Q2 (a) 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,"

Q2 (b) List the hardware units that must be present in the embedded systems.

Answer

The hardware units available in an embedded systems are power source, clock oscillator circuit & clocking unit, real time clock & timer, reset circuit, power up reset, watchdog timer reset, Input, output, I/O ports, buses & interfaces, memories, DAC & ADC, interrupt handler, keypad/keyboard, pulse dialer, modem, transceiver, LCD & LED displays, GPIB link, linking and interface buses & units. In power souse we must oncentrate on power consumption and power dissipation. In clock circuit any one of the following is used that is crystal or ceramic or IC based clock. Memory is an important part of any embedded system design and is heavily influenced by the software design.

Q3 (a) Give a detailed description on the basic architecture of a general purpos processor. Give suitable diagrams also.

Answer

8085 consists of various units and each unit performs its own functions. The various units of a microprocessor are listed below

· Accumulator
· Arithmetic and logic Unit
· General purpose register
· Program counter
· Stack pointer
· Temporary register
· Flags
· Instruction register and Decoder
· Timing and Control unit
· Interrupt control
· Serial Input/output control
· Address buffer and Address-Data buffer
· Address bus and Data bus

Accumulator

Accumulator is nothing but a register which can hold 8-bit data. Accumulator aids in storing two quantities.The data to be processed by arithmetic and logic unit is stored in accumulator.It also stores the result of the operation carried out by the Arithmetic and Logic unit.

The accumulator is also called an 8-bit register. The accumulator is connected to Internal Data bus and ALU (arithmetic and logic unit). The accumulator can be used to send or receive data from the Internal Data bus.

Arithmetic and Logic Unit

There is always a need to perform arithmetic operations like +, -, *, / and to perform logical operations like AND, OR, NOT etc. So there is a necessity for creating a separate unit which can perform such types of operations. These operations are performed by the Arithmetic and Logic Unit (ALU). ALU performs these operations on 8-bit data.

But these operations cannot be performed unless we have an input (or) data on which the desired operation is to be performed. So from where do these inputs reach the ALU? For this purpose accumulator is used. ALU gets its Input from accumulator and temporary register. After processing the necessary operations, the result is stored back in accumulator.

General Purpose Registers

Apart from accumulator 8085 consists of six special types of registers called General Purpose Registers.These general purpose registers are used to hold data like any other registers. The general purpose registers in 8085 processors are B, C, D, E, H and L. Each register can hold 8-bit data. Apart from the above function these registers can also be used to work in pairs to hold 16-bit data.They can work in pairs such as B-C, D-E and H-L to store 16-bit data. The H-L pair works as a memory pointer.A memory pointer holds the address of a particular memory location. They can store 16-bit address as they work in pair.

Program Counter and Stack Pointer

Program counter is a special purpose register.

Consider that an instruction is being executed by processor. As soon as the ALU finished executing the instruction, the processor looks for the next instruction to be executed. So, there is a necessity for holding the address of the next instruction to be executed in order to save time. This is taken care by the program counter.

A program counter stores the address of the next instruction to be executed. In other words the program counter keeps track of the memory address of the instructions that are being executed by the microprocessor and the memory address of the next instruction that is going to be executed.

Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed. Program counter is a 16-bit register.

Stack pointer is also a 16-bit register which is used as a memory pointer. A stack is nothing but the portion of RAM (Random access memory).

Stack pointer maintains the address of the last byte that is entered into stack.

Each time when the data is loaded into stack, Stack pointer gets decremented. Conversely it is incremented when data is retrieved from stack.

Temporary Register

As the name suggests this register acts as a temporary memory during the arithmetic and logical operations. Unlike other registers, this temporary register can only be accessed by the microprocessor and it is completely inaccessible to programmers. Temporary register is an 8-bit register.

Q3 (b) Explain the following terms:-

(i) SOC

(ii) Device programmer

(iii) ASIP

Answer

(i) System-on-a-chip or system on chip (SoC or SOC) refers to integrating all components of a computer or other electronic system into a single integrated circuit (chip). It may contain digital, analog, mixed-signal, and often radio-frequency functions – all on a single chip substrate. A typical application is in the area of embedded systems.

The contrast with a microcontroller is one of degree. Microcontrollers typically have under 100K of RAM (often just a few KBytes) and often really are single-chip-systems; whereas the term SoC is typically used with more powerful processors, capable of running software such as Windows or Linux, which need external memory chips (flash, RAM) to be useful, and which are used with various external peripherals. In short, for larger systems System-on-a-chip is hyperbole, indicating technical direction more than reality: increasing chip integration to reduce manufacturing costs and to enable smaller systems. Many interesting systems are too complex to fit on just one chip built with a process optimized for just one of the system's tasks.

When it is not feasible to construct an SoC for a particular application, an alternative is system in package (SiP) comprising a number of chips in a single package. In large volumes, SoC is believed to be more cost effective than SiP since it increases the yield of the fabrication and because its packaging is simpler.

Another option, as seen for example in higher end cell phones and on the Beagle Board, is package on package stacking during board assembly. The SoC chip includes processors and numerous digital peripherals, and comes in a ball grid package with lower and upper connections. The lower balls connect to the board and various peripherals, with the upper balls in a ring holding the memory busses used to access NAND flash and DDR2 RAM. Memory packages could come from multiple vendors.

A SoC consists of both the hardware and the software that controls the microcontroller, microprocessor or DSP cores, peripherals and interfaces. The design flow for an SoC aims to develop this hardware and software in parallel.

Most SoCs are developed from pre-qualified hardware blocks for the hardware elements described above, together with the software drivers that control their operation. Of particular importance are the protocol stacks that drive industry-standard interfaces like USB. The hardware blocks are put together using CAD tools; the software modules are integrated using a software development environment.

A key step in the design flow is emulation: the hardware is mapped onto an emulation platform based on a field programmable gate array (FPGA) that mimics the behavior of the SoC, and the software modules are loaded into the memory of the emulation platform. Once programmed, the emulation platform enables the hardware and software of the SoC to be tested and debugged at close to its full operational speed.

After emulation the hardware of the SoC follows the place and route phase of the design of an integrated circuit before it is fabricated.

Chips are verified for logical correctness before being sent to foundry. This process is called functional verification, and it accounts for a significant portion of the time and energy expended in the chip design life cycle. Verilog and VHDL are typical hardware description languages used for verification. With the growing complexity of chips, hardware verification languages like System Verilog, SystemC, e, and OpenVera are also being used. Bugs found in the verification stage are reported to the designer.

(ii) For programming a circuit, it is either inserted into a socket on top of the programmer, or the programmer is directly connected by an adapter to the circuit board (In-System Programming). Afterwards the data is transferred into the circuit by applying signals to the connecting pins. Some circuits have a serial interface for receiving the programming data (JTAG interface). Other circuits require the data on parallel pins, followed by a programming pulse with a higher voltage for programming the data into the circuit.

Usually device programmers are connected to a personal computer through a printer connector, USB port or LAN interface. A software program on the computer then transfers the data to the programmer, selects the circuit and interface type, and starts the programming process.

There are four general types of device programmers: Gang programmers for mass production, development programmers for development and small-series production, pocket programmers for development and field service, and specialized programmers for certain circuit types only, f.i. EPROM programmers. Early device programmers had the size of a shoe box and a weight of up to 4 kg; the latest generation device programmers are pocket sized, weigh less than 200 g and require no external power supply. These types of programmers can be used in field service for maintenance or setup of machinery that contains programmable circuits.

A challenge for device programmer manufacturers is the design of the pin drivers that are directly connected to the circuit to be programmed. Due to the many different programmable circuits, every pin driver must be able to apply different voltages in a range of 0-25 Volts, clock rates of up to 40 MHz, and logic inputs with adjustable threshold. Modern programmers use a dedicated integrated circuit for the pin drivers.

In the early days of computing, before terminal and graphical display devices, a programmer was a device used to configure a program for a computer. It usually consisted of switches and LEDs, where instructions had to be entered one by one by setting the switches in a series of "on" and "off" positions. The positions of the switches corresponded to computer instructions, similar to how assembly language is used today. Such hardware programmers are almost never seen or used today.

(iii) An **application-specific instruction-set processor** (**ASIP**) is a component used in system-on-a-chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC.

Some ASIPs have a configurable instruction set. Usually, these cores are divided into two parts: *static* logic which defines a minimum ISA and *configurable* logic which can be used to design new instructions. The configurable logic can be programmed either in the field in a similar fashion to an FPGA or during the chip synthesis.

Field programmable gate array (FPGA) is a programmable logic array in which internal connections of logic blocks can be programmed in the field to realize the desired digital circuit. FPGA provides the system designer with a great deal of flexibility and is an excellent alternative to standard SSI, MSI, and VLSI logic devices.

They combine the flexibility of mask programmable gate arrays with the convenience of field programmability.These features make it possible to combine many portions of discrete logic, otherwise available on multichips, in to a single FPGA device. The user

programmable feature makes it possible to develop application specific instructio processor (ASIP) with comfortable ease.

The complexity and sophistication of data handling and control of ASIP execution make tools that can analyse sequential applications and derive user specific design implementations extremely desirable. A complier and synthesis system is used to analyse the input sequential code and partition of the data and computation among FPGA architectural blocks for ASIP execution. The compiler analyses the set of design stages and schedules them on to the target architecture, respecting the original data dependencies and the target architecture's FPGA and memory capacity constraints.

Q3 (c) Explain briefly pipelining.

Answer Page Number 60 of Text Book I

Q4 (a) Tabulate the uses of Timer device with applications and explanations.

Answer

An on-delay timer will wait for a set time after a line of ladder logic has been true before turning on, but it will turn off immediately. An off-delay timer will turn on immediately when a line of ladder logic is true, but it will delay before turning off. Consider the example of an old car. If you turn the key in the ignition and the car does not start immediately, that is an on-delay. If you turn the key to stop the engine but the engine doesn't stop for a few seconds that is an off delay. An on-delay timer can be used to allow an oven to reach temperature before starting production. An off delay timer can keep cooling fans on for a set time after the oven has been turned off. A retentive timer will sum all of the on or off time for a timer, even if the timer never finished. A no retentive timer will start timing the delay from zero each time. Typical applications for retentive timers include tracking the time before maintenance is needed. A non retentive timer can be used for a start button to give a short delay before a conveyor begins moving.

A timer with automatic reload capability will have a latch register to hold the count written by the processor. When the processor writes to the latch, the count register is written as well. When the timer later overflows, it first generates an output signal. Then, it automatically reloads the contents of the latch into the count register. Since the latch still holds the value written by the processor, the counter will begin counting again from the same initial value.

Such a timer will produce a regular output with the same accuracy as the input clock. This output could be used to generate a periodic interrupt like a real-time operating system (RTOS) timer tick, provide a baud rate clock to a UART, or drive any device that requires a regular pulse.

A variation of this feature found in some timers uses the value written by the processor as the endpoint rather than the initial count. In this case, the processor writes into a terminal count register that is constantly compared with the value in the count register. The count register is always reset to zero and counts up. When it equals the value in the terminal count register, the output signal is asserted. Then the count register is reset to zero and the

process repeats. The terminal count remains the same. The overall effect is the same as a overflow counter. A periodic signal of a pre-determined length will then be produced.

If a timer supports automatic reloading, it will often make this a software-selectable feature. To distinguish between a count that will not repeat automatically and one that will, the hardware is said to be in one of two modes: one-shot or periodic. The mode is generally controlled by a field in the timer's control register. An input capture timer, has a latch connected to the timer's count register. The timer is run at a constant clock rate (usually a derivative of the processor clock), so that the count register is constantly incrementing (or decrementing, for a down counter). An external signal latches the value of the free-running timer into the processor-visible register and generates an output signal (typically an interrupt). One use for an input capture timer is to measure the time between the leading edge of two pulses. By reading the value currently in the latch and comparing it with a previous reading, the software can determine how many clock cycles elapsed between the two pulses. In some cases, the timer's count register might be automatically reset just after its value is latched. If so, the software can directly interpret the value it reads as the number of clock ticks elapsed. An input capture pin can usually be programmed to capture on either the rising or falling edge of the input signal.

Q4 (b) How will you set watchdog timer to restart the processor at every 2 ms?

Answer

```
main(void)
{
   hwinit();

   for (;;)
   {
      *pWatchdog = 200;
      read_sensors();
      control_motor();
      display_status();
   }
}
//This is according to 2 msec logic to be created.
```

Q4 (c) Explain the principle and working of UART with a suitable diagram.

Answer

The Universal Asynchronous Receiver/Transmitter (UART) takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms. Serial

transmission of digital information (bits) through a single wire or other medium is muc more cost effective than parallel transmission through multiple wires.

The UART usually does not directly generate or receive the external signals used between different items of equipment. Separate interface devices are used to convert the logic level signals of the UART to and from the external signaling levels. External signals may be of many different forms. Examples of standards for voltage signaling are RS-232, RS-422 and RS-485 from the EIA. Historically, the presence or absence of current (in current loops) was used in telegraph circuits. Some signaling schemes do not use electrical wires. Examples of such are optical fiber, IrDA (infrared), and (wireless) Bluetooth in its Serial Port Profile (SPP). Some signaling schemes use modulation of a carrier signal (with or without wires). Examples are modulation of audio signals with phone line modems, RF modulation with data radios, and the DC-LIN for power line communication.

Communication may be "full duplex" (both send and receive at the same time) or "half duplex" (devices take turns transmitting and receiving).

Receiver:

All operations of the UART hardware are controlled by a clock signal which runs at a multiple (say, 16) of the data rate - each data bit is as long as 16 clock pulses. The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit. If the apparent start bit lasts at least one-half of the bit time, it is valid and signals the start of a new character. If not, the spurious pulse is ignored. After waiting a further bit time, the state of the line is again sampled and the resulting level clocked into a shift register. After the required number of bit periods for the character length (5 to 8 bits, typically) have elapsed, the contents of the shift register is made available (in parallel fashion) to the receiving system. The UART will set a flag indicating new data is available, and may also generate a processor interrupt to request that the host processor transfers the received data. In some common types of UART, a small first-in, first-out FIFO buffer memory is inserted between the receiver shift register and the host system interface. This allows the host processor more time to handle an interrupt from the UART and prevents loss of received data at high rates.

Transmitter:

Transmission operation is simpler since it is under the control of the transmitting system. As soon as data is deposited in the shift register after completion of the previous character, the UART hardware generates a start bit, shifts the required number of data bits out to the line, generates and appends the parity bit (if used), and appends the stop bits. Since transmission of a single character may take a long time relative to CPU speeds, the UART will maintain a flag showing busy status so that the host system does not deposit a new character for transmission until the previous one has been completed; this may also be done with an interrupt. Since full-duplex operation requires characters to be sent and received at the same time, practical UARTs use two different shift registers for transmitted characters and received characters.

Application Transmitting and receiving UARTs must be set for the same bit speed, character length, parity, and stop bits for proper operation. The receiving UART may detect some mismatched settings and set a "framing error" flag bit for the host system; in exceptional cases the receiving UART will produce an erratic stream of mutilated characters and transfer them to the host system.

Typical serial ports used with personal computers connected to modems use eight da bits, no parity, and one stop bit; for this configuration the number of ASCII character per second equals the bit rate divided by 10.

Some very low-cost home computers or embedded systems dispensed with a UART and used the CPU to sample the state of an input port or directly manipulate an output port for data transmission. While very CPU-intensive, since the CPU timing was critical, these schemes avoided the purchase of a costly UART chip. The technique was known as a bit-banging serial port.

Q5 (a) Explain the memory allocation schemes in an embedded system. Also give a short note on extended memory.

Answer

Whether you're using only static memory, a simple stack, or dynamic allocation on a heap, you have to proceed cautiously. Embedded programmers cannot afford to ignore the risks inherent in memory utilization.

Every program uses random access memory (RAM), but the ways in which that memory is divided among the needy parts of the system varies widely. This article surveys the options available in hopes that the reader will be better equipped to choose an approach for a given project.

The mechanisms include statically allocating all memory, using one or more stacks, and using a heap. We will examine how the heap implementation can impact fragmentation and real-time performance.

Static memory allocation

If all memory is allocated statically, then exactly how each byte of RAM will be used during the running of the program can be established at compile time. The advantage of this in embedded systems is that the whole issue of memory-related bugs—due to leaks, failures, and dangling pointers—simply does not exist. Many compilers for 8-bit processors such as the 8051 or PIC are designed to perform static allocation. All data is either global, file static or function static, or local to a function. The global and static data is allocated in a fixed location, since it must remain valid for the life of the program.

The local data is stored in a block set aside for each function. This means that if a function has a local variable x, then x is stored in the same place for every invocation of that function. When the function is not running, that location is usually not used. This approach is used in C compilers when the hardware is not capable of providing suitable support for a stack.

This approach prohibits the use of recursion or any other mechanism that requires reentrant code. For example, an interrupt routine can't call a function that may also be called by the main flow of execution. In return for this loss of flexibility, the programmer is guaranteed no run-time memory allocation issues. It might be useful if all compilers gave the programmer the option of not using the stack. By statically defining all of the space, the programmer sacrifices some flexibility and efficiency, in exchange for extra robustness.

Some clever compilers may establish that two particular functions can't be simultaneously active and, so, allow the memory blocks associated with those two functions to overlap. This approach puts an extra restriction on the code that function pointers can't be used.

To benefit from the inherent memory safety of a completely static environment, it important that the programmer avoid introducing dangers by trying to implement dynamic memory (such as reusing global data for different purposes) on top of the static environment.

For large systems, completely static allocation is not feasible since an enormous amount of RAM would eventually be required to satisfy every possible execution path of the program.

Stack-based memory management

The next step up in complexity is to add a stack. Now a block of memory is required for every call of a function, and not just a single block for each function in existence. The blocks are stored on a stack, and are usually called stack frames.

The stack grows and shrinks as the program executes, and for many programs, it isn't possible to predict, at compile time, what the worst case stack size will be. A multitasking system will have one stack per task (plus possibly an extra one for interrupts). Some judgment must be exercised to make sure that each stack is big enough for all of its activities. It's an awful shame to suffer from an untimely stack overflow especially if one of the other stacks has a reserve of space that it never uses. Unfortunately, most embedded systems do not support any kind of virtual memory management that would allow the tasks to draw from a common pool as the need arises.

One rule of thumb is to make each stack 50% bigger than the worst case seen during testing. In order to apply this rule, the programmer must know how big the stack, or stacks, became during testing. One simple technique is to "paint" the stack space with a simple pattern. As the stack grows and shrinks it will overwrite the area with its data. At a later time, a simple loop can run through the stack's predefined area to detect the furthest extent of the stack. Figure 2 shows an example of the life of a simple stack. The simple pattern written to the stack should be non-zero, since it is quite common to have data on the stack which has been assigned to zero. It would be difficult to distinguish this data from unused stack space.

Many RTOSes offer a stack size tracing feature. If yours does not, or if you are not using an RTOS, it's not difficult to implement it yourself, though it is likely to be non-portable. The technique can be used during the testing phase to refine the stack sizes, and it can also be used on a production system to give early warning of a stack that exceeds a watermark that the designers did not expect to be reached. In this case, the watermark level on the stack is checked to see if the pattern has been overwritten. An expensive measurement of the exact extent of the stack is unnecessary. Checking the watermark on every write to the stack would be difficult and expensive, but it can be checked easily on a timed basis. This may not catch a stack overrun due to infinite recursion, which would overflow the stack very quickly, but it would catch a case where the stack grew a small amount bigger than the designers expected.

The previously described technique fails in one scenario. Consider a large local array which extends beyond the top of the stack. If the program does not write any data to the array, the pattern will not get overwritten. The highest legal piece of stack space will contain the pattern, and so it will look as if the stack did not overflow. Data pushed onto the stack will overwrite some other area of memory, but checking the stack will indicate no problem. If you guess that this is what has happened then the easiest way to check is

to make the stack size much bigger, and check the size again. Now that the array is within the bounds of the bigger stack, the true worst case stack size will be found.

Heap-based memory management

Many objects, structures, or buffers require a lifetime that does not match the invocation of any one function. This is particularly true in event-driven programs, which is typical of many embedded systems. One event may cause an item to be created, and that item will remain in use until some other event leads to its demise. In C programs, heap management is carried out by the malloc () and free () functions. The malloc () function allows the programmer to acquire a pointer to an available block of memory of a specified size. The free () function allows the programmer to return a piece of memory to the heap when the application has finished with it.

While stack management is handled by your compiler, heap management requires care by the programmer. A number of particularly devious bugs can creep into your program by way of the heap.

At a certain point in the code, you may be unsure if a particular block is no longer needed. If you free () this piece of memory, but continue to access it (probably via a second pointer to the same memory), your program may function perfectly until that particular piece of memory is reallocated to another part of the program. Then two different parts of the program will proceed to write over each other's data. If you decide to not free the memory on the grounds that it may still be in use, then you may not get another opportunity to free it (since all pointers to the block may have gone out of scope or been reassigned to point elsewhere). In this case, the program logic will not be affected. But if the piece of code that leaks memory is visited on a regular basis, the leak will tend towards infinity, as the execution time of the program increases.

Ultimately, the amount of physical memory will decide how long the program can execute. On many desktop applications, a small leak is acceptable, say a compiler which leaks 100 bytes for every 1,000 lines compiled. Such a program can still happily compile a 100,000-line file on a modern PC, since on exit of the program all allocated memory will be recovered. However, on many embedded systems, no upper limit on the life of the program is acceptable. Any memory leak is a bug and should be rectified by correcting the logic of the application program.

In addition to leaks, there is another problem called fragmentation, which can't be corrected at the application level. This problem is inherent in most implementations of malloc (). It is caused by the blocks of memory available being broken down into smaller pieces as many allocations and frees are performed.

The heap is a large block of memory that is made up of smaller blocks of memory allocated to the application and blocks that are free. Each block, allocated or freed, contains a header. The Free List pointer always points to the first available block. When an allocation is requested, this list is iterated, searching for a block to return. Ideally, a block of exactly the right size is available. If not, some larger block is broken into two. In this way, an initial heap of one large block can become a heap containing a linked list of many small blocks that are free, interspersed with many blocks that have been allocated to the application.

The danger of fragmentation has been overestimated by academic experiments that focused on randomly sized allocations. In practice, allocations tend to come in a limited number of sizes. In a survey of a number of Unix applications, it was found that 90% of

allocations were covered by six sizes, and 99.9% of allocations were covered by 14 sizes. This means that the probability of finding a block that exactly matches the size of any given request is far higher than would be estimated given a random distribution of allocation sizes.

Fragmentation can also be reduced by using the appropriate policy when allocating and freeing blocks. Possible allocation policies include:

- First Fit: allocate (and possibly split) the first block found that is large enough to fulfill the request
- Best Fit: allocate the best fit after an exhaustive search

Possible free list management policies include:

Address Order: Sort the free list in order of address, to simplify merging of adjacent free blocks

Recently-Used Order: Maintain the list in most recently used order, to match patterns of use where similar sizes are allocated and freed in bursts

Q5 (b) Consider a byte - addressable computer with 16 - bit addresses a cache capable of storing a total of 4 K bytes of data and blocks of 16 bytes. Show the format (including field names and sizes) of memory address for:

(i) Direct mapped **(ii) Fully associative**

(iii) 4 - way set associative

Answer

i. $16K = 2^4 * 2^{10} = 2^{14}$

$2^{14}/ 2^5 = 2^9$ blocks in cache, so 9 bits are needed for the block field.

5 bits are needed for the word field, leaving 2 for the tag.

ii. Again, 5 bits are needed for the word field, leaving 11 for the tag.

iii. There are $2^9/2^4= 32$ sets in cache, so 5 bits are needed for the set field.

We still need 5 bits for the word field, leaving 6 for the tag field.

Q5 (c) Draw the external block diagram of a ROM and describe it.

Answer

The 8031 chip is a ROM less version of the 8051. In other words, it is exactly like any member of the 8051 family such as the 8751 or 89C51 as far as executing the instructions and features are concerned, but it has no on-chip ROM. Therefore, to make the 8031 execute 8051 code, it must be connected to external ROM memory containing the program code. In many systems where the on-chip ROM of the 8051 is not sufficient, the use of an 8031 is ideal since it allows the program size to be as large as 64K bytes.

Although the 8031 chip itself is much cheaper than other family members, an 8031-based system is much more expensive since the ROM containing the program code is connected externally and requires more supporting circuitry. First, we review some of the pins of the 8031/51 used in external memory interfacing. Since the PC (program counter) of the 8031/51 is 16-bit, it is capable of accessing up to 64K bytes of program code. In the

8031/51, port 0 and port 2 provide the 16-bit address to access external memory. Of thes two ports, PO provides the lower 8 bit addresses AO - A7, and P2 provides the upper 8 bit addresses A8 - A15. More importantly, PO is also used to provide the 8-bit data bus DO - D7. In other words, pins PO.O - P0.7 are used for both the address and data paths. This is called address/data multiplexing in chip design. Of course the reason Intel used address/data multiplexing in the 8031/51 is to save pins. How do we know when PO is used for the data path and when it is used for the address path? This is the job of the ALE (address latch enable) pin.

ALE is an output pin for the 8031/51 microcontroller. Therefore, when ALE = 0 the 8031 uses PO for the data path, and when ALE = 1, it uses it for the address path. As a result, to extract the addresses from the PO pins we connect PO to a 74LS373 latch and use the ALE pin to latch the address. This extracting of addresses from PO is called address/data demultiplexing.

It is important to note that normally ALE = 0, and PO is used as a data bus, sending data out or bringing data in. Whenever the 8031/51 wants to use PO as an address bus, it puts the addresses AO - A7 on the PO pins and activates ALE = 1 to indicate that PO has the addresses

Q6 (a) What are the characteristics taken into consideration when interfacing a device and a port?

Answer

Start Bits and Stop Bits

In asynchronous communication, at least two extra bits are transmitted with the data word; a start bit and a stop bit. Therefore, if the transmitter is using an 8-bit system, the actual number of bits transmitted per word is ten.

In most protocols the start bit is a logic 0 while the stop bit is logic 1.

Therefore, when no data is being sent the data line is continuously HIGH.

The receiver waits for a 1 to 0 transition. In other words, it awaits a transition from the stop bit (no data) to the start bit (logic 0). Once this transition occurs the receiver knows a data byte will follow.

Since it knows the data rate (because it is defined in the protocol) it uses the same clock as frequency as that used by the transmitter and reads the correct number of bits and stores them in a register. For example, if the protocol determines the word size as eight bits, once the receiver sees a start bit it reads the next eight bits and places them in a buffer.

Once the data word has been read the receiver checks to see if the next bit is a stop bit, signifying the end of the data. If the next bit is not a logic 1 then something went wrong with the transmission and the receiver dumps the data.

If the stop bit was received the receiver waits for the next data word, ie; it waits for a 1 to 0 transition.

The 8051 Serial Port

The 8051 includes an on-chip serial port that can be programmed to operate in one of four different modes and at a range of frequencies. In serial communication the data is rate is known as the baud rate, which simply means the number of bits transmitted per second. In the serial port modes that allow variable baud rates, this baud rate is set by timer 1.

The 8051 serial port is full duplex. In other words, it can transmit and receive data at th same time. The block diagram above shows how this is achieved. If you look at the memory map you will notice at location 99H the serial buffer special function register (SBUF). Unlike any other register in the 8051, SBUF is in fact two distinct registers - the write-only register and the read-only register. Transmitted data is sent out from the write-only register while received data is stored in the read-only register. There are two separate data lines, one for transmission (TXD) and one for reception (RXD). Therefore, the serial port can be transmitting data down the TXD line while it is at the same time receiving data on the RXD line.

The TXD line is pin 11 of the microcontroller (P3.1) while the RXD line is on pin 10 (P3.0). Therefore, external access to the serial port is achieved by connecting to these pins. For example, if you wanted to connect a keyboard to the serial port you would connect the transmit line of the keyboard to pin 10 of the 8051. If you wanted to connect a display to the serial port you would connect the receive line of the display to pin 11 of the 8051. This is detailed in the diagram below.

Transmitting and Receiving Data

Essentially, the job of the serial port is to change parallel data into serial data for transmission and to change received serial data into parallel data for use within the microcontroller.

Serial transmission is changing parallel data to serial data.

Serial reception is changing serial data into parallel data.

Both are achieved through the use of shift registers.

As discussed earlier, synchronous communication requires the clock signal to be sent along with the data while asynchronous communication requires the use of stop bits and start bits. However, the programmer wishing to use the 8051 need not worry about such things. To transmit data along the serial line you simply write to the serial buffer and to access data received on the serial port you simply read data from the serial buffer.

For example:

MOV SBUF, #45H - this sends the byte 45H down the serial line

MOV A, SBUF - this takes whatever data was received by the serial port and puts it in the accumulator.

How do we know when the complete data byte has been sent?

As mentioned earlier, it takes a certain length of time for a data byte to be transmitted down the serial line (determined by the baud rate). If we send data to SBUF and then immediately send more data to SBUF, as shown below, the initial character will be overwritten before it was completely shifted down the line.

```
MOV SBUF, #23H
MOV SBUF, #56H
```

Therefore, we must wait for the entire byte to be sent before we send another. The serial port control register (SCON) contains a bit which alerts us to the fact that a byte has been transmitted; ie; the transmit interrupt flag (TI) is set by hardware once an entire byte has been transmitted down the line. Since SCON is bit-addressable we can test this bit and wait until it is set, as shown below:

```
MOV SBUF, #23H; send the first byte down the serial line
JNB TI, $; wait for the entire byte to be sent
```

CLR TI; the transmit interrupt flag is set by hardware but must be cleared by software
MOV SBUF, #56H; send the second byte down the serial line
Similarly, we need to know when an entire byte has been received by the serial port. Another bit in SCON, the receive interrupt flag (RI) is set by hardware when an entire byte is received by the serial port. The code below shows how you would program the controller to wait for data to be received and to then move that data into the accumulator.
JNB RI, $; wait for an entire byte to be received
CLR RI; the receive interrupt flag is set by hardware but must be cleared by software
MOV A, SBUF; move the data stored in the read-only buffer to the accumulator

Q6 (b) List the features of synchronous, iso-synchronous and asynchronous serial communication?

Answer

Asynchronous serial communication describes an asynchronous, serial transmission protocol in which a start signal is sent prior to each byte, character or code word and a stop signal is sent after each code word. The start signal serves to prepare the receiving mechanism for the reception and registration of a symbol and the stop signal serves to bring the receiving mechanism to rest in preparation for the reception of the next symbol. A common kind of start-stop transmission is ASCII over RS-232, for example for use in teletypewriter operation.

In the diagram, two bytes are sent, each consisting of a start bit, followed by seven data bits (bits 0-6), a parity bit (bit 7), and one stop bit, for a 10-bit character frame. The number of data and formatting bits, the order of data bits, and the transmission speed must be pre-agreed by the communicating parties.

The "stop bit" is actually a "stop period"; the stop period of the transmitter may be arbitrarily long. It cannot be shorter than a specified amount, usually 1 to 2 bit times. The receiver requires a shorter stop period than the transmitter. At the end of each character, the receiver stops briefly to wait for the next start bit. It is this difference which keeps the transmitter and receiver synchroized.

When devices exchange data, there is a flow or stream of information between the two. In any data transmission, the sender and receiver must have a way to extract individual characters or blocks (frames) of information. Imagine standing at the end of a data pipe. Characters arrive in a continuous stream of bits, so you need a way to separate one block of bits from another. In asynchronous communications, each character is separated by the equivalent of a flag so you know exactly where characters are located. In synchronous communications, both the sender and receiver are synchronized with a clock or a signal encoded into the data stream.

In synchronous communications, the sender and receiver must synchronize with one another before data is sent. To maintain clock synchronization over long periods, a special bit-transition pattern is embedded in the digital signal that assists in maintaining the timing between sender and receiver. In this method, the bit stream pictured at the top is meshed with the clock pulse pictured in the middle to produce the transmission signal shown at the bottom.

Synchronous communications are either character oriented or bit oriented. Character-oriented transmissions are used to send blocks of characters such as those found in ASCII (American Standard Code for Information Interchange) files. Each block must have a

starting flag similar to asynchronous communications so the receiving system ca initially synchronize with the bit stream and locate the beginning of the characters. Two or more control characters, known as SYN (synchronous idle) characters, are inserted at the beginning of the bit stream by the sender. These characters are used to synchronize a block of information. Once correct synchronization has been established between sender and receiver, the receiver places the block it receives as characters in a memory buffer.

Bit-oriented synchronous communication is used primarily for the transmission of binary data. It is not tied to any particular character set, and the frame contents don't need to include multiples of eight bits. A unique 8-bit pattern (01111110) is used as a flag to start the frame.

An entirely different form of synchronous communications can be seen in the form of chat and instant messaging. Like a voice telephone call, a chat or instant messaging session is live and each user responds to the other in real time. In contrast, discussion forums and electronic mail are asynchronous communications. Some amount of time may pass before a person responds to a message. In a discussion forum, a message sits in a message queue for other people to read and respond to at any time, or until the message falls out of the queue. These two forms of communication, which are accessible to any Internet user from just about any Web-attached system, may be the most profound aspect of the Internet. They promote a new form of instant global communication and collaboration. In the case of discussion forums and e-mail, delayed communication gives respondents time to think about their response and gather information from other sources before responding.

Q6 (c) What is the advantage of Direct Memory Access? Give a diagram to explain it.

Answer

DMA is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy overhead. Otherwise, the CPU would have to copy each piece of data from the source to the destination, making itself unavailable for other tasks. This situation is aggravated because access to I/O devices over a peripheral bus is generally slower than normal system RAM. With DMA, the CPU gets freed from this overhead and can do useful tasks during data transfer (though the CPU bus would be partly blocked by DMA). In the same way, a DMA engine in an embedded processor allows its processing element to issue a data transfer and carries on its own task while the data transfer is being performed.

A DMA transfer copies a block of memory from one device to another. While the CPU initiates the transfer by issuing a DMA command, it does not execute it. For so-called "third party" DMA, as is normally used with the ISA bus, the transfer is performed by a DMA controller which is typically part of the motherboard chipset. More advanced bus designs such as PCI typically use bus mastering DMA, where the device takes control of the bus and performs the transfer itself. In an embedded processor or multiprocessor system-on-chip, it is a DMA engine connected to the on-chip bus that actually administers the transfer of the data, in coordination with the flow control mechanisms of the on-chip bus.

A typical usage of DMA is copying a block of memory from system RAM to a buffer o the device or vice versa. Such an operation usually does not stall the processor, which as a result can be scheduled to perform other tasks unless those tasks include a read from or write to memory. DMA is essential to high performance embedded systems. It is also essential in providing so-called zero-copy implementations of peripheral device drivers as well as functionalities such as network packet routing, audio playback and streaming video. Multicore embedded processors (in the form of multiprocessor system-on-chip) often use one or more DMA engines in combination with scratchpad memories for both increased efficiency and lower power consumption. In computer clusters for high-performance computing, DMA among multiple computing nodes is often used under the name of remote DMA. There are two control signal used to request and acknowledge a DMA transfer in microprocess-based system. The HOLD pin is used to request a DMA action and the HLDA pin is an output acknowledges the DMA action.

DMA can lead to cache coherency problems. Imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X. If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X.

Similarly, if the cached copy of X is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value of X.

This issue can be addressed in one of two ways in system design: Cache-coherent systems implement a method in hardware whereby external writes are signaled to the cache controller which then performs a cache invalidation for DMA writes or cache flush for DMA reads. Non-coherent systems leave this to software, where the OS must then ensure that the cache lines are flushed before an outgoing DMA transfer is started and invalidated before a memory range affected by an incoming DMA transfer is accessed. The OS must make sure that the memory range is not accessed by any running threads in the meantime. The latter approach introduces some overhead to the DMA operation, as most hardware requires a loop to invalidate each cache line individually.

Hybrids also exist, where the secondary L2 cache is coherent while the L1 cache (typically on-CPU) is managed by software.

Q7 (a) Explain the various RTOS task scheduling models. Why is priority inversion problem? When does it occur?

Answer

In typical designs, a task has three states: 1) running (executing on the CPU), 2) ready (ready to be executed), 3) blocked (waiting for input/output). Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can greatly vary, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state (see: Resource Starvation)

Usually the data structure of the ready list in the scheduler is designed t minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit preemption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per ready-queue entry, and restoration of the highest-priority ready task will take 5 to 30 instructions.

In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Algorithms

Some commonly used RTOS scheduling algorithms are:

- Cooperative scheduling
- Preemptive scheduling
- Rate-monotonic scheduling
- Round-robin scheduling
- Fixed priority pre-emptive scheduling, an implementation of preemptive time slicing
- Fixed-Priority Scheduling with Deferred Preemption
- Fixed-Priority Non-preemptive Scheduling
- Critical section preemptive scheduling
- Static time scheduling
- Earliest Deadline First approach
- Advanced scheduling using the stochastic and MTG

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

Temporarily masking/disabling interrupts General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wishes. Modern CPUs don't allow

user mode code to disable interrupts as such control is considered a key operating system resource. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single-processor systems, if the application runs in kernel mode and can mask interrupts, often interrupt disablement is the best (lowest overhead) solution to prevent simultaneous access to a shared resource. While interrupts are masked, the current task has exclusive use of the CPU since no other task or interrupt can take control, so the critical section is protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency, or else this method increases the system's maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few instructions and contains no loops. This method is ideal for protecting hardware bit-mapped registers when the bits are controlled by different tasks.

Q7 (b) List the ways in which an RTOS handles the ISR in a multitasking environment.

Answer

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

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

OSes which implement the segmented architecture are more predictable an can deal with higher interrupt rates compared to the unified architecture.

Q7 (c) Discuss with a diagram Task synchronization model for a specific application.

Answer

(Kernel-level threading)

Threads created by the user are in 1-1 correspondence with schedulable entities in the kernel. This is the simplest possible threading implementation. Win32 used this approach from the start. On Linux, the usual C library implements this approach. The same approach is used by Solaris, NetBSD and FreeBSD.

N: 1 (User-level threading)

An N: 1 model implies that all application-level threads map to a single kernel-level scheduled entity; the kernel has no knowledge of the application threads. With this approach, context switching can be done very fast and, in addition, it can be implemented even on simple kernels which do not support threading. One of the major drawbacks however is that it cannot benefit from the hardware acceleration on multi-threaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time. It is used by GNU Portable Threads.

N:M (Hybrid threading)

N: M maps some N number of application threads onto some M number of kernel entities, or "virtual processors." This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "N: M" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required. In the N: M implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls. However, this increases complexity and the likelihood of priority inversion, as well as suboptimal scheduling without extensive coordination between the user land scheduler and the kernel scheduler.

Q8 (a) Enlist the standard features of events and compare the methods of intertask communication.

Answer

There are several ways of implementing the scheduler -- preemptive or cooperative, round robin or with priority. In a cooperative or non-preemptive system, tasks cooperate with one another and relinquish control of the CPU themselves. In a preemptive system, a task may be preempted or suspended by different task, either because the latter has a higher priority or the time slice of the former one is used up. Round robin scheduler switches in one task after another in a round robin manner whereas a system with priority will switch in the highest priority task.

For many small microcontroller based embedded systems, a cooperative (or non-preemptive), round robin scheduler is adequate. This is the simplest to implement and it does not take up much memory. Ravindra Karnad has implemented such a scheduler for 8051 and other microcontrollers. In his implementation, all tasks must behave cooperatively. A task waiting for an input event thus cannot have infinite waiting loop such as the following:

```
While (TRUE)
{
        Check input
        ...
       }
```

This will hog processor time and reprieve others of running. Instead, it may be written as:

```
If (input TRUE)
{
 ...
}
Else (timer[i]=100ms)
```

In this case, task i will check the input condition every 100 ms, set in the associated timer$[$i$]$. When the condition of input is false, other tasks will have a chance to run.

The job of the scheduler is thus rather simple. When there is clock interrupt, all task timers are decremented. The task whose timer reaches 0 will be run. The greatest virtue of the simple task scheduler ready lies in the smallness of the code, which is of course very important in the case of microcontrollers. The code size ranges from 200 to 400 byes.

Page Number 212 of Text-Book -II

Q8 (b) Give a short note on the working of mail boxes and pipes in an embedded system.

Answer

Pipes depend on the convention that every program has initially available to it (at least) two I/O data streams; standard input and standard output (numeric file descriptors 0 and 1 respectively). Many programs can be written as *filters*, which read sequentially from standard input and write only to standard output.

Normally these streams are connected to the user's keyboard and display, respectively. But Unix shells universally support *redirection* operations which connect these standard input and output streams to files. Thus, typing

```
ls >foo
```

sends the output of the directory lister ls(1) to a file named ‘foo’. On the other hand, typing:

```
wc <foo
```

causes the word-count utility wc(1) to take its standard input from the file ‘foo’, and deliver a character/word/line count to standard output.

The pipe operation connects the standard output of one program to the standard input of another. A chain of programs connected in this way is called a *pipeline*. If we write

```
ls | wc
```

Pipes may have ends that may be moved around and bound to different peers at different times. Point-to-point and propagate pipes may be supported. Pipes may connect peers that have a direct physical link and peers that do not have a direct link. Peers may communicate through pipes without knowing on which peer a pipe endpoint is bound. A message is sent to all peer endpoints currently connected (listening) to the pipe. The set of connected endpoints may be obtained from a pipe service using a pipe binding protocol.

Q8(c) What are queue related functions?

Answer

A task has three states: 1) running (executing on the CPU), 2) ready (ready to be executed), 3) blocked (waiting for input/output). Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can greatly vary, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state.

Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit preemption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per ready-queue entry, and restoration of the highest-priority ready task will take 5 to 30 instructions.

In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Q9 (a) Explain the need of tasks for priority and encapsulation in real-tim operating system.

Answer

```
import curses as c

def doKeyEvent(key):
   if key == '\x00' or key == '\xe0': # non ASCII key
      key = screen.getch() # fetch second character
   screen.addstr(str(key)+' ')

def doQuitEvent(key):
   raise SystemExit

# clear the screen of clutter, stop characters auto
# echoing to screen and then tell user what to do to quit

screen = c.initscr()
c.noecho()
screen.addstr("Hit space to end...\n")

# Now mainloop runs "forever"
while True:
    ky = screen.getch()
    if ky != -1:
      # send events to event handling functions
      if ky == ord(" "): # check for quit event
           doQuitEvent(ky)
      else:
           doKeyEvent(ky)

c.endwin()
```

Page Number 244 of Text-Book-II

Q9 (b) What are the efficient memory management techniques for saving memory space and power?

Answer

All too often, programs written for embedded systems grow and grow until they exceed the available program space. There are a variety of techniques for dealing with the out-of-memory problem:

- re-compile with the "-Os" (optimize for size) option
- find and comment-out "dead code"
- "refactor" repeated sections into a common subroutine

- trade RAM space for program space.
- put a small interpreter in "internal program memory" that loads and interprets "instructions".
    1. use "instructions" -- perhaps p-code or threaded code -- that are more compact than directly coding it in assembly language. Or
    2. place these "instructions" can be placed in EEPROM or external serial Flash that couldn't otherwise be used as program memory. Or
    3. Both. This technique is often used in "stamp" style CPU modules.
- add more memory (perhaps using a paging or banking scheme)

Most CPUs used in desktop machines have a "memory management unit" (MMU). The MMU handles virtual memory, protects regions of memory used by the OS from untrusted programs.

Most embedded systems do not have a MMU. We discuss the two versions of Linux that can run on a system that does not have a MMU in Embedded Systems/Linux.

Text Book

1. Embedded System Design, A Unified Hardware/Software Introduction, Frank Vahid / Tony Givargis, 2006 reprint, John Wiley Student Edition.

2. An Embedded Software Primer, David .E. Simon, Fourth Impression 2007, Pearson Education.