Q.2 a. Mention the characteristics of embedded systems.

Answer:

Embedded systems have several common characteristics:

1) *Single-functioned:* An embedded system usually executes only one program, repeatedly. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.1

2) *Tightly constrained:* All computing systems have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

3) *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 delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

b. Illustrate with a diagram the working of a single-purpose processor.

Answer:

A *single-purpose* processor is a digital circuit designed to execute exactly one program. For example, consider a digital camera. All of the components of a digital camera other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An embedded system designer creates a single-purpose processor by designing accustom digital circuit, as discussed in later chapters. Many people refer to this portion of the implementation simply as the "hardware" portion (although even software requires a hardware processor on which to run). Other common terms include coprocessor and accelerator. Using a single-purpose processor in an embedded system may result in several design metric benefits and drawbacks, which are essentially the inverse of those for general purpose processors. Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications. The use of a single-purpose processor in any embedded system represents an exact fit of the desired functionality, nothing more, and nothing less. Figure 1 illustrates the architecture of such a single-purpose processor for the

example. Since the example counts from one to N, we add an *index* register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labelled *total* to the datapath. Since the example's only arithmetic operation is addition, we add a single adder to the datapath. Since the processor only executes this one program, we hardwire the program directly into the control logic.

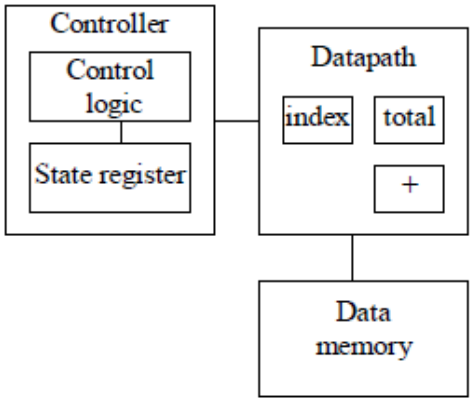


Figure1. Implementing desired functionality on single-purpose processors

c. Discuss the optimizing design metrics in the design of embedded systems.

Answer:

The embedded-system designer must of course construct an implementation that fulfils desired functionality, but a difficult challenge is to construct an implementation that simultaneously optimizes numerous design metrics. For our purposes, an implementation consists of a software processor with an accompanying program, a connection of digital gates, or some combination thereof. A design metric is a measurable feature of a system's implementation. Common relevant metrics include:

i. *Unit cost:* the monetary cost of manufacturing each copy of the system, excluding NRE cost.

ii. *NRE cost (Non-Recurring Engineering cost):* The monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost (hence the term "non-recurring").

iii. *Size:* the physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.

iv. *Performance:* the execution time or throughput of the system.

v. *Power:* the amount of power consumed by the system, which determines the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.

vi. *Flexibility:* the ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.

vii. *Time-to-market:* The amount of time required to design and manufacture the system to the point the system can be sold to customers.

viii. *Time-to-prototype:* The amount of time to build a working version of the system, which may be bigger or more expensive than the final system implementation, but can be used to verify the system's usefulness and correctness and to refine the system's functionality.

ix. *Correctness:* our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.

x. *Safety:* the probability that the system will not cause harm.

Q.3 a. Write a program in assembly language to clear an array M[256]. Assuming M starts at location 256 (and thus ends at location 511).Write the assumptions for the assembly language code.

Answer:

```
Assumption  R0,R1,R3 are 8 bit registers of the processor.
MOV R0, #0;
 MOV R1, #256; // no. of location is256 and it is unsigned 8-bit.
MOV R3, #256; //Array M starts at memory location 256
Loop: JZ R1, Stop; // Done if R1=0
MOV @R3, R0; //load 0 in M[i]where i=(256,511)
INC R3;// R3 is incremented by 1 so points to next memory address
DEC R1;// R1 is decremented by 1 as the counter should be decremented after
each
        //operation.
JMP  Loop; // Jump always
Stop:End
```

b. What are the different stages of execution of instructions?

Answer:

A microprocessor's execution of instructions consists of several basic stages:

1. *Fetch instruction*: the task of reading the next instruction from memory into the instruction register.
2. *Decode instruction*: the task of determining what operation the instruction in the instruction register represents (e.g., add, move, etc.).
3. *Fetch operands*: the task of moving the instruction's operand data into appropriate registers.
4. *Execute operation*: the task of feeding the appropriate registers through the ALU and back into an appropriate register.

5. *Store results*: the task of writing a register into memory.
If each stage takes one clock cycle, then we can see that a single instruction may take several cycles to complete.

c. Write the benefits of choosing a single purpose processor over a general purpose processor.

Answer:

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. The manufacturer 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 designer choosing to use a standard 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, since the processor is customized for the particular 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. There are of course tradeoffs. If we are already using a general-purpose processor, then implementing a task on an additional single-purpose processor rather than in software may add to the system size and power consumption. We often refer to standard single-purpose processors as peripherals, because they usually exist on the periphery of the CPU. However, microcontrollers tightly integrate these peripherals with the CPU, often placing them on-chip, and even assigning peripheral registers to the CPU's own register space. The result is the common term "on chip peripherals,"

Q.4 a. A particular motor operates at 10 revolutions per second when its controlling input voltage is 3.7 V. Assume that you are using a microcontroller with a PWM whose output port can be set high (5 V) or low (0 V). (i) Compute the duty cycle necessary to obtain 10 revolutions per second. (ii) Provide values for a pulse width and period that achieve this duty cycle.

Answer:

The duty cycle is the percentage of time the signal is high compared to the signal's period.
Given: Input voltage is 3.7 V. The output port of the PWM can be set high (5 V) or low (0 V).

(a) 3.7V / 5V = .74 = 74% duty cycle
(b) There are infinitely many answers. Example:period = 100 ns (pick any reasonable value)pulse width = .74 * period = .74 * 100 ns = 74 ns

b. Describe the working of an UART.

Answer:

A *UART* (Universal Asynchronous Receiver/Transmitter) receives serial data and 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 few available I/O pins. For our purpose in this section, 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 be transmitted 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:
$Baudrate = (2^{smod} / 32) * oscfreq / (12 * (256 - TH1))$
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 of a 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.

Q.5 a. Write any two cache replacement policies.

Answer:

The *cache-replacement policy* is the technique for choosing which cache block to replace when a fully-associative cache is full, or when a set-associative cache's line is full. Note that there is no choice in a direct-mapped cache; a main memory address always maps to the same cache address and thus replaces whatever block is already there.
There are three common replacement policies.
i. A *random* replacement policy chooses the block to replace randomly. While simple to implement, this policy does nothing to prevent replacing block that's likely to be used again soon.
ii. A *least-recently used (LRU)* replacement policy replaces the block that has not been accessed for the longest time, assuming that this means that it is least likely to be accessed in the near future. It associates with each page the time of its last use. It does not suffer from Belady's anomaly. This policy provides for an excellent hit/miss ratio but requires expensive hardware to keep track of the times blocks are accessed.
iii. A *first-in-first-out (FIFO)* replacement policy uses a queue of size N, pushing each block address onto the queue when the address is accessed, and then choosing the block to replace by popping the queue. This policy is easy to understand and implement; but the performance is poor. In FIFO, it might happen that adding more pages causes more fault.This phenomenon is known as Belady's anomaly,

b. From the given following three cache designs, find the one with the best performance by calculating the average cost of access. Show all calculations.

(i) 4 Kbyte, 8-way set-associative cache with a 6% miss rate; cache hit costs one cycle, cache miss costs 12 cycles
(ii) 8 Kbyte, 4-way set-associative cache with a 4% miss rate; cache hit costs two cycles, cache miss costs 12 cycles.
(iii) 16 Kbyte, 2-way set-associative cache with a 2% miss rate; cache hit costs three cycles, cache miss costs 12 cycles.

Answer:

i) 4 Kb, 8-way set-associative cache with a 6% miss rate; cache hit costs 1 cycle,cache miss costs 12 cycles.miss rate = .06hit rate = 1- miss rate = .94.94 * 1cycle (hit) + .06 * 12 cycles (miss) = .94 + .72 = 1.66 cycles avg.

ii.) 8 Kb, 4-way set-associative cache with a 4% miss rate; cache hit costs 2 cycles,cache miss costs 12 cycles.miss rate = .04hit rate = 1 – miss rate = .96.96 * 2 cycles (hit) + .04 * 12 cycles (miss) = 1.92 + .48 = 2.4 cycles avg.

iii.) 16 Kb, 2-way set-associative cache with a 2% miss rate; cache hit costs 3 cycles,cache miss costs 12 cycles.miss rate = .02hit rate = 1 – miss rate = .98.98 * 3 cycles (hit) + .02 * 12 cycles (miss) = 2.94 + .24 = 3.18 cycles avg.

c. What is DRAM?

Answer:

Dynamic random access memory (DRAM) is a type of random access memory that stores each bit of data in a separate capacitor within an integrated circuit. Since real capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a dynamic memory as opposed to SRAM and other static memory. DRAM is usually arranged in a rectangular array of charge storage cells consisting of one capacitor and transistor per data bit. The figure to the right shows a simple example with a 4 by 4 cell matrix. Modern DRAM matrices are many thousands of cells in height and width. The long horizontal lines connecting each row are known as word-lines. Each column of cells is composed of two bit-lines, each connected to every other storage cell in the column.

The main memory (the "RAM") in personal computers is Dynamic RAM (DRAM), as is the "RAM" of home game consoles (PlayStation, Xbox 360), laptop, notebook and workstation computers. The advantage of DRAM is its structural simplicity: only one transistor and a capacitor are required per bit, compared to six transistors in SRAM. This allows DRAM to reach very high densities. Unlike flash memory, it is volatile memory (cf. non-volatile memory), since it loses its data when power is removed. The transistors and capacitors used are extremely small—millions can fit on a single memory chip.

Q.6 a. Illustrate the functioning of two protocol control methods:
(i) Strobe
(ii) Handshake

Answer:

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 one control 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 2(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 de-asserts 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 uses a 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 2(b) shows a handshake protocol with a 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 the data bus and then de-asserts the request line so that the servant can stop putting data on the data bus. The servant de-asserts the acknowledge line, and both actors are then ready for the next 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 servants) with varying response times, unlike a strobe protocol. However, when response time 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 clock cycle if the master is synchronizing the bus control signals. A handshake also requires an extra line for acknowledge.

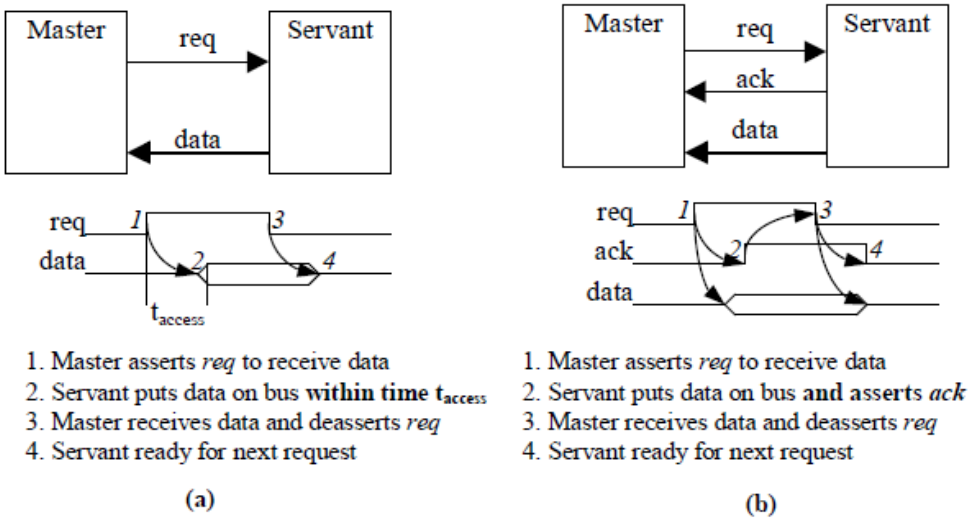


Figure2. Two protocol control methods: (a) strobe, (b) handshake

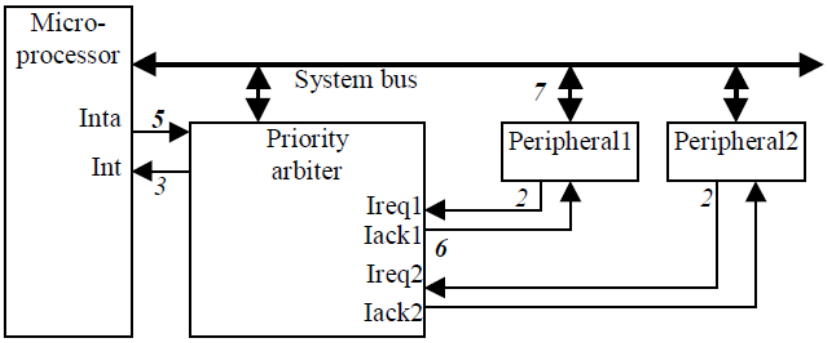
b. Discuss the steps of Arbitration using a priority arbiter.

Answer:

One arbitration method uses a single-purpose processor, called a priority arbiter. We illustrate a priority arbiter arbitrating among multiple peripherals using vectored interrupt to request servicing from a microprocessor, as illustrated in Figure 3. Each of the peripherals makes its request to the arbiter. The arbiter in turn asserts the microprocessor interrupt, and waits for the interrupt acknowledgment. The arbiter then provides an acknowledgement to exactly one peripheral, which permits that peripheral to put its interrupt vector address on the data bus (which, as you'll recall, causes the microprocessor to jump to a subroutine that services that peripheral). Priority arbiters typically use one of two common schemes to determine priority among peripherals: fixed priority or rotating priority. In *fixed priority* arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as a number, so if there are four peripherals, each

peripheral is ranked 1, 2, 3 or 4. If two peripherals simultaneously seek servicing, the arbiter chooses the one with the higher rank.

In *rotating priority* arbitration (also called round-robin), the arbiter changes priority of peripherals based on the history of servicing of those peripherals. For example, one rotating priority scheme grants service to the least-recently serviced of the contending peripherals. This scheme obviously requires a more complex arbiter. We prefer fixed priority when there is a clear difference in priority among peripherals. However, in many cases the peripherals are somewhat equal, so arbitrarily ranking them could cause high-ranked peripherals to get much more servicing than low-ranked ones. Rotating priority ensures a more equitable distribution of servicing in this case.



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. **Microprocessor asserts *Inta*.**
6. **Priority arbiter asserts *Iack1* to acknowledge Peripheral1.**
7. **Peripheral1 puts its interrupt address vector on the system bus**
8. **Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).**
9. Microprocessor resumes executing its program.

Figure3. Arbitration using a priority arbiter.

c. What are Interrupts? What is an ISR? How is it invoked?

Answer:

Suppose a program running on a microprocessor must, among other tasks, read and process data from a peripheral whenever that peripheral has new data; such processing is called *servicing*. If the peripheral gets new data at unpredictable intervals, then how can the program determine when the peripheral has new data? The most straightforward approach is to interleave the microprocessor's other tasks with a routine that checks for new data in the peripheral, perhaps by checking for a 1 in a particular bit in a register of the peripheral. This repeated checking by the microprocessor for data is called *polling*. Polling is simple to implement, but this repeated checking wastes many clock cycles, so may not be acceptable in many cases, especially when there are numerous peripherals to be checked. We could check at less-frequent intervals, but then we may not process the data quickly enough.

To overcome the limitations of polling, most microprocessors come with a feature called *external interrupt*. A microprocessor with this feature has a pin, say *Int*. At the end of executing each machine instruction, the processor's controller checks *Int*. If *Int* is asserted, he microprocessor jumps to a particular address at which a subroutine exists that services the interrupt. This subroutine is called an *Interrupt Service Routine*, or *ISR*. Such I/O is called *interrupt-driven I/O*. One might wonder if interrupts have really solved the problem with polling, namely of wasting time performing excessive checking, since the interrupt pin is "polled" at the end of every microprocessor instruction. However, in this case, the polling of the pin is built right into the microprocessor's controller hardware, and therefore can be done simultaneously with the execution of an instruction, resulting in no extra clock cycles. There are two methods by which a microprocessor using interrupts determines the address, known as the *interrupt address vector*, at which the ISR resides. In some processors, the address to which the microprocessor jumps on an interrupt is *fixed*. The assembly programmer either puts the ISR there, or if not enough bytes are available in that region of memory, merely puts a jump to the real ISR there.

Q.7 a. Give an example to use Semaphore as a signalling device.

Answer: Refer article 6.3, pages 162-163 from Text-Book- II

b. What are the problems of shared data and how are they removed?

Answer: Refer article 6.3, pages 147, 148, 153-155 from Text-Book- II

c. Explain the function of a scheduler.

Answer:

Scheduling is the process of controlling and prioritizing messages sent to a processor. An internal operating system program, called the scheduler, performs this task. The goal is maintaining a constant amount of work for the processor, eliminating highs and lows in the workload and making sure each process is completed within a reasonable time frame. While scheduling is important to all systems, it is especially important in a real-time system.

Since nearly every operation on a computer has at least a small amount of processor time involved, the processor can be a major source of slowdowns and bottlenecks. In order to alleviate the strain on the processor, and make sure tasks are completed in a timely manner, most operating systems use some form of task scheduling. The operating system scheduling process varies based on the system, but they tend to fall within familiar categories.

Scheduling is typically broken down into three parts: long-, mid- and short-term scheduling. Not every operating system fully uses each type — midterm and long-term are often combined — but they will use some combination of them. Each type of scheduling provides a slightly different benefit to the system.

Long-term scheduling revolves around admitting programs to the scheduling process. When a new program initiates, the long-term scheduler determines if there is enough space for the new entrant. If there isn't, then the scheduler delays the activation of the program until there is enough room.

Q.8 a. What is a pipe?

Answer: Refer article 7.1 page 181 from Text Book- II

b. Discuss the rules to be followed by Interrupt Routines in an RTOS environment.

Answer: Refer article 7.5, Pages 199-202 from Text Book- II

c. Write a pseudocode to delay a task by using RTOS delay Function.

Answer: Refer article 7.2, page 185 from Text-Bok-II

Q.9 a. What is hard real time scheduling?

Answer:

A real-time system is one that must process information and produce a response within a specified time, else risk severe consequences, including failure. That is, in a system with a real-time constraint it is no good to have the correct action or the correct answer ***after*** a certain deadline. A system can be defined to be a *hard* real-time system if the damage has the potential to be catastrophic (i.e. where the consequences are incommensurably greater than any benefits provided by the service being delivered in the absence of failure like (aircraft crashing, car skidding, patient dying before corrective action is performed)).

b. What do you mean by encapsulating semaphore? Write a program to show how a semaphore is encapsulated.

Answer: II(8.4 page 244-247)

c. Discuss briefly about the ways by which memory space can be conserved.

Answer: II(8.6, page 254-255)

TEXT BOOK

I. Systems Analysis and Design Methods, Jeffrey L Whitten, Lonnie D Bentley, Seventh Edition, TMH, 2007