at the falling edge of T_3 & shortly there after stop driving the address signals there by completing the transaction.

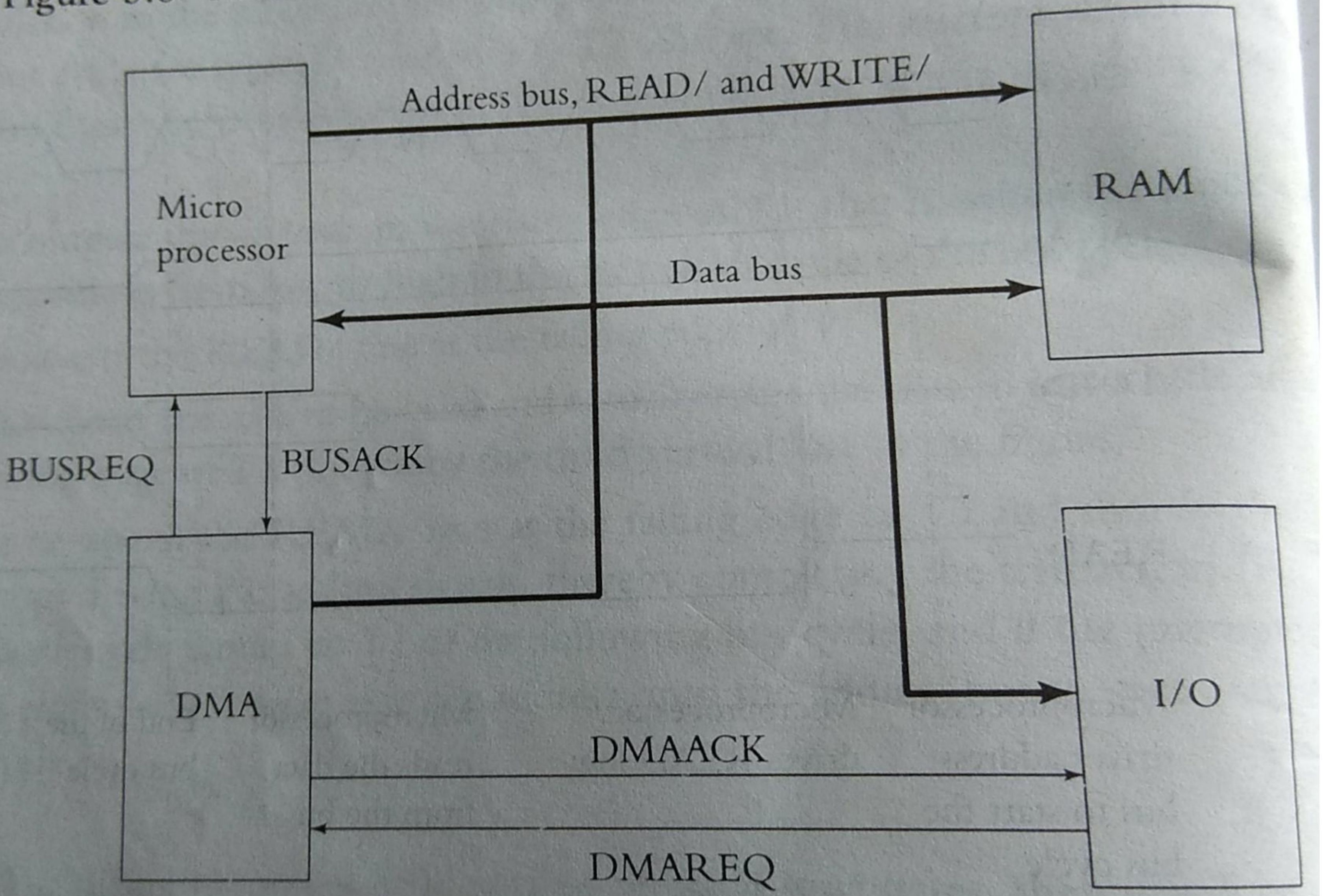
Direct memory access

Direct memory access (**DMA**) is a feature of systems that allows certain hardware subsystems to access main system memory (RAM) independently of the central processing unit(CPU).

One way to get the data into the output of the system quickly is to use of direct memory access. DMA is circuit that can read the data from the I/O devices such as serial port or a network and then write it into the memory or read from the memory and write to the I/O devices. When the I/O devices has data to be moved into the RAM, it asserts the DMAREQ signal to the DMA circuit.

The DMA circuit is turn asserts the BUSREQ signal to the microprocessor. When the microprocessor is ready to give up the bus-which may mean not executing instructions for the short period during which the DMA does it work-it asserts the BUSACK signal. The DMA circuitry then places the address into which the data is to be written on the address bus, asserts DMAACK back to the I/O device and asserts WRITE/ to the RAM. The I/O device puts the data on the data bus for the RAM, completing the write cycle. Fig:

Figure 3.8 Architecture of a System with DMA

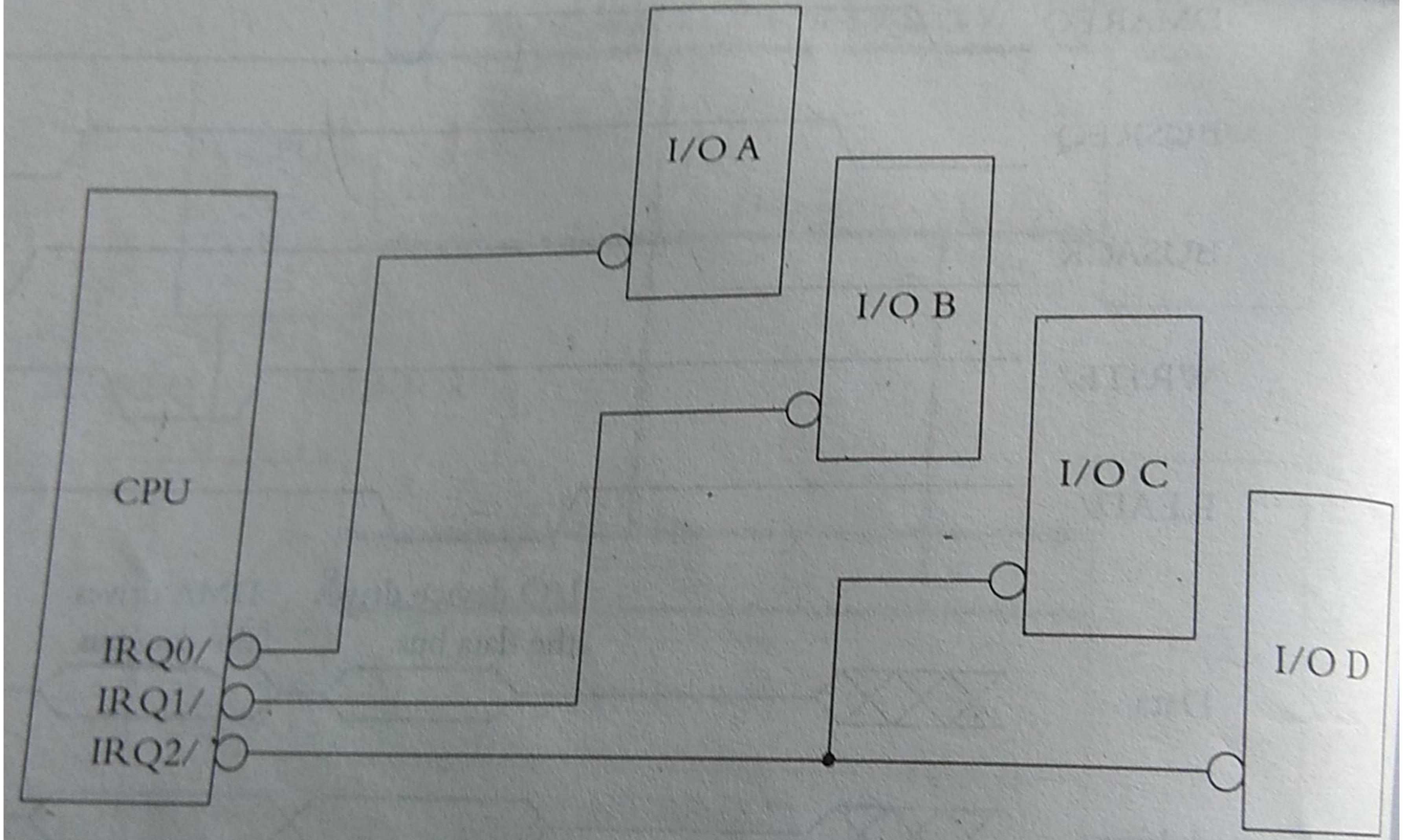


Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done.

Interrupts

Interrupts is the process of opposing or stopping the running execution files if there is some happen occurred. Interrupt told to stop doing what it is doing and execute some other pieces of software. The signals that tells the microprocessor i.e. it is time to run the interrupt is the interrupt request.

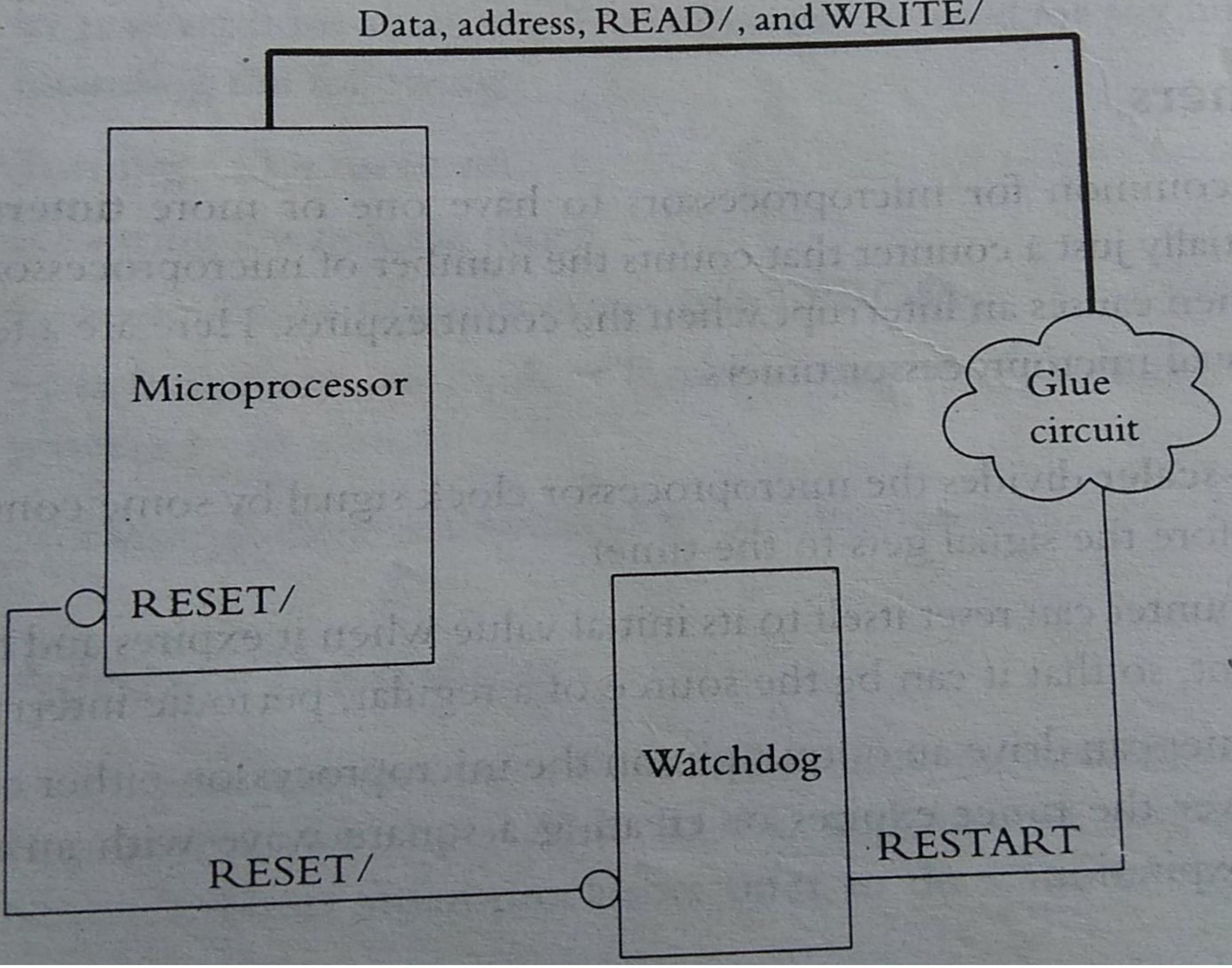
Figure 3.12 Interrupt Connections



Watchdog timer

One common part that use in embedded system circuit is use of watchdog timer. A watchdog timer contains a timer that expires after the certain interval unless it is started. The watchdog timer has an output that pulses should the timer ever expire. The way that the watchdog timer are connected into the circuit is shown in figure. The output of watchdog timer is attached to the RESET/ signal on the microprocessor. If the timer is expired the pulse on its output signal resets the microprocessor and start the software over the beginning. Different watchdogs circuits require different pattern of the signals on their input to restart them. Some glue circuitry may be necessary to allow the microprocessor to change the RESTART signal appropriately.

Figure 3.18 Typical Use of a Watchdog Timer



UART(my definition)

- A universal asynchronous receiver/transmitter (UART) is a microchip that performs serial-to-parallel conversion of data received from peripheral devices and parallel-to-serial conversion of data coming from the CPU for transmission to peripheral devices. The UART chip has control capabilities and the ability to send an interrupt request to the processor that can be tailored in a way that minimizes the software management of the communication link between a computer and a peripheral device.

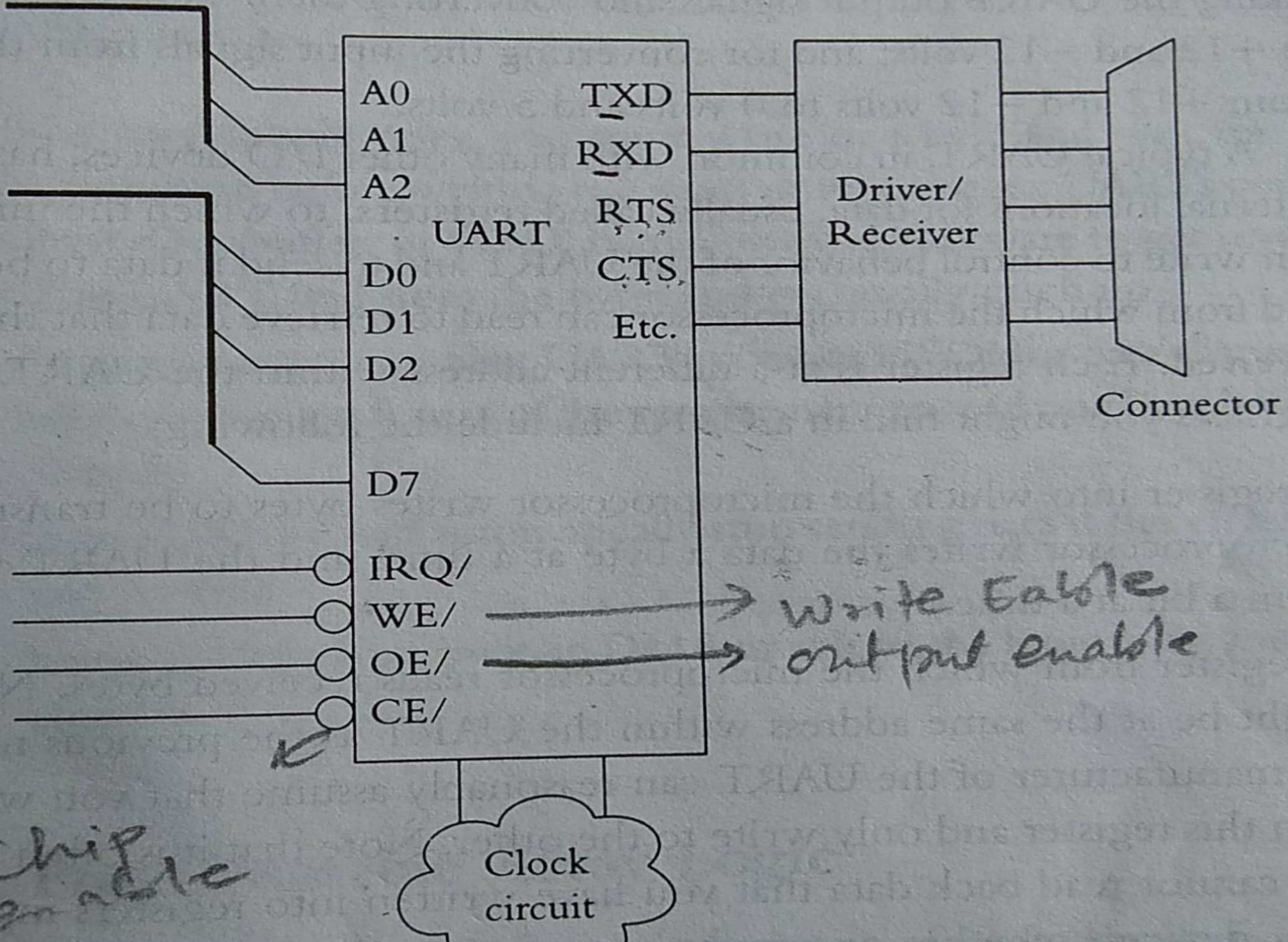
UART Functions

- Converts parallel data into serial data for outbound communications
- Converts serial data into parallel data for inbound communications
- Adds a parity checking bit on outbound transmissions and checks the parity bit for inbound transmissions
- Handles interrupt requests and device management, which may require the computer and the device to coordinate the speed of operation.

UART(Universal Asynchronous Receiver/transmitter and RS-232)

A **UART** is a common device i.e. used in many embedded systems. The main purpose of UART is to convert the data to and from a several interface i.e. an interface in which the bits that makeup the data send one after another. A very common standard for serial interface is RS-232 i.e. used between computer and modem as well as computer and mouse. A typical UART and its connection are shown in figure. On the left hand side of the UART are those signals that attach to the bus structure address line, data line, read and write line and interrupt line. At the bottom of the UART is the connection to a clock circuit.

Figure 3.13 A System with a UART



The clock circuit for the UART is separated from the microprocessor clock circuit because it must run at a frequency i.e. the multiple of common bit rate. The signal on the right are those that go to the serial port, a line for transmitting bits one after another(TXD), a line for receiving the bits(RXD), and some standard control line used in the RS-232 serial protocol. Generally UART usually runs at the standard 3 or 5 volt of the circuit.

Built-Ins on the microprocessor

1) Timer

It is common for microprocessors to have one or more timers. A timer is essentially a counter that counts the number of microprocessor clock cycle and causes an interrupt when the counter expires. The features of microprocessor counter are

- –A pre-scalar divides the microprocessor clock signal by some constant before the signal gets to the timer.
- –The counter can reset itself to its initial value when it expires and then continue to the count so that it can be the source of a regular period interrupt.

–The timer can drive an output pin on the microprocessor either causing the pulse whenever a timer expires or creating a square wave with an edge with every timer expiration.

–A timer has an input pin that enable or disable the counting, the timer circuit also may be able to function a counter that counts pulse on that input pin.

2) DMA:

Some DMA channels are used in built microprocessor chips. Since the DMA channel and the microprocessor contained for the bus and certain processes are simplify if the DMA channel and a microprocessor are on the same chips.

3)I/O Pins:

It is common for the microprocessor intended for embedded system to contain anywhere from a few dozen of I/O pins. These pin can be configured as a output that the software can set high or low directly usually by writing to a register or they can be configure as input that software can read. These pin can be used for any number of purpose including the following points

A) Turning on and off the led

B) Resetting the watchdogs timer

C) Reading from a one pin or two pin EEROM

D) Switching from one bank of ROM to another if there is more Ram than the processor can address.

4) Address decoding:

Some microprocessor offers to do some of that address decoding for you by having a handful of chip enable output pins that can be connected directly to the other chips. Typically, the software has to tell the microprocessor that the address range should assert the various chip enable outputs.

OR

Address decoding refers to the way a computer system decodes the **addresses** on the **address** bus to select memory locations in one or more memory or peripheral devices.

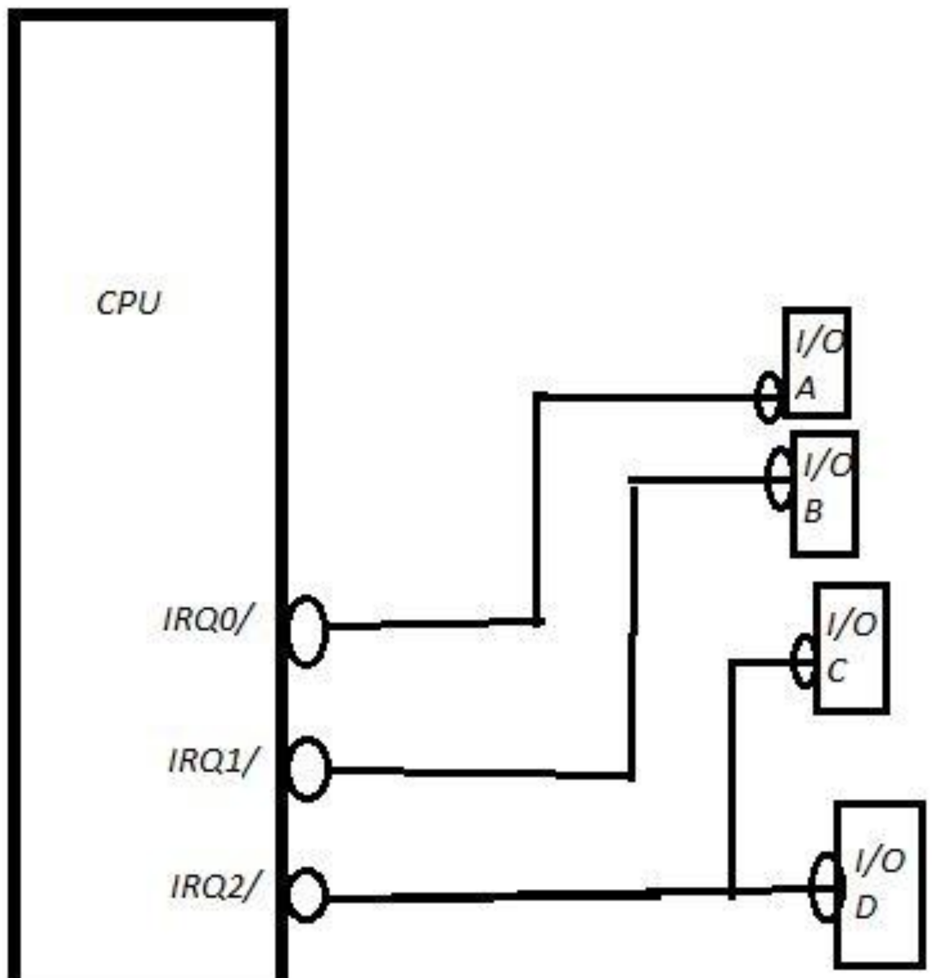
5) Memory cache and instruction pipeline:

A number of microprocessor particularly faster RISC(Reduced Instruction Set Computer)system contains a memory cache or cache in the same chip with the microprocessor . They are small but extremely fast memory that the microprocessor uses to speed up its work. The microprocessor can fetch the items that happens to be in the cache when they are needed much more quickly then it can fetch the item from separate memory chip.

An instruction pipeline or pipeline is similar to a memory cache in that microprocessor and load into the pipeline instruction that it will need. The difference between pipelining and cache are the pipelines typically much smaller than cache & the logic behind them is often much simpler and microprocessor uses them only for instruction not for the data.

Unit 2
introduction

Interrupt causes the µP in the embedded system to suspend doing whatever it is doing to execute some different code instead the code that will response whatever event causes the interrupt. Interrupt can solve the response problem without some difficult programming and without introducing some new problems of their own.



INTERRUPT
CONNECTION

Assembly Language

It is human readable form of the instruction that the µP really knows how to do it. Assembler translates the ALL into MLL. When the compiler translates C, most of the statement becomes multiple instruction for the µP to execute. The typical µP has a set of registers called general purpose register.

Most of the µPs have several special purpose registers called PC which keeps the addresses of next instruction that the µP executes. Most of the SPs which stores the memory address of the top of the general purpose µP stack. Some µPs can only do arithmetic in special register called AC. A.L.L. have “JUMP” instruction that unconditionally continue the execution from break point.

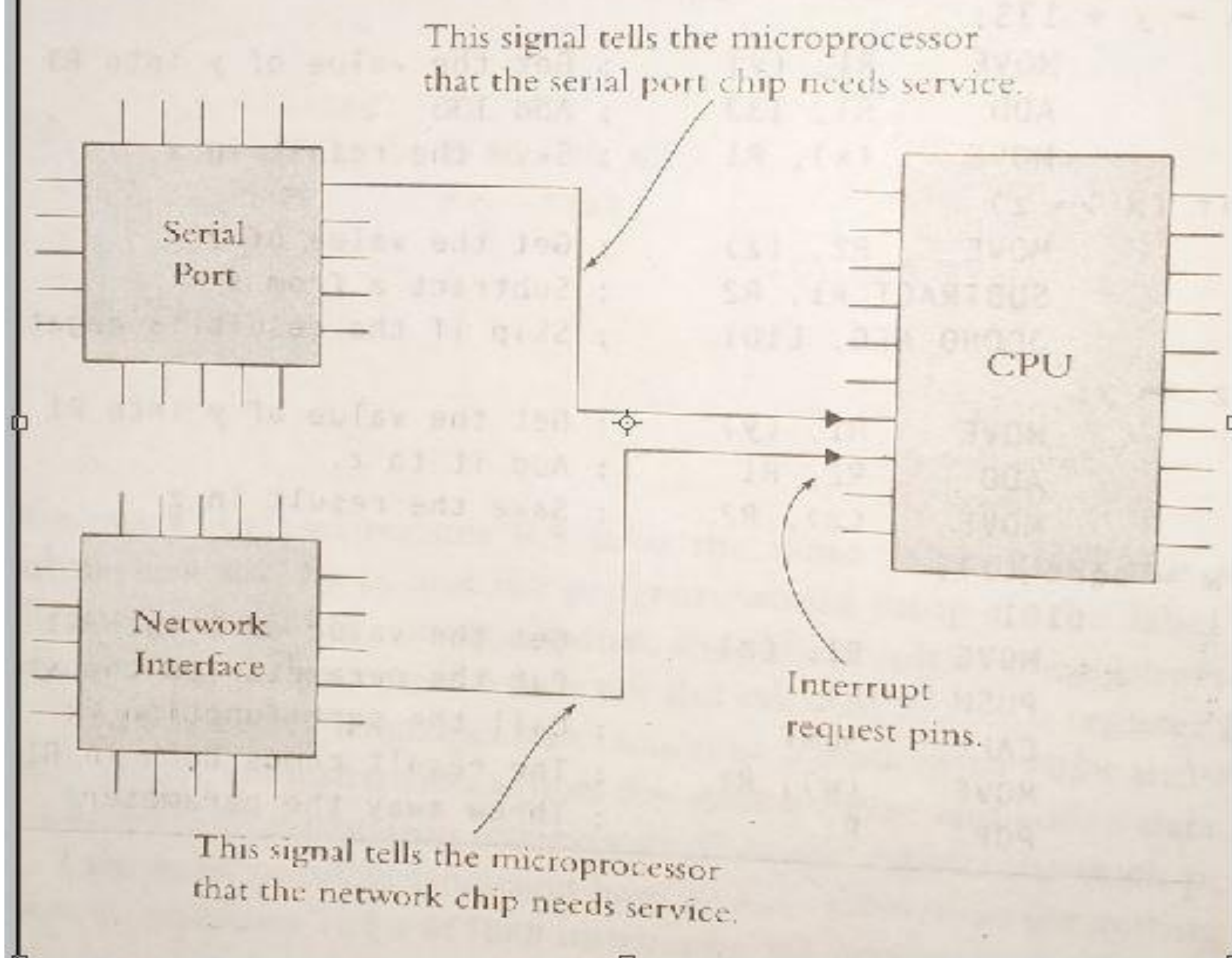
Interrupt Basic

In Interrupt Basics, we study what the µPs typically do when an interrupt happens, what interrupt routines typically do and how they are usually written. Each of these chips have a pin that it asserts when it requires services. Interrupt routines are the sub-routines of that do whatever needs to be done when interrupt signals occur.

For eg. When the interrupt come from the serial port chip and that has service a character from the serial port, then the interrupt routine must read the characters from the serial port chip and put into the memory.

An interrupt routine sometimes called interrupt handler or interrupt service routine

Figure 4.2 Interrupt Hardware


This signal tells the microprocessor that the serial port chip needs service.
Serial Port
CPU
Network Interface
Interrupt request pins.
This signal tells the microprocessor that the network chip needs service.

The last instruction to be executed in an interrupt routine is an ALL “RETURN” instruction. When it gets there, the µP retrieves the address of next instruction from the stack, and resume execution.

Disabling interrupts

Almost every system allows you to disable interrupts, usually in a variety of ways. To begin with, most I/O chips allow our programs to tell them not to interrupt, even if they need the microprocessor’s attention. This stops the interrupt signals at the source. Most microprocessors allow your program to tell them to ignore incoming signals on their interrupt request pins.

Most of a microprocessor have non-maskable interrupt i.e a hardware interrupt that cannot be ignore by standard interrupt masking techniques in the system. Some microprocessor use different mechanism for disabling and enabling the interrupt.

These microprocessor assign a priority to each interrupt request signals and allow the program to specify the priority of the lowest priority interrupt that is will to handle at any given time. It can disable all interrupt by setting the acceptable priority higher than that of any interrupt it can enable all interrupt by setting the acceptable priority of low.

Shared data problem

One problem that arises as soon as you use the interrupt routines need to communicate with the rest of your code. It is usually neither possible nor desirable for the µP to do all it’s work in interrupt routines. Therefore interrupt routines need to signal the task code to do follow up processing.

For this to happen, the interrupt routines and the task code must share one or more variables that they can use to communicate with one another.

\ OR

The shared data problem occurs when several functions (or ISRs or tasks) shares a variable. Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable before the completion of previous task operations.

Example

```
static int iTemperatures[2];
void interrupt vReadTemperatures (void)
{
 iTemperature[0]=!! read in values from hardware
 iTemperature[1]=!! read in values from hardware
}
void main(void)
{
int iTemp0,iTemp1;
while (TRUE)
{
 iTemp0=iTemperature [0];
 i Temp1=iTemperature [1];
 If (iTemp0!=iTemp1)
 !! Set off howling alarm;
}
}
```

Above program code segment illustrate the shared data problem which is the part of nuclear reactor monitoring system. This code monitors the two temperatures which are always supposed to be equal if they differ, it indicates that there is some malfunctioning in the reactor.

The main function is in infinite loop making sure that two temperatures are same. The interrupt routine vReadTemperatures (void) happens periodically. If one or both of the temperatures changes or per haves the timer interrupt in every few millisecond causes the µP to jump into the interrupt routine of new temperature.

Solving shared data problem

- The first method for solving shared data problem is to disable the interrupt whenever task code uses the shared data. The hardware can assert the interrupt signal requesting service and then µP will not jump to the interrupt routine while the interrupts are disable.

Interrupt Latency

- Interrupts are the tools for getting better response from our system and because the speed with which an embedded system can respond is always of interest, one obvious question is, “How fast may system respond to each interrupt?” The answer of this question depends upon a number of factors:-

1. The longest period of time during which the interrupts is disables.
2. The period of time it takes to execute any interrupt routines for interrupts that are of higher priority that the one in question.
3. How long it takes the μP to stop what is doing, do the necessary bookkeeping, and start executing instructions within the interrupt routine.
4. How long it takes the interrupt routine to save the context and then do enough work that what it has accomplished counts as ‘respone’.

The term interrupt latency refers to the amount of time it takes to execute or to respond by the system to an interrupt.

For example, suppose that you are writing a system that controls a factory, and that every second your system gets 2-dozen of interrupts to which it must respond promptly to keep the factory running smoothly .

Suppose that your system monitors a decoder that checks for gas leaks, and that your system must call the fire department and shunt down the affected part of the factory if a gas leak is detected. The interrupt routine that handles gas leaks need to be relatively high priority to get the µP’s attention first, especially if those interrupt routines open and close electrical switches and cause an explosion.

UNIT -3

SURVEY OF SOFTWARE ARCHITECTURE

Introduction

The most important factor that determines which architecture will be the most appropriate for any given system & how much we control over the system response. To achieve the good response it depends not only on the absolute response time requirement but also on the speed of the up & and the other processing requirements. All with different deadlines & different priorities, the different software architecture are:

i. Round Robin
ii. Round Robin with interrupt
iii. Function queue scheduling architecture.
iv. Real time operating system architecture.

Round Robin:

Round Robin architecture is the simplest architecture for survey of any embedded system which consists of no interrupt. The main loop simply checks each of the I/O devices in turn & services that needs to service. Following code segments represents the prototype for Round Robin.

```
void main()
    {
        while(TRUE)
        {
            if(!! I/O device A need service)
            {
                !!Take care of I/O Devices A
                !! Handle data to or from I/O Device A
             }
            if(!! I/O Device B needs service)
            {
                !! Take care of I/O device B
                !! Handle data to or from I/O device B
            }
    etc.
    etc.
    if (!! I/O device Z needs service)
    {
       !! Take care of I/O device Z
       !! Handle data to or from I/O Device Z
     }
  }
}
```

From above code segment there is no interrupt occur, no share data , no latency concern & therefore always an attractive potential architecture. For example a digital multimeter that measures electrical resistance, current & potential in the units of Ohm's , amperes & volts each in the units of ohm's , amperes & volts each in several different ranges . A typical multimeter have two probs that the user touches two points on the circuit to be measure & a digital display & the rotary switch that selects which measurement to make in what the range. The system makes continous measurement & changes the display to reflect the most recent measurements. The possible pseudo code for multimeter is given below.

```
Void vDigitalMultiMeterMain (void)
{
Enum {OHMS_1,OHMS_10,……,VOLTS_100}
eSwitchPosition;
While(TRUE)
{
eSwitchPosition=!!Read the position of the switch;
switch(eSwitchPosition)
{
case OHMS_1:
!! Read hardware to measure ohms
!!Format result
break;
case OHMS_10
!! Read hardware to measure ohms
!!Format result
break;
.
.
.
case VOLTS_100:
!! Read hardware to measure ohms
!!Format result
break;

}
!! Wrtie result to display
}
}
```

Advantage and disadvantages of round robin architecture:

Advantage:

- The round robin architecture has only one advantage over other architecture is **simplicity.**

Disadvantages:

- If any one device needs response in less time then it takes the Microprocessor to get around the main loop in the worst case scenario.
- Even if one of the required response time are absolute deadlines the system may not work well if these is any lengthy processing to do.

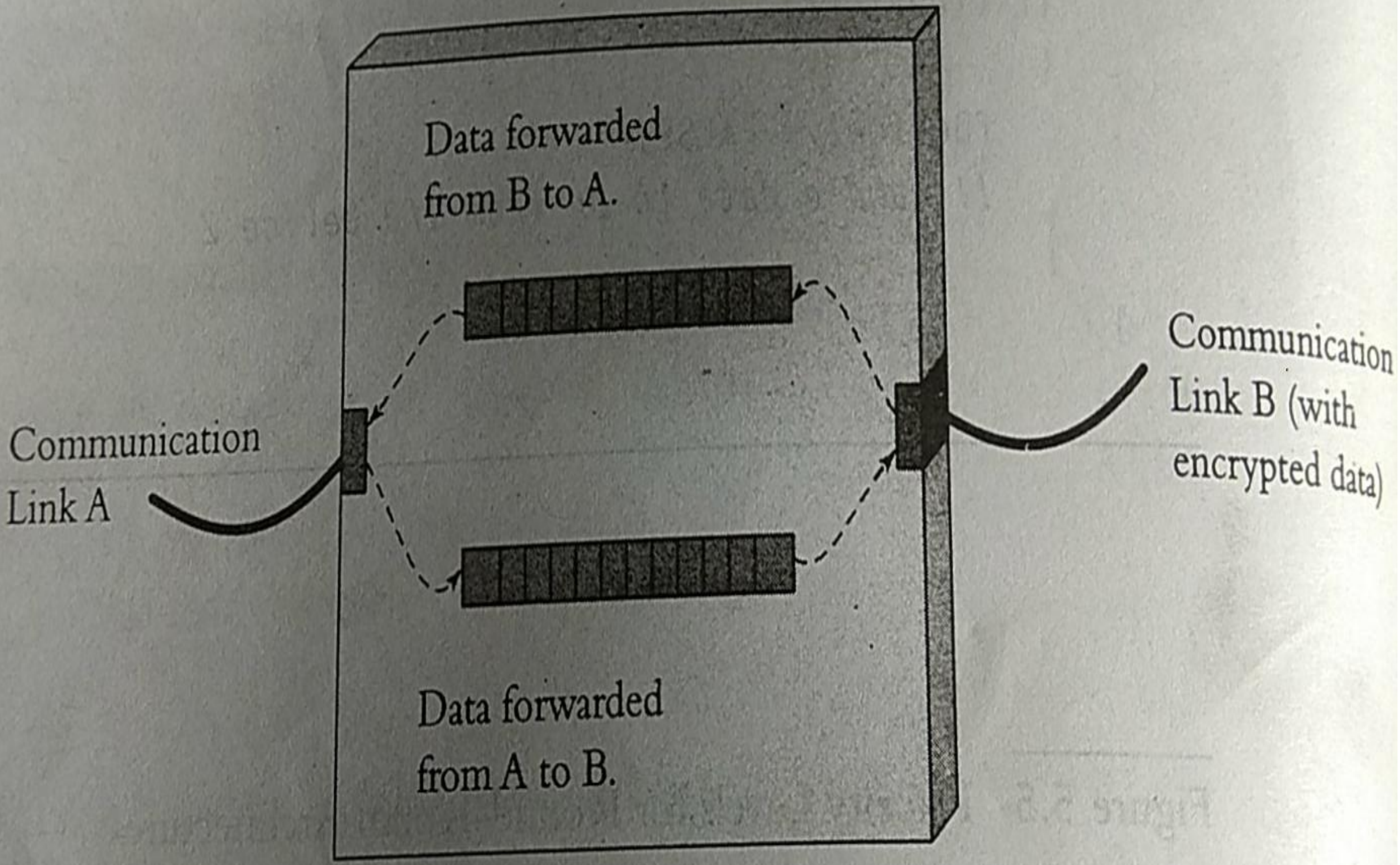
Round Robin with interrupt:

In this architecture interrupt routine deals with the very urgent needs of hardware & sets of flag. The main loop polls the flag & does any follow of processing required by the interrupt. This architecture gives little bit more control over the priorities. The interrupt routine can get good response because the hardware interrupt signals causes the microprocessor to stop what ever it is doing in the main function & execute the interrupt routine instead. Effectively all the processing that we put into the interrupt routine has higher priority than the task code in the main routine.

The example of Round Robin with interrupt are a simple bridge & the Bar Code Scanner.

One example of Round Robin with interrupt is communication bridge which is device with two parts on it that forwards the data traffic received on the first port to the second & vice - versa which is shown in figure given.

Figure 5.6 Communications Bridge


Data forwarded from B to A.
Communication Link A
Communication Link B (with encrypted data)
Data forwarded from A to B.

Function queue scheduling Architecture:

In this architecture the interrupt routines adds the functions pointer to the queue of the function. The main routine just reads pointer from the queue and calls the function . In this architecture the worst wait for the highest priority task code function is the length of the longest of the task code function. This worst case if the longest task code function has just started when the interrupt for highest priority device occurs Under the Round Robin with interrupt architecture all of the task code gets a change to run .

Under this architecture lower priority functions may never execute if the priority function frequently enough to use up all the microprocessor available time. Although the function queue scheduling architecture reduces the worst case response for the high priority task code & it may still not be good enough because one of the lower priority task code functions are quite long and it affect the response for the highest priority function.

Real time operating system architecture:

In real time operating system architecture the other interrupt routines take care of the most urgent operations. They then signal that there is work for the task code to do. Some features are:

- The necessary signaling between the interrupt routine & the task code handled by the real time operating system (RTOS).
- No loop in the code decides what needs to be done next. Code inside the real time operating system decides which of the task code function should be run.
- The RTOS can suspend one task code sub-routine in the middle of its processing in order to run.

SELECTING AN ARCHITECTURE:

Few suggestions for selecting the system architecture are:

- Select the simplest architecture that will meet your response requirement. Writing embedded system software is complicated enough without choosing unnecessarily complex architecture for your system.
- IF the system has response requirement that might necessitate using RTOS. We should lean towards using real time operating system(RTOS)
- If it makes sense for our system we can create hybrids for the architectural system.

Unit-4

Real Time Operating System and services

Introduction :

Embedded system is a field in which the terminology is inconsistent. Many people uses acronym real time OS are different from desktop machine OS such as windows , or Linux etc. On a first place, on a desktop computer the OS takes the controls of the machine as soon as it is turned on & starts the application separately from the OS. In embedded system, you usually link your application and the RTOS. At boot up time, your application usually gets control first, and it then starts the RTOS. Thus , the application and the RTOS are much more tightly tied to one another than are an application and its desktop operating system.

In the second place, many RTOSs do not protect themselves as carefully from your application as do desktop OS. For example, whereas most desktop operating systems check that any pointer you pass into a system function is valid, many RTOSs skip this step in the interest of better performance.

In the third place, to save memory RTOSs typically include just the service that you need for your embedded system and no more. Some RTOS available to as areVxWorks, VRTX, pSOS, Nucleus, C Executive, LynxOS, QNX, Multitask!, AMX etc.

Task and Task state:

The basic building blocks of software written under an real time operating system is task are very simple to write under most real time operating system & task is simply a sub-routine. Task state is the state that occurs during processing or execution by the system to any task.

Basically there are three task state which are:

i. Running
ii. Ready
iii. Blocked

- Running:

  Running means that the microprocessor is executing the instructions that make up this task . If there is only one microprocessor than only one task that is in the running state at any given time.

- Ready:

  Which means that some other task is in the running state but that task has the things that it could be do if the microprocessor becomes available. Any number of task can be in this state.

- Blocked:

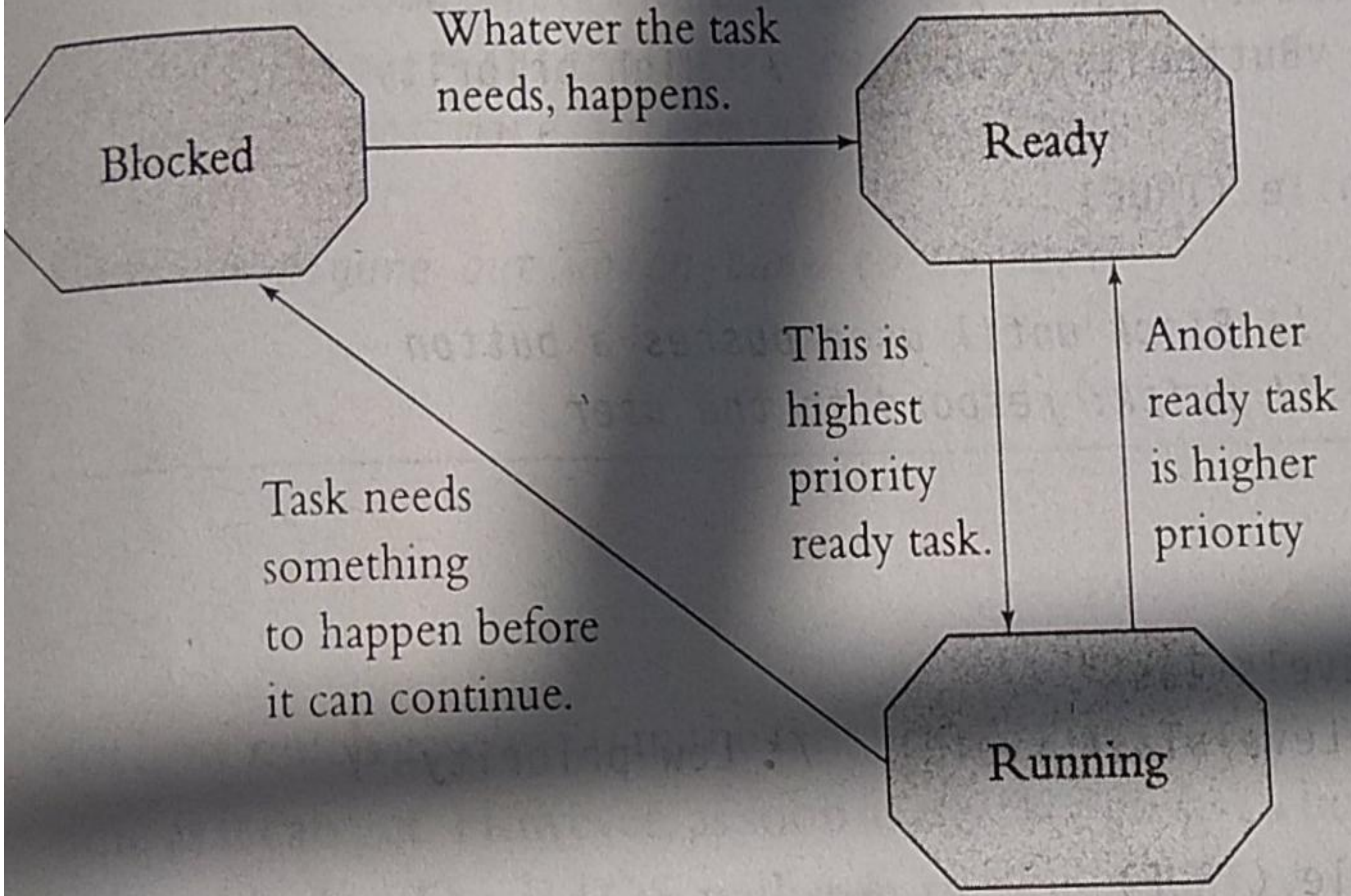
  which means that this task has not got anything to do right now even if the microprocessor becomes available. Task get into this state because they are waiting for some external events.

The Scheduler:

A part of real time operating system called scheduler which keeps the track of the state of each task & decides which one task should go into the running state. The scheduler in most real time operating system are entirely simple minded about which task should get the processor. They looked at priority, we assign to the task & among them which are not in blocked state.

One tasks with highest priority runs & remaining of other task are waiting for ready state. The lower priority task just have to wait the scheduler assuming that we know what we were doing when setting the task priorities. Figure below shows the transactions among the three tasks state.

Figure 6.1 Task States


Whatever the task needs, happens.
Blocked
Ready
This is highest priority ready task.
Another ready task is higher priority
Task needs something to happen before it can continue.
Running

Here we will adopt the fairly common use of the verb block to mean “move into the blocked state”, the verb run to mean “move into the running state” and the verb switch to mean “change which task is in the running state.” Here are following points for figure:

- A task will only block because it decides for itself that it has run out of things to do. Other tasks in the system or the scheduler cannot decide for a task that needs to wait for something.

– While a task is blocked, it never gets microprocessor. Therefore, an interrupt routine or some other task in the system must be able to signal that whatever task was waiting for has happened. Otherwise, the task will be blocked forever.

– The shuffling (to switch) of tasks between the ready and running states is entirely the work of the scheduler. Task can block themselves, and tasks and interrupt routines can move other tasks from the blocked state to the ready state, but the scheduler has control over the running state.

TASK and DATA:

Each task has its own private context which includes the register value, a program counter & a stack. However all other data i.e global , static, initialized & uninitialized & every thing else is shared among all of the task in the system.

The RTOS typically has its own private data structure which are not available to any other task. Since we can share the date variables among the task, it is easy to move the date from one task to another task which only have access to some other variables.

Figure:

Semaphore

- Semaphore means a hardware or software flag. In multitaskingsystems, a semaphore is a variable with a value that indicates the status of a common resource. It's used to lock the resource that is being used. A process needing the resource checks the semaphore to determine the resource's status and then decides how to proceed.

Semaphores and shared Data:

A common use of semaphore are the simple way to communicate from one task to another task or from an interrupts routine to another task. Let us example the railway baron discovered that it was bad for business if their train run into one another. There solution to this problem was to use signals called semaphores. When the first train enters into the protected section of track, the semaphores behind it automatically lowers.

When the train cross the protected section than next train starts to enter the protected section. The general idea of semaphore in RTOS is similar to the idea of railway semaphores. Trains do two thinks with semaphores.

- When a train leaves the protected section of track, it raises the semaphore.
- When a train comes to a semaphore, it waits for the semaphore to rise, if necessary, passes through the (now raised) semaphore, and lowers the semaphore.

RTOS semaphores:

Although the word was originally coined for a particular concept, the word semaphore is now one of the most slippery in the embedded systems word. It seems to mean almost as many different things as there are software engineers or at least as these are RTOS, some RTOS even have more than one kind of semaphore. RTOS use get and give, take and release, pend and post, p and v, wait and signal, and any number of other combinations. We will use take (for lower) and release (for raise). This kind of semaphore is most commonly called binary semaphores.

A typical RTOS binary semaphores work like this: tasks can call two RTOS functions, TakesSemaphore and ReleaseSemaphore. If one task has called Take-Semaphore to take the semaphore and has not called ReleaseSemaphore to release it, then any other task that calls TakeSemaphore will block unit the first task calls ReleaseSemaphore. Only one task can have the semaphore at a time.

Message , Queues , Mailboxes & Pipes:

Task must be able to communicate with one another to coordinate their activities or to share the data.

for example: In underground tank monitoring system the task that calculate the amount of gas in the tank must let other parts of the system to know how much gasoline is available. for example, if there is two task, task1 & task2 each of which has a number of high priority urgent things to do. These all signals to send to the system consist of message.

Mailbox:

In general mailbox are much like queues. The typical RTOS has function to create, to write, & to read from the mailboxes & perhaps function to check whether the mailbox contain any message & to destroy the mailbox if it is no longer needed. The details of mailbox however are different in different RTOS. Some of the variation of mailboxes are:

- Although some RTOS allow a certain number of messages in each mailbox. A number of messages in each mailbox. A number that we can usually choice when we create the mailbox.
- In some RTOS the number of messages in each mailbox is unlimited.
- In some RTOS we can priority mailbox messages. Higher priority messages will be read before lower priority messages.

- Mailboxes provide a means of passing messages between tasks for data exchange or task synchronization. For example, assume that a data gathering task that produces data needs to convey the data to a calculation task that consumes the data.

Pipes:

Pipes are also much like queues. In pipes the RTOS can create them, write to them, read from them & so on. The details of pipes however like the details of mailboxes and queues, vary form RTOS to RTOS. Some variations of pipes are given below:

- Some RTOS allow to write the message of varying length onto the pipe.
- Pipes in some RTOS are entirely byte oriented.
- Some RTOS uses the standard c-library functions fread & fwrite to read from and to write into the pipe.

Events:

Event is an action performed by the user such as clicking of mouse, key press etc. An event is essentially a boolean flag that the task can be set or reset & other task can wait for. For eg: When the user pulls the trigger on the cordless bar code scanner the task that turn on the laser scanning mechanism & tries to recognized the bar code that must start. The events provides the easy way to do these task. The user pulls the trigger sets on events for which the scanning task is waiting.

Some standard features of events are:

- More than one task can blocked waiting for the same event & the RTOS will unblock all of them when an event occurs.
- RTOS typically forms group of events & task can wait for any sub-set of events within the group.
- Different RTOSs deal in different ways with issue of resetting an event after it has occurred and tasks that were waiting for it have been unblocked. Some RTOSs reset events automatically.

Memory Management:

Most RTOS have some kinds of memory management sub-system although some offers the equivalent of library functions like malloc, calloc & free. Real time system avoid this two functions because they are typically slow & their execution time is unpredictable. The fixed favor function that allocate and fixed the fixed size buffer & most RTOS offer fast & predictable function for purpose. The reqbuf & getbuf function allocate the memory buffer from the various poll. Each returns a pointer to the allocate buffer. The only difference between them is if no memory buffers are available than the get buffer will block the task that calls it where as reqbuf will return a null pointer.

Interrupt routine in RTOS Environment:

Interrupt routine in most RTOS environment must follow two rules that do no apply to the task code.

- An interrupt routine must not call any RTOS function that might block the caller. Therefore interrupt routine must not get the semaphores , read from queue mailbox that might be empty and wait for events.
- An interrupt routine may not call any RTOS function that might cause the RTOS to switch task unless the RTOS know the interrupt routine.

Unit: 5

BASIC DESIGN using RTOS

Intro

It can be more difficult event to specify a real time system properly than to specify the desktop application. In addition to answering the question "what must the system do?" the specification must answer question about how fast must do it?" We can simply simplified that the cordless bar code scanner will send bar codes across the radio links to the cash register, the cashier will become unproductive & bored if it is long to wait for the signals that the bar code got successfully.

Further we must know how critical timing is. It may satisfactory for the cordless bar code scanner to respond on time. We must have some feels for the speed of microprocessor to know which companies will take long enough to affect other deadlines is a necessary design consideration.

We will use our general software engineering skills in designing embedded system software. The concern for structure modified encapsulation & maintainability are important in the embedded world as in the application world.

Systems with absolute deadlines, such as the nuclear reactor system, are called **hard real-time system**. Systems that demand good response but that allow some fudge in the deadlines are called **soft real-time system.**

To design effectively, you must know something about the hardware. For example, suppose your system will receive data on a serial port at 9600 bits(1000character) per second. If each received character will cause an interrupt, then your software design must accommodate a serial-port interrupt routine that will execute about 1000 times each second.

Principles

Here we will discuss the design consideration that have application to a broad range of embedded system.

General Operations

Embedded system is very commonly have nothing to do until the passage of time or some external event requires a response. If no print data arrives laser printers do nothing other than wake up every minute or so & move the printer drum a little. If the user does not pull the trigger or press one of the keyboard button then the cordless bar code scanner even goes so far as to turn the microprocessor off. Since external events generally causes interrupts & since we make the passage of time causes interrupt by setting up a hardware timer interrupt tends to be driving force of embedded system design technique.

Encapsulating semaphores & Queues:

<u>Encapsulating semaphores</u>

Semaphore can cause various bugs. At least some of those bugs stem from undisciplined use i.e allowing code in many different modules to use. The semaphores & hopping that they all use it correctly. We can squash these bugs before they get crawling simply by hiding the semaphores & the data that it protects inside of a module thereby encapsulating both.

Encapsulating Queues:

Similarly mistakes dealing with queues can be reduced by encapsulation queues typically have a protocol or format associated with multiple messages passing calls increases the probability of the communication mistakes. Encapsulating such calls into functions that take specific arguments reduces the errors.

Hard real time scheduling consideration

The issues that arises in hard real-time system are we must somehow guarantee that the system will meet the hard deadlines.

To some extend the ability to meet hard deadlines come from writing fast codes. Writing fast code for real time systems are not very different from writing fast codes for applications. We can characterized real-time system as:

- Made of N task that executes periodically every in units of time.
- Each task worst case execution time Cn units of time & deadlines of Dn.
- Assume task switching time is O & no blocking on semaphores.
- Each task has priority Pn.

Timer Function

Most embedded system keeps the track of the passes of time. Timer function represents the timing execution of microprocessor & its delay in the embedded system.

For example: in library management system to extend its battery life the cordless bar code scanner most turn itself after a certain no of seconds. System with network connection most wait for acknowledgement to data that they have send & retransmit the data if an acknowledgement does not show up on a time manufacturing system most wait for Robert alarms to move or for motors to come up to the speed.

Saving Memory Space

In embedded system we may be short of code space, we may be short of data space. They are interchangeable code are stored in ROM & data are stored in RAM. In RTOS each task need memory space for its task. Each function call, function parameter, local variable takes up a certain number of bytes on the task depending upon other microprocessor & compile. We must then add space for the worst case nesting of interrupt routine & some space is needed RTOS itself.

The method of saving memory is fill each stack with some recognizable data pattern at startup run the system for the period of time stop it & then examine how much of the date pattern was overwritten on each stack.

Here few ways to save code space are:

- Make sure that you are not using two functions to do something.
- Check that your development tools are not sabotaging you.
- Configure your RTOS to contain only those function that you need.

Saving power

A very common power saving mode is one in which the microprocessor stops the executing instructions that stops any built in peripherals & stops its clock circuit. The primary methods for preserving the battery power is turn off the parts or all of the system whenever possible. That includes the microprocessor another typical power saving mode is a one in which the microprocessor stops executing instructions but the board peripherals continue to operate. Any interrupt starts the microprocessor up again & microprocessor will execute the corresponding interrupt routines & then assumes the task code from the instructions that follows the one that put the microprocessor to sleep.

This mode saves less power than desired above. However no special hardware is required & don't have to start our software from beginning.

Example of a system to design:

The underground tank monitoring system monitors up to eight underground tanks by reading thermometers and the levels of floats installed in those tanks. To read a float level in one of the tanks, the microprocessor must send a command to the hardware to tell it which tank to read from. When the hardware has obtained a new float reading a few milliseconds later, it interrupts; the microprocessor can read the temperature in any time. Since gasoline expands and contracts substantially with changes in temperature, the system use both the temperature and the float level to calculate the number of gallons of gasoline in a tank.

The user interface consists of a 16-button keypad, a 20-character liquid crystal display, and a thermal printer. With the keypad, the user can tell the system to display various information such as the levels in the tanks or the temperatures or the time of day or the overall system status.

The system also has a connector to which a loud alarm bell can be attached to alert the gas station attendants if a leak is detected or if a tank looks as if it is about to overflow.

figure:

UNIT - 6

Embedded Software development tools

Host & target machine:

The embedded world there are many number of reasons to do our actual programming work on the system other then the one of which the software will eventually run in the target system may not have a keyboard screen a disc drive & other peripherals necessary for programming. It have not enough memory to run our editor program, the microprocessor in the target system don't support editor software, therefore must programming work for embedded system is done on the host, a computer system on which all the programming tools run only after the program has been written, compiled assembled & linked and moved to the target the system that is shipped to the customers.

Cross compilers

Most desktops systems used as hosts come with compilers, assemblers, linkers & so on for building programs that will run on the host. These tools are called native tools. The binary code produced by native tools is not understood by our target system microprocessor thus we need a compiler that runs on our host system but produces a binary instructions that will be understood by our target microprocessor such a program is called cross compiler.

Cross assembler & tool chains:

Another tool that we will need if we must write any of our program in assembly language is a cross assembler. A cross assembler is an assembler that runs in our host but produces binary instructions appropriate for our target.

The output files from each tool become the input files for the next because of this , the tools must be compatible with one another. A set of tools that is compatible in this way is called tool chain.

Linker / Locator for embedded software

FIGURE: Tool chain for building the embedded software

The job of cross compiler is to read in a source file & produce an object file suitable for the linkers. A linker for an embedded system most do a number of things differently from a native linkers. In fact the two programs are used for the embedded system specifying the location & link type, this is called linker & locator.

Getting embedded software into the target system

There are several ways of getting the embedded software into the target system some are:

i. PROM Programmers

The classic way to get the software from the locator output files into the target system is to use the files into the target system is to use the file to create a ROM OPROM. As creating ROM is only appropriate when software development has been completed, since the tooling cost to build ROMs is quite high.

Putting the program into the PROM requires a device & the programmer to insert the data called a PROM programmers. This is appropriate if our volumer are not large enough to justify using a ROM, if we plan to change software or while we are debugging. If we plan to use PROMs & a PROM programmers, for debugging purposes it is useful to build versions of the target system in which the PROM is placed in the socket on the target system rather than being soldered directly in the circuit. Then when we find bug we can remove the PROM containing the software with the bugs from the target system & put into the eraser like it is an erasable PROM or into the waste basket.

Fig: schematic edge view of a socket

ii. ROM Emulator

fig: ROM emulator

Another popular mechanism for getting the software into the target system for debugging purpose is to use of ROM emulator. A device that replaces the ROM in the target system. From the point of view of the rest of hardware in the target system the emulator look just like a ROM. However ROM emulator contains a large box of electronics and a serial port or a network connection through which it can be connected to your host. As with PROM programmers we must ensure that downloads new code into the ROM emulator understand the format of file that locator creates.

iii. Monitor

Another option on a system with communication port is use of monitor , a program that reside on the target ROM & knows how to load a new program into the system. A typical monitor allow to sent the software across the serial port , stores that software into the target RAM & then run it. Sometimes a monitor performs some of the function of locator as well & some monitor offers a few debugging services such as setting break points & displaying the memory & register value.

iv. Flash

Flash memory is the flash socket in our target system acts as EPROM. Most PROM programmer can program flash memory. Using flash memory make possible update our software containing the target machine from communication links. The reasons for using flash memory are:

- We can load new software into our system for debugging without pulling the chips of socket.
- Downloading a new software into a flash across a serial port or h/w connection is much faster.
- User can easily update his embedded software from hardware using flash memory.

UNIT: 7

DEBUGGING

Testing on host machines

fig: Test System

The target system is a trouble. Some testing environment The main goal of typical testing process are:

- Find the bugs early in the development process.

  Many studies have shown that this save time & money. In any case testing early gives some ideas of how many bugs you have & therefore how much trouble you are in but the target system is very often not available early in the process or the hardware may be unstable.
- Exercise all of the codes

  This includes all of the exceptional cases even through you hope that they will never happen. But is varies from difficult to impossible to exercise all of the codes invariably deals with unlikely situations.
- Develop reusable, repeatable tests

  It is extra ordinary frustrating to see a bug once but then not be able to find it because it refuses to happen again.
- Leave an " audit trail" of the test results

  Noticing that telegraph "seems to work" in the network environment is not nearly as valuable as knowing & storing exactly what data it send out in response to received data.

Instruction set simulator

Some of the short coming of the method can be overcome with an instruction set simulator or simulator , a software tools that runs on your host & simulates the behavior of microprocessor & memory in target system. To use simulator we can run our software through the cross compiler & linker locator just as building the real system. The simulator knows the target microprocessor architecture & instruction set. The user interfaces on most simulator are simulate to that of debugger allowing to run the program setting the break points, examining & changing the data in the memory & in the register single steps through the program. Simulator have some useful abilities they are:

i. Determining response & throughput

Simulators do not run at the same speed as the target microprocessor. Most will give the statistics from which

we can derive the time that given piece of code that will execute. For example the simulator can report the number of target microprocessor instruction it has executed all the number of target bus cycle it has simulated.

ii. Testing assembly language code

Since the simulator uses the target instruction set code return in assembly languages poses no problems we can run that code through the cross compiler & linker locator & then load into the simulator.

iii. Resolving portability issues.

Since we can use same tools chain to develop code for simulator & for final products. We should have some unpleasant surprises when we move from simulator to the target system or from host to the target system.

iv. Testing code dealing with peripherals built into the microprocessor

Most simulator will simulates the target microprocessor built in peripherals. The software uses built in timers to simulate & when time expires it will cause the simulated microprocessor to jump into the interrupt routine.

v. Others hardware

The simulator will simulates the microprocessor, ROM, RAM & built in peripherals & to extend the system the custom hardware , some specialized radios, sensor , ASIC , etc.

The assert MACRO

The assert macro is one good technique that application programmer use or at least should be used the applied embedded system. The macro takes a single parameter evaluates false than the asserts causes the programs to crash. Usually printing some useful message along the way.

Laboratory tools

i. Voltmeter & Ohm meter

It is a device used in lab. The voltmeter is used to check & test the voltage across any circuit where as Ohm meter is used to test & check the resistance. These are the extraordinary useful tools & not terribly expensive voltmeter measure the voltage difference between two points & ohmmeter measures the resistance between two points. A product commonly knows as multimeter.

Fig:

ii. Oscilloscopes

Oscilloscopes or scope is a device that graph voltage versus time. Time is graphed along a horizontal axis & voltage is graphed along vertical. An oscilloscope is an analog device that is it detects not just whatever a signal is high or low. Some features of oscilloscopes are:

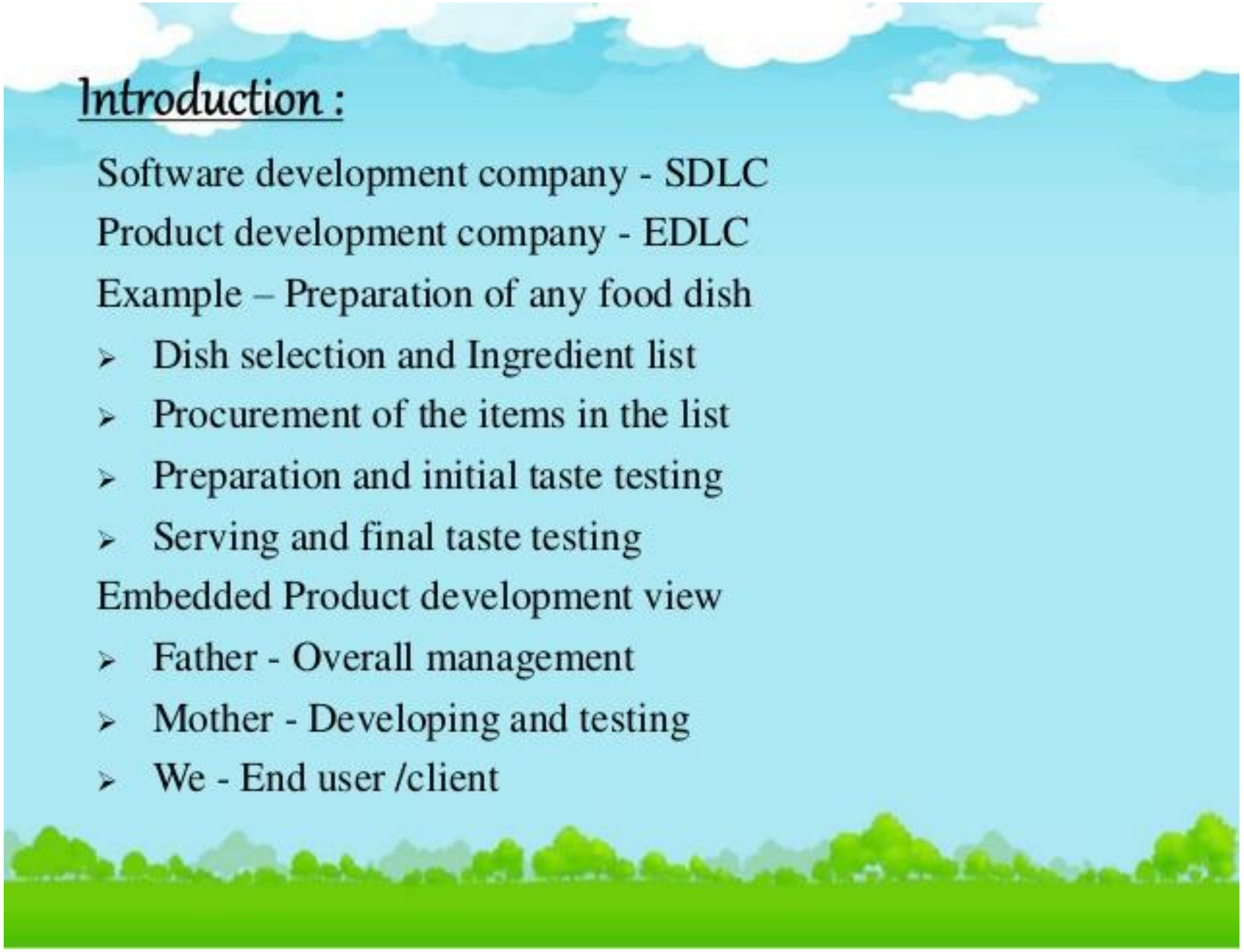
- Monitor one or two signals simultaneously.
- Adjust the time & voltage scales over a fairly wide range.
- Adjust the vertical level on the oscilloscope screen that corresponds to ground.
- Adjust the oscilloscope start graphing through the use of trigger mechanism.

iii. Software only monitors   figure

Another widely available debugging tool is one often called a monitor. Monitor allows to run software on the actual target microprocessor while still giving you a debugging interface similar to that of an in-circuit emulators. Monitor differ significantly from one another however we must examine carefully to know what we are getting. The typical working of monitors are

- One part of the monitor is a small program that resides in the ROM on the target system in the lab. A program that knows how to receive software on a serial port or across the network.
- Another part of the monitor is a program that runs on host system & communication with debugging with debugging kernel over the network.
- We can use instruct the monitor to set break parts run the program & so on. The user interface can run on host system & communicate the instruction to be debugging kernel running on the target.

Monitors can extra ordinarily valuable & they can give a debugging interface without any modification on the hardware called software only monitors.



Introduction :

Software development company - SDLC

Product development company - EDLC

Example – Preparation of any food dish

- Dish selection and Ingredient list
- Procurement of the items in the list
- Preparation and initial taste testing
- Serving and final taste testing

Embedded Product development view

- Father - Overall management
- Mother - Developing and testing
- We - End user /client

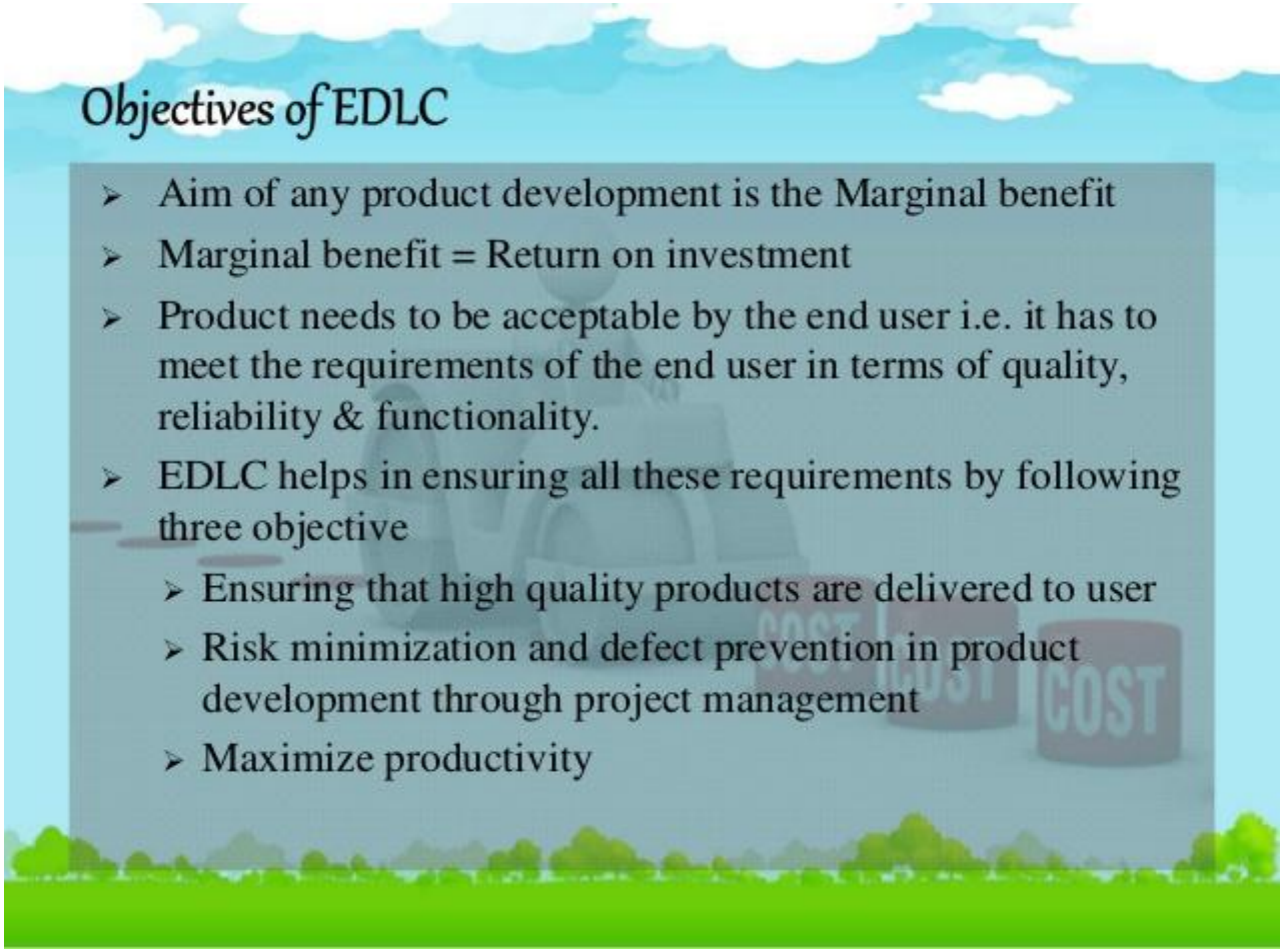
What is EDLC?

- EDLC is an Analysis-Design-Implementation based problem solving approach for the product development.

  - Analysis – What product need to be developed
  - Design – Good approach for building it
  - Implementation – To develop it

Why edlc?

- Essential in understanding the scope and complexities involved in any Embedded product development.
- Defines interaction and activities among Various groups of product development sector.
  - Project management
  - System design and development
  - System testing
  - Release management and quality assurance



Objectives of EDLC

- Aim of any product development is the Marginal benefit
- Marginal benefit = Return on investment
- Product needs to be acceptable by the end user i.e. it has to meet the requirements of the end user in terms of quality, reliability & functionality.
- EDLC helps in ensuring all these requirements by following three objective
  - Ensuring that high quality products are delivered to user
  - Risk minimization and defect prevention in product development through project management
  - Maximize productivity

Ensuring high quality products

- The primary definition of quality in any embedded product development is return on investment achieved by the product.
- In order to survive in market, quality is very important factor to be taken care of while developing the product.
- Qualitative attributes depends on the budget of the product so budget allocation is very important.
- Budget allocation might have done after studying the market, trends & requirements of product, competition .etc.

Risk minimization & defect prevention through project management

- Project management (PM)
  - Adds an extra cost on budget
  - But essential for ensuring the development process is going in right direction
- Projects in EDLC requires Loose project management or tight project management.
- PM is required for
  - Predictability
    - Analyze the time to finish the product (PDS = no of person days)
  - Co-ordination
    - Resources (developers) needed to do the job
  - Risk management
    - Backup of resources to overcome critical situation
    - Ensuring defective product is not developed

Increased productivity

- Measure of efficiency as well as ROI

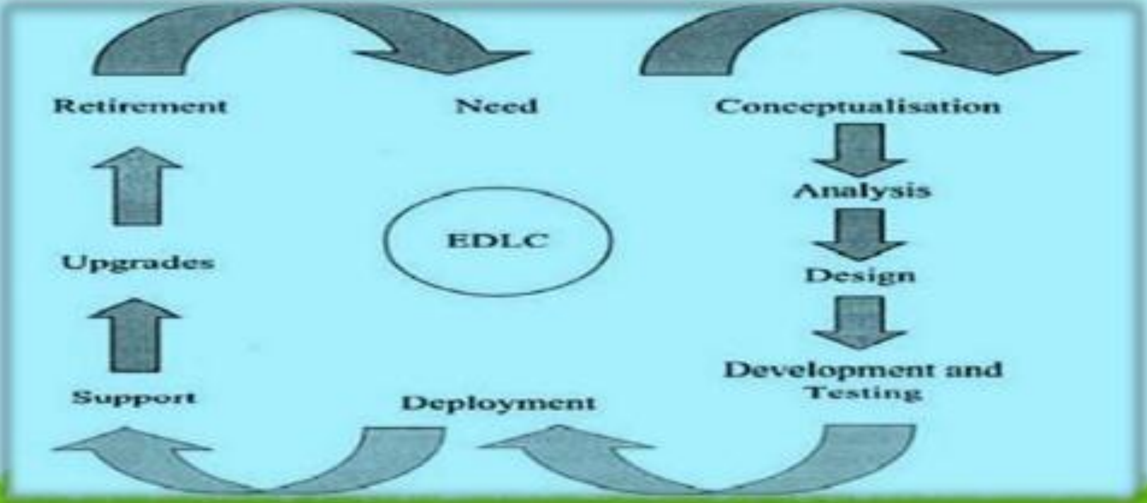
Different ways to improve the productivity are

- Saving the manpower
    - X members – X period
    - X/2 members – X period
- Use of automated tools where ever is required
- Re-usable effort – work which has been done for the previous product can be used if similarities present b/w previous and present product.
- Use of resources with specific set of skills which exactly matches the requirements of the product, which reduces the time in training the resource

Different phases of edlc

- A life cycle of product development is commonly referred as the "model"
- A simple model contains five phases
  - Requirement analysis
  - Design
  - Development and test
  - Deployment and maintenance
- The no of phases involved in EDLC model depends on the complexity of the product

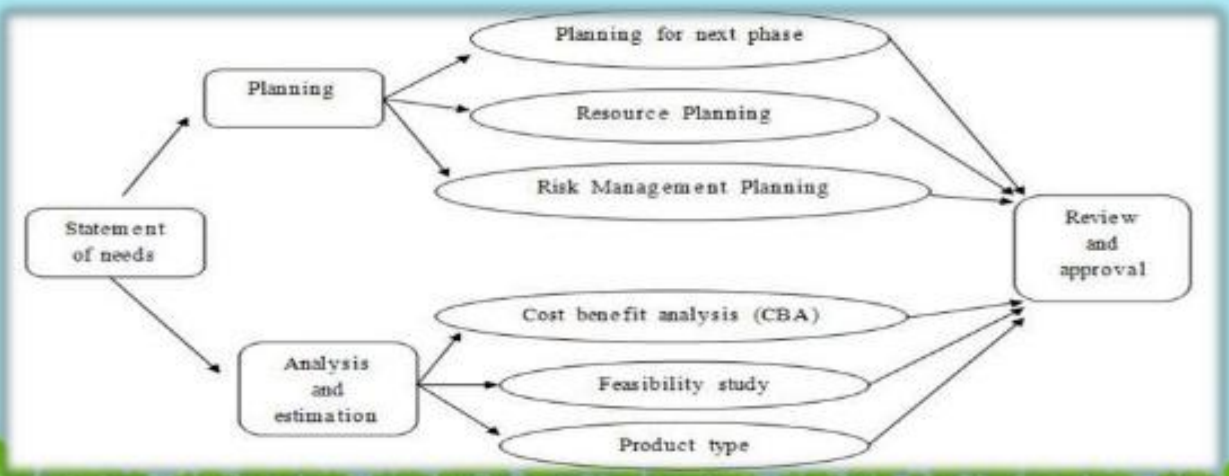
Classic Embedded product development life cycle model



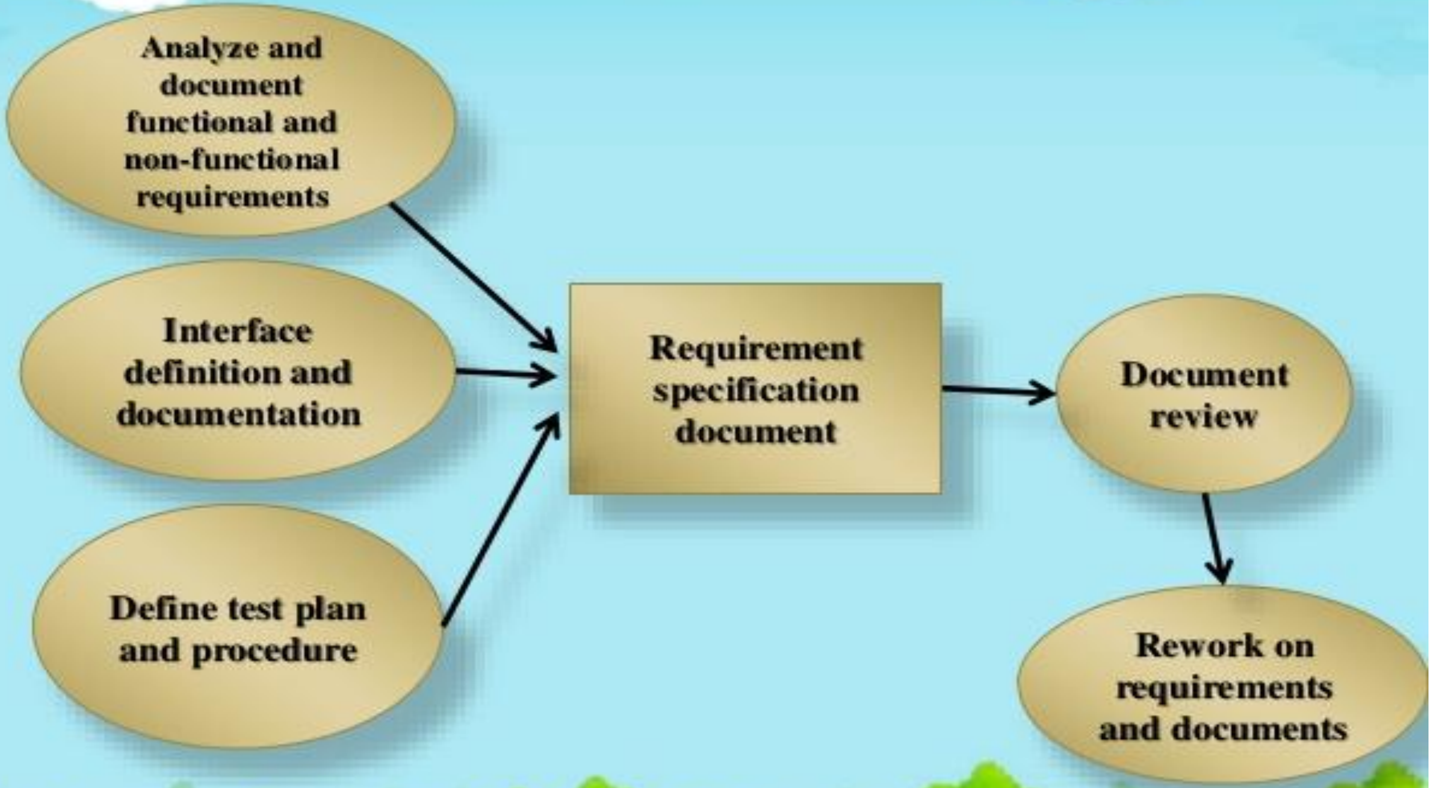
NEED:

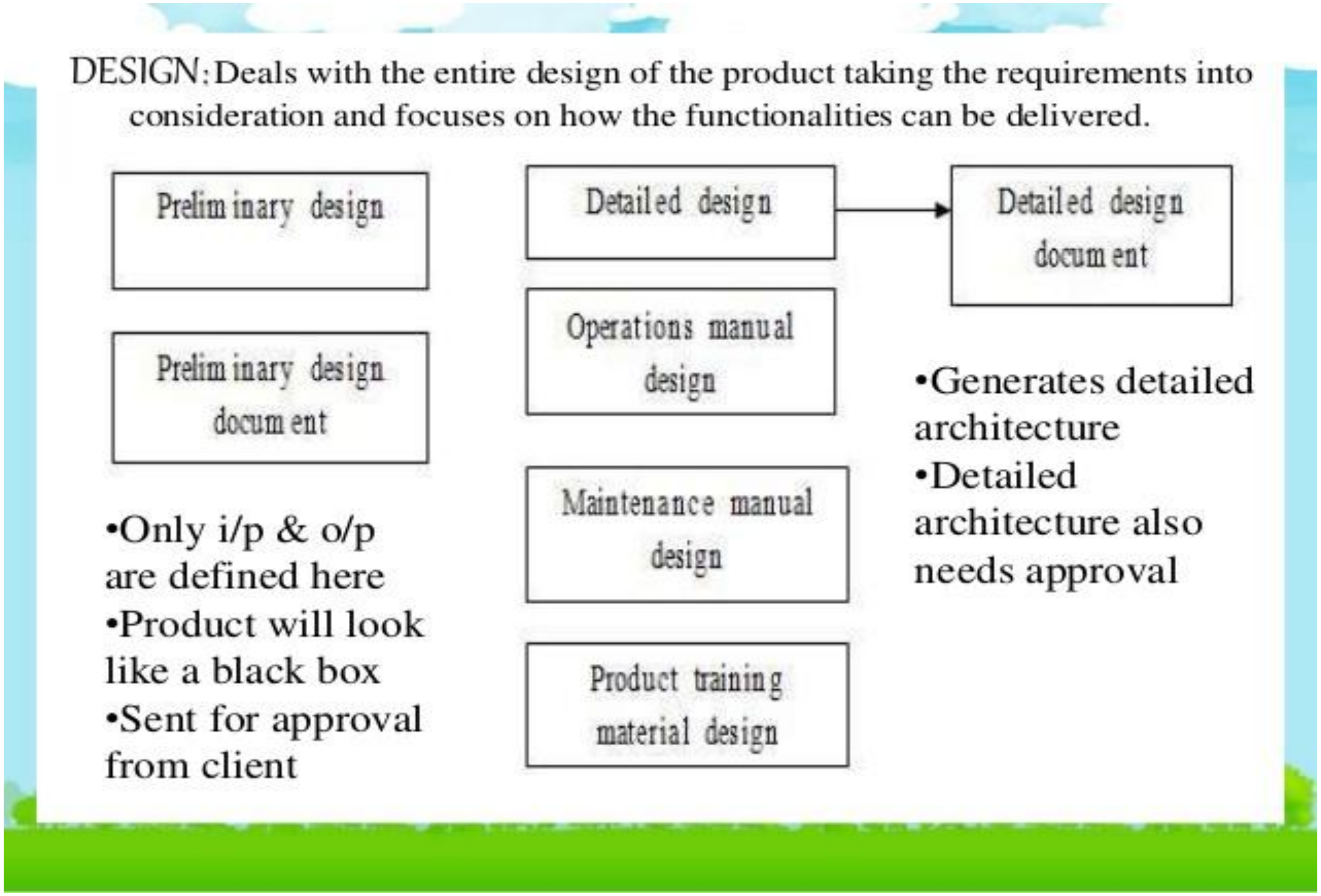
- Any embedded product may evolves as an output of a need.
- Need may come from an individual/from public/from company(generally speaking from an end user/client)
  - New/custom product development
  - Product re-engineering
  - Product maintenance

CONCEPTUALIZATION:



ANALYSIS:


Analyze and document functional and non-functional requirements
Interface definition and documentation
Define test plan and procedure
Requirement specification document
Document review
Rework on requirements and documents



DESIGN: Deals with the entire design of the product taking the requirements into consideration and focuses on how the functionalities can be delivered.

Preliminary design

Preliminary design document

- Only i/p & o/p are defined here
- Product will look like a black box
- Sent for approval from client

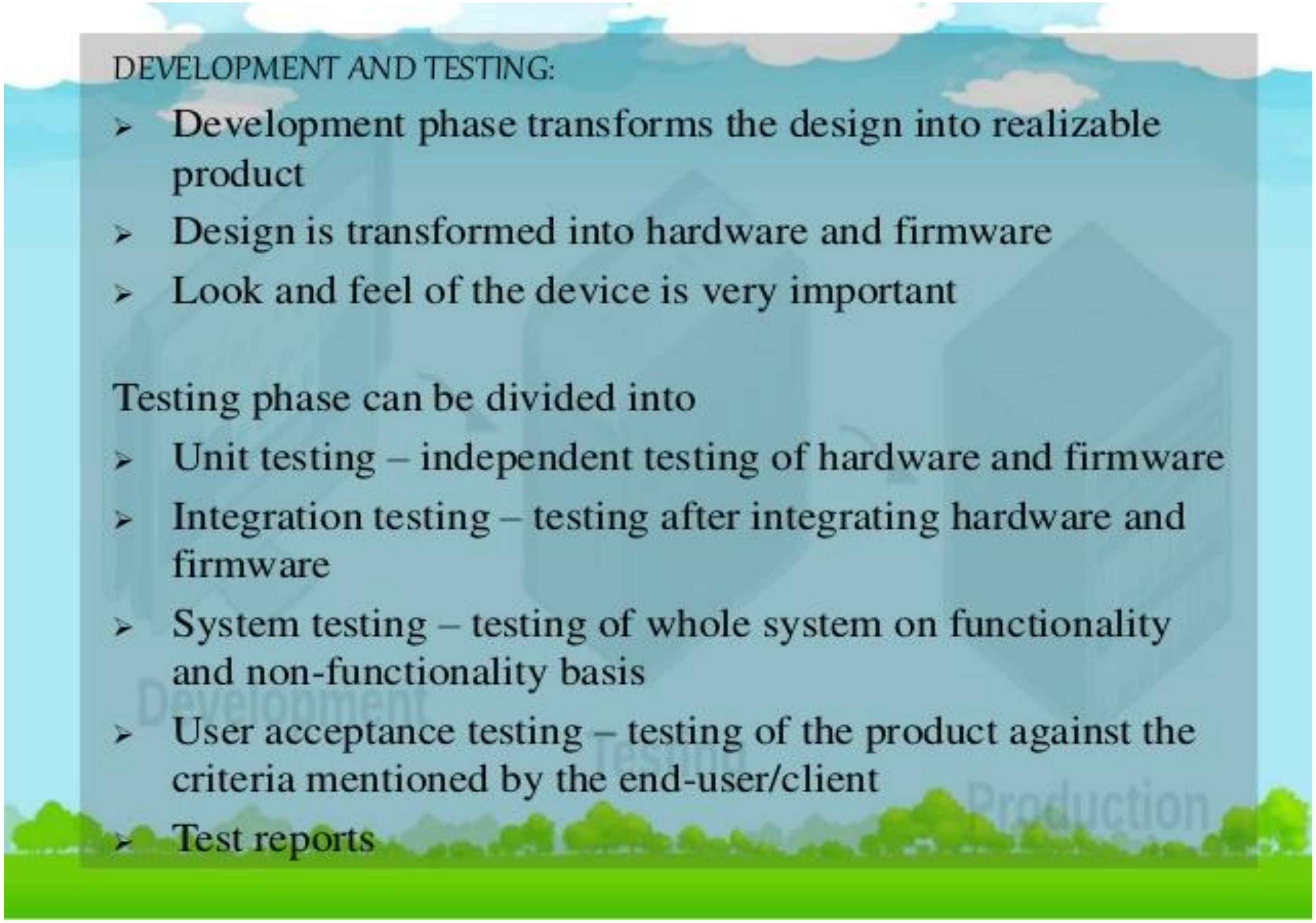
$$\begin{tikzcd}\text{Detailed design} \arrow[r] & \text{Detailed design document}\end{tikzcd}$$

Operations manual design

Maintenance manual design

Product training material design

- Generates detailed architecture
- Detailed architecture also needs approval



DEVELOPMENT AND TESTING:

- Development phase transforms the design into realizable product
- Design is transformed into hardware and firmware
- Look and feel of the device is very important

Testing phase can be divided into

- Unit testing – independent testing of hardware and firmware
- Integration testing – testing after integrating hardware and firmware
- System testing – testing of whole system on functionality and non-functionality basis
- User acceptance testing – testing of the product against the criteria mentioned by the end-user/client
- Test reports

DEPLOYMENT:

- A process of launching fully functional model into the market

SUPPORT:

- Deals with the operation and maintenance of the product
- Support should be provide to the end user/client to fix the bugs of the product

UPGRADES:

- Releasing of new version for the product which is already exists in the market
- Releasing of major bug fixes.

RETIREMENT/DISPOSAL:

- Everything changes, the technology you feel as the most advanced and best today may not be the same tomorrow
- Due to this the product cannot sustain in the market for long
- It has to be disposed on right time before it causes the loss.

Ways of DMA implementation

- *Direct Memory Access* (DMA) transfers data from an I/O device to the computer's memory. This method bypasses the CPU, using instead a device on the system board called a DMA controller. To transfer data without executing commands through the CPU, the card that implements the DMA signals the DMA controller when data is ready for transfer. During the actual transfer, the DMA controller acquires control over the CPU's data and address busses. To signal the end of the actual transfer, the DMA controller is programmed with the amount of data (number of words) to be transferred. The transfer can be prematurely terminated by initializing either the I/O device or the DMA controller during the transfer.

There are three limiting factors in DMA transfers: no program intervention by the CPU, long transfer latency times, and 64KB transfer boundaries. The first limiting factor, lack of program intervention by the CPU, is the reason that DMA is utilized in the first place. In order for conditions to be implemented that effect how data is selected for transfer, the peripheral card must be "smart" enough to detect when to change these selections. The second limiting factor, transfer latency time, occurs due to the fact that the CPU must finish its current instruction before yielding the bus to the DMA controller. If the CPU is executing a string or I/O instruction with a *rep* prefix, up to 128KB memory and/or I/O cycles could occur before the DMA controller gains access to the bus. Empirical testing has found some "AT" style computers with latencies between the DMA request and DMA cycle of up to 16 microseconds, thus limiting instantaneous transfer rates to 62KHz. This limitation can be overcome with peripheral FIFOs, allowing an average A/D transfer rate up to one million samples/second on an AT and up to four million samples/second on EISA computers. The third limiting factor, 64KB transfer boundaries, can be overcome by programming techniques.

- **Types of Data Transfers**

  The simplest DMA transfer is one where a device sends the same type of data. A data acquisition card, for example, would transfer one channel's data at a fixed frequency and gain for the duration of the transfer. Most data acquisition cards however allow DMA data transfers from a series of channels starting at the lowest channel to a programmed channel number, at a fixed rate.

- **DMA Controller on the PC**
- The DMA controller implemented on the PC and compatibles (8237A or its equivalent) transfers a maximum of 64KB words using DMA channels 5 to 7. This controller does not address memory in the same manner as the CPU. Figure 1 illustrates the two methods of addressing memory. The CPU addresses memory in real mode by shifting the Base register four bits to the left and adding the Offset register forming a 20-bit address. The DMA controller addresses memory by appending the Offset register to the first four bits in the Page register. This addressing scheme can cause problems when allocating a buffer for the DMA transfer. The allocated buffer should not contain more than one DMA page.

- **Lab Master AD**
- The Lab Master AD is one of the first smart cards for data acquisition available for the IBM ISA or EISA bus computers. It is equipped with 16 single or eight differential fully-programmable Analog-to-Digital (AD) channels with 12-bit resolution, two Digital-to-Analog (DA) channels, eight digital-input and eight digital-output channels, eight digital-expansion channels, and five 16-bit programmable timer/counters tied into a 4MHz base frequency. This card has a 2048 word FIFO buffer for AD and one 1024 word FIFO buffer shared by both the DA channels. The Lab Master makes use of both the Single and Demand transfer modes for AD and DA DMA transfers. The number of AD channels can be expanded to 64 single or 32 differential AD channels.

- **Data Acquisition Application**

  The routines in this article use DMA to create a working, high-speed data-acquisition system. I designed the routines to make it easy to port them to any type of data acquisition application. You can easily modify the code to provide variable channel, frequency, and gain sequences, a pseudo real-time display, and some signal processing capabilities.

Figure 6.5 Data in an RTOS-Based Real-Time System

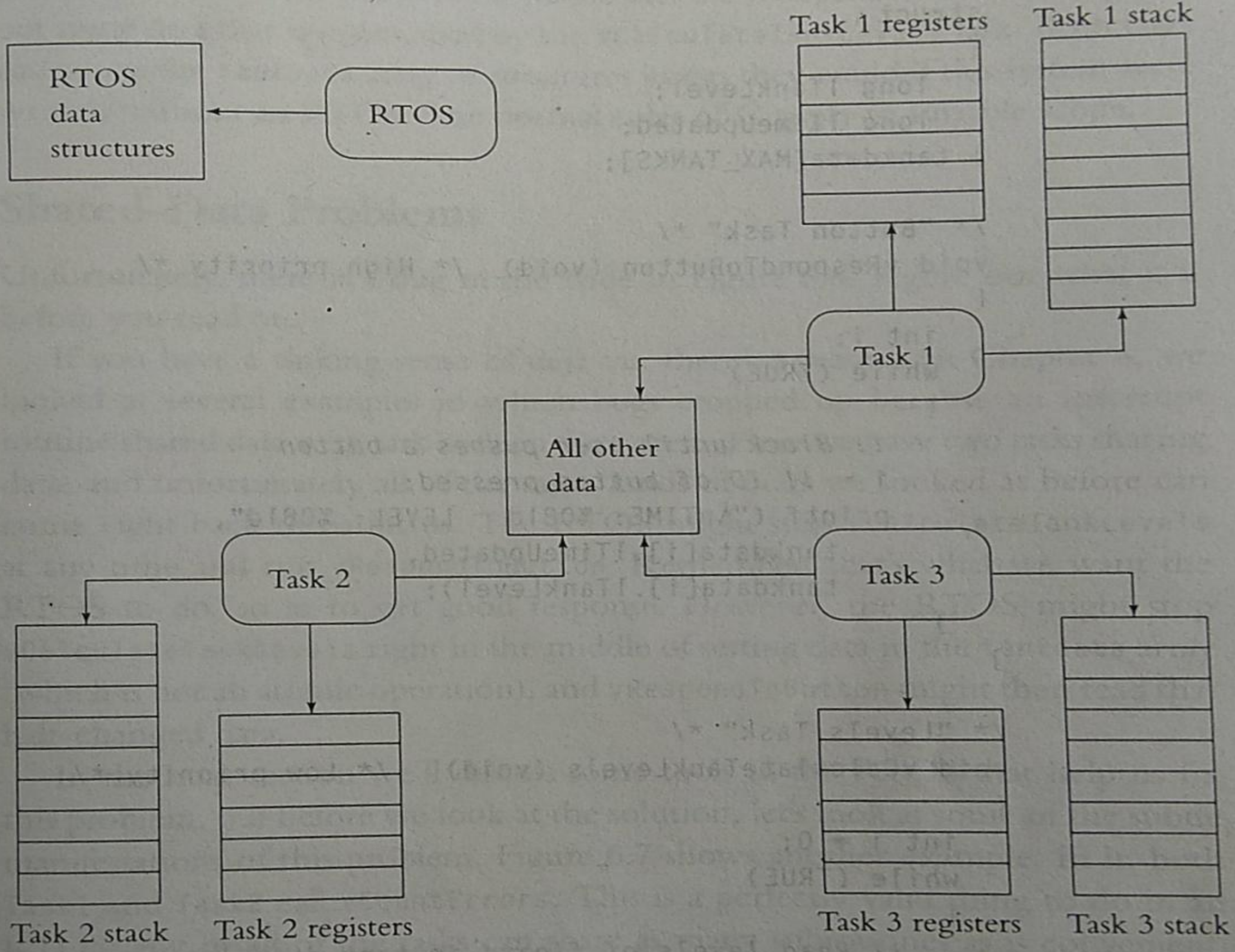


Figure 8.7 Tank Monitoring System

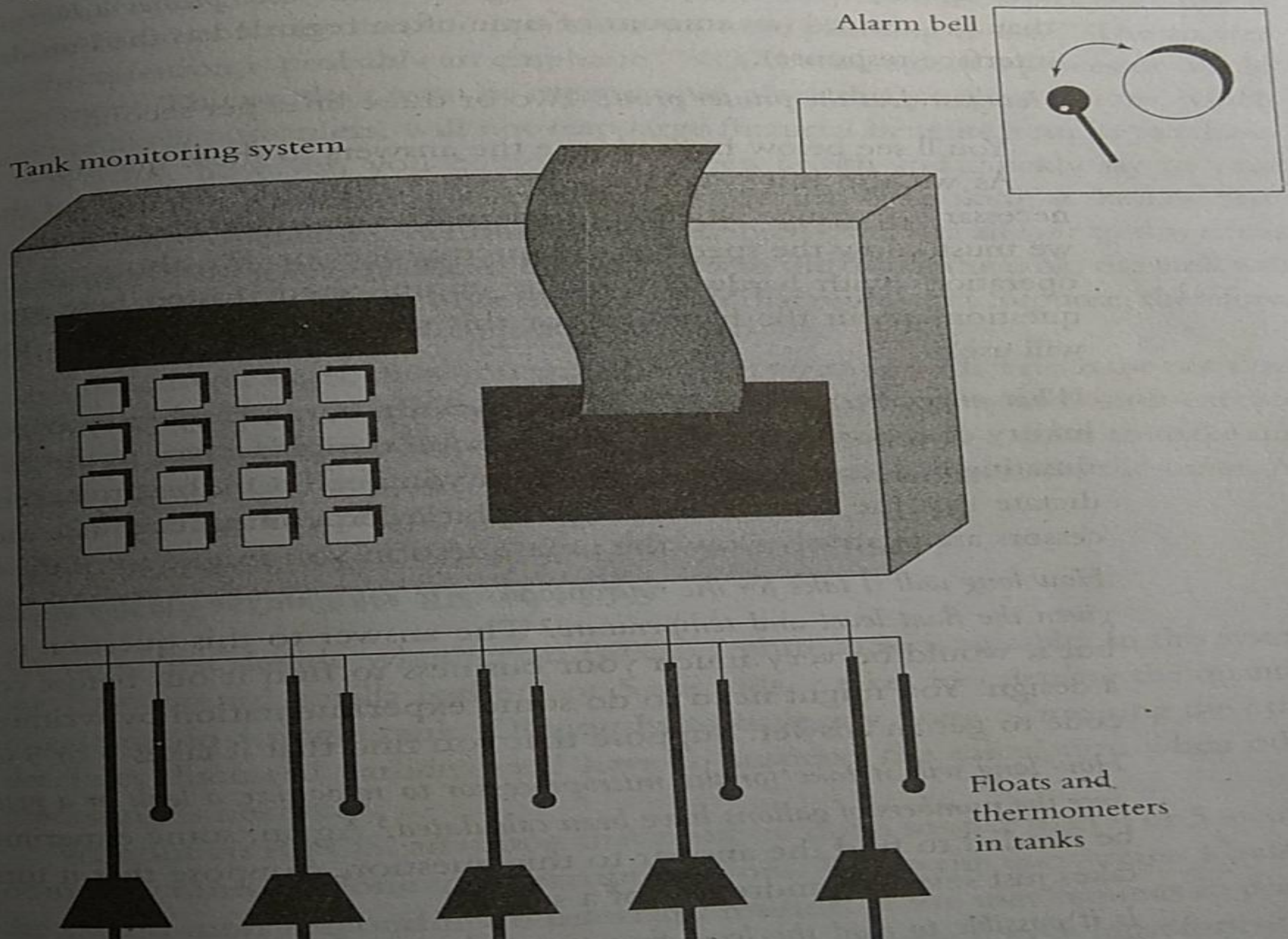


Figure 9.1 Tool Chain for Building Embedded Software

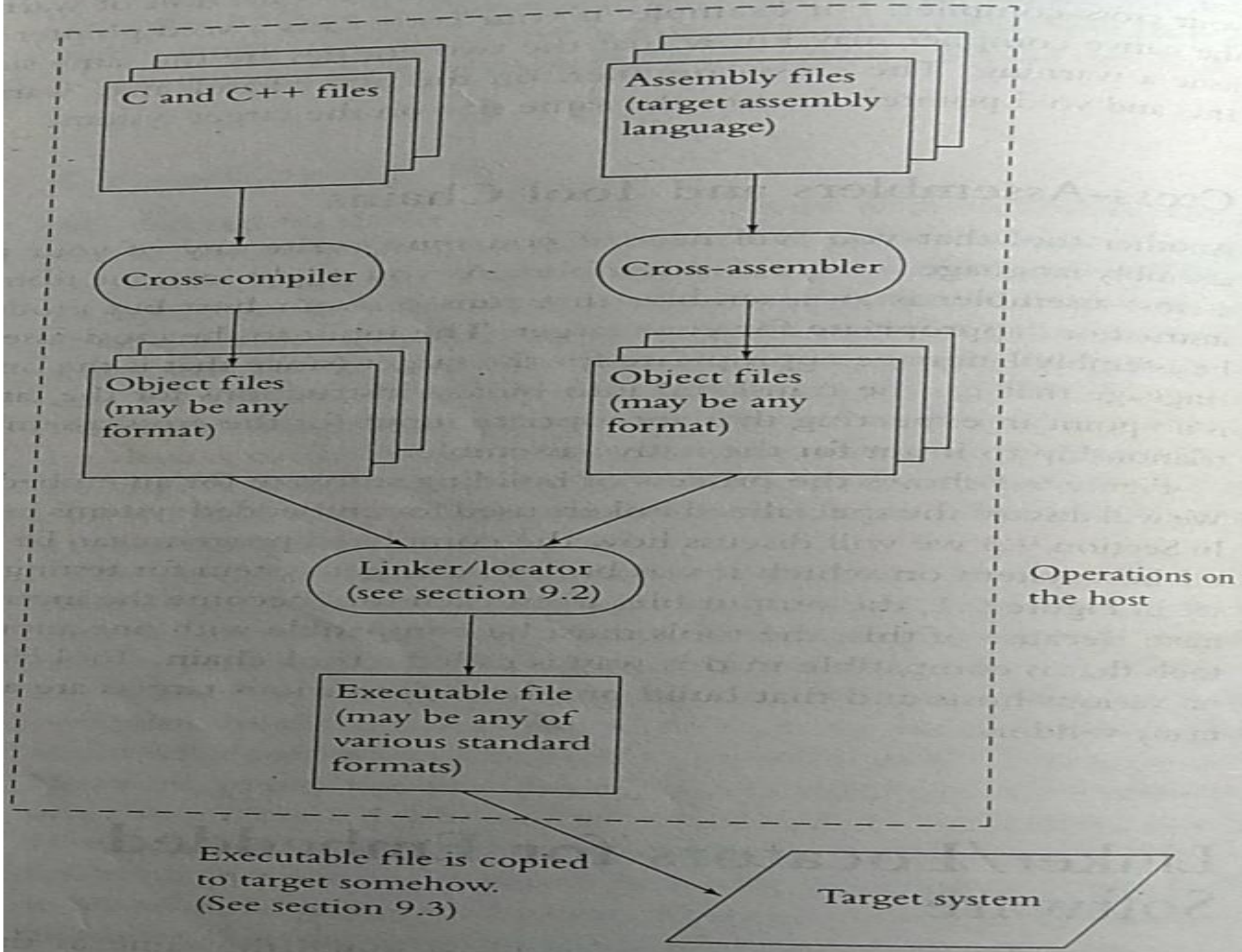

C and C++ files
Assembly files (target assembly language)
Cross-compiler
Cross-assembler
Object files (may be any format)
Object files (may be any format)
Linker/locator (see section 9.2)
Operations on the host
Executable file (may be any of various standard formats)
Executable file is copied to target somehow. (See section 9.3)
Target system

Figure 9.9 ROM Emulator

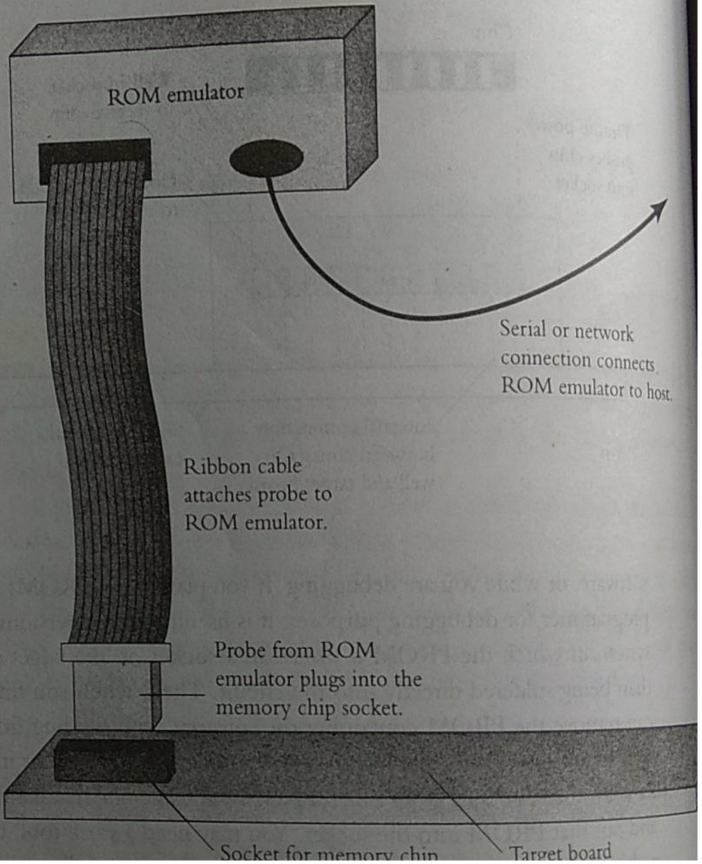

ROM emulator
Serial or network connection connects ROM emulator to host.
Ribbon cable attaches probe to ROM emulator.
Probe from ROM emulator plugs into the memory chip socket.
Socket for memory chip
Target board

Figure 9.8 Schematic Edge View of a Socket

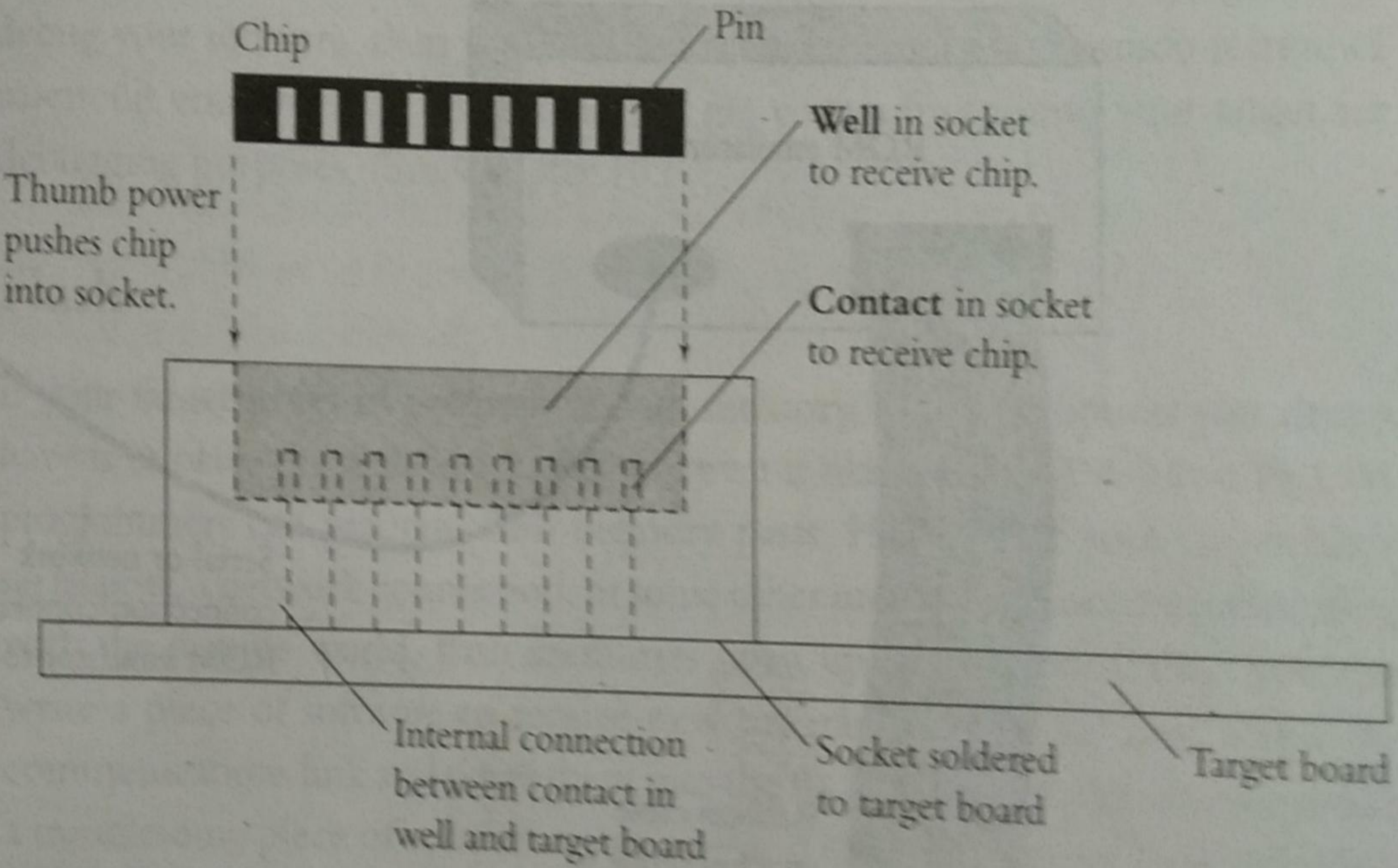

Chip
Pin
Well in socket to receive chip.
Thumb power pushes chip into socket.
Contact in socket to receive chip.
Internal connection between contact in well and target board
Socket soldered to target board
Target board

Figure 9.9 ROM Emulator

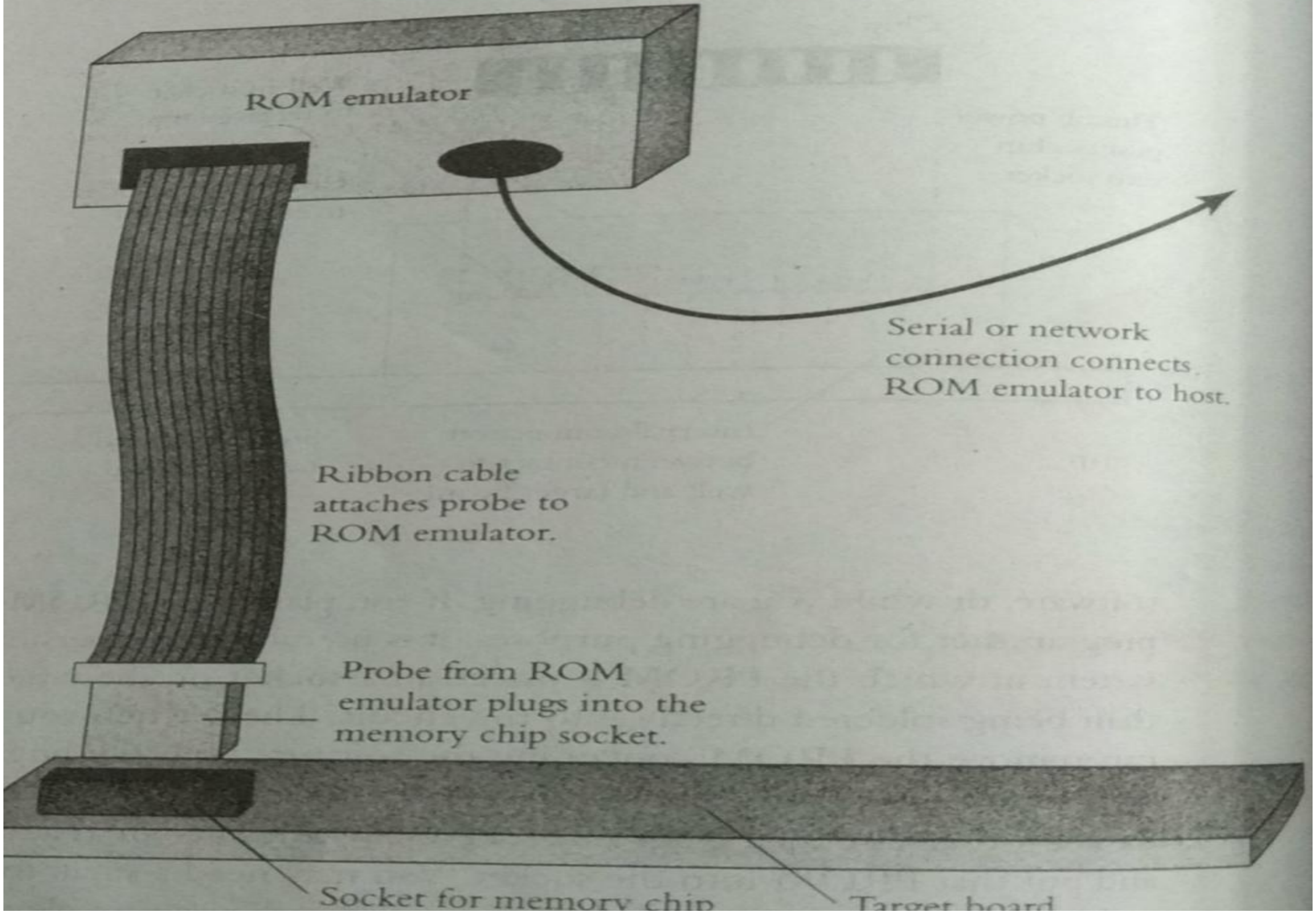

ROM emulator
Serial or network connection connects ROM emulator to host.
Ribbon cable attaches probe to ROM emulator.
Probe from ROM emulator plugs into the memory chip socket.
Socket for memory chip
Target board

Figure 10.1 Test System

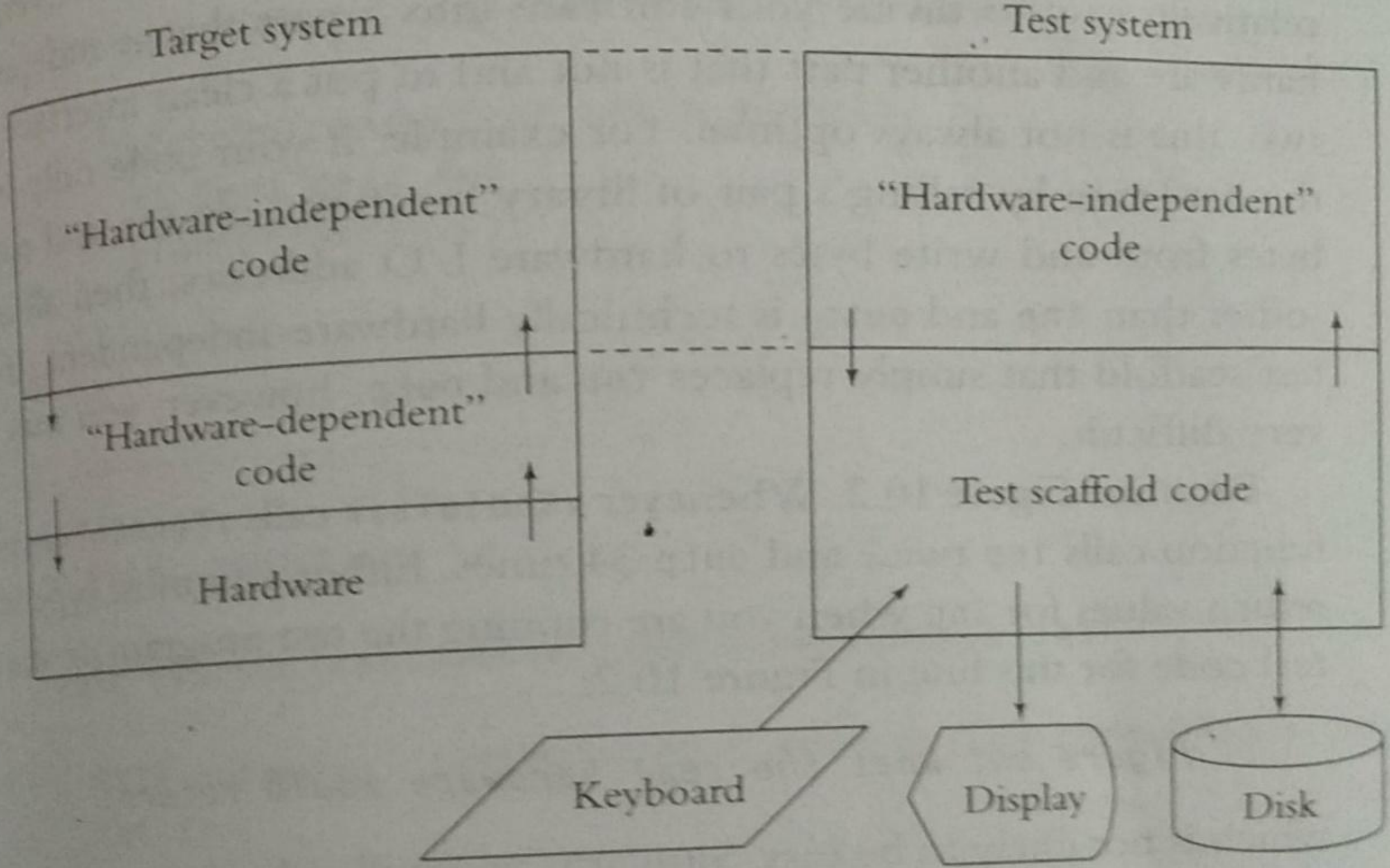
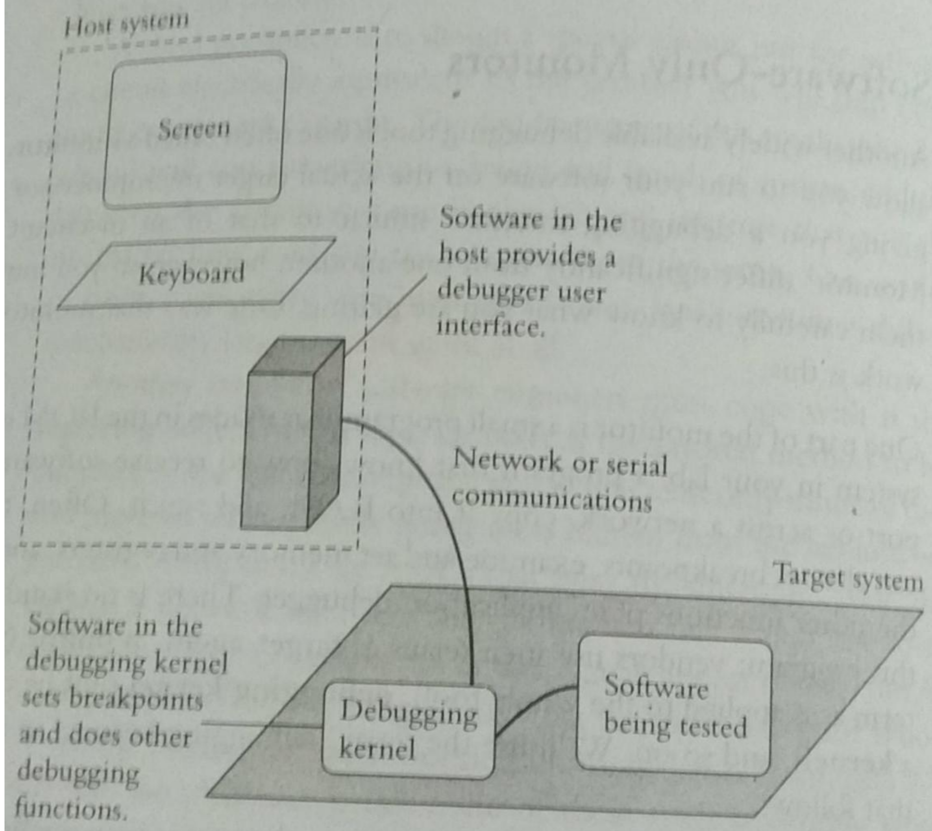

Target system
Test system
"Hardware-independent" code
"Hardware-dependent" code
Hardware
"Hardware-independent" code
Test scaffold code
Keyboard
Display
Disk

Figure 10.19 Software-Only Monitors


Host system
Screen
Keyboard
Software in the host provides a debugger user interface.
Network or serial communications
Target system
Software in the debugging kernel sets breakpoints and does other debugging functions.
Debugging kernel
Software being tested