Q.2 a. Discuss design metrics and explain with an example how metrics typically compete with each other. (5)

Answer:

Design metrics, NRE cost, Unit Cost, Size, Performance, Power, Flexibility, Time to prototype, Time to prototype, Time to market, Maintainability, Correctness, Safety

An example of competitive metrics—size v/s performance and so on

b. What are the IC Technology available for digital circuit implementation? Explain the current state of art for this. (5)

Answer:

Full-Custom/Semicustom/PLD latest trends of technology node e.g 32 nanometer for FPGaA and 22nanometer in labs

c. Design process of a chip is itself quite complex and is constantly evolving. Discuss the steps in Design process and give an example of how it is being improved. (6)

Answer:

Design technology involves the manner in which we convert our concept of desired system functionality into an implementation. We must not only design the implementation to optimize design metrics, but we must do so quickly. As described earlier, the designer must be able to produce larger numbers of transistors every year to keep pace with IC technology. Hence, improving design technology to enhance productivity has been a key focus of the software and hardware design communities for decades.

To understand how to improve the design process, we must first understand the design process itself. Variations of a *top-down design* process have become popular in the past decade, an ideal form of which is illustrated in Figure 1.11. The designer refines the system through several abstraction levels. At the system level, the designer describes the desired functionality in some language, often a natural language like English, but preferably an executable language like C; we shall call this the *system specification*. The designer refines this specification by distributing portions of it among several general and/or single-purpose processors, yielding *behavioral specifications* for each processor. The designer refines these specifications into *register-transfer* (RT) *specifications* by converting behavior on general-purpose processors to assembly code, and by converting behavior on single-purpose processors to a connection of register-transfer components and state machines. The designer then refines the register-transfer-level specification of a single-purpose processor into a *logic specification* consisting of Boolean equations; no refinement of a general-purpose processor's assembly code is done at this level. Finally, the designer refines the remaining specifications into an implementation, consisting of machine code for general-purpose processors and a gate-level netlist for single-purpose processors.

Steps oin design Process page 17 Fig1.11 of Vahids book,

Compilation/Synthesis

Improvements in design process with use of libraries/IP

Test/Verification

High Level languages C to VHDL etc

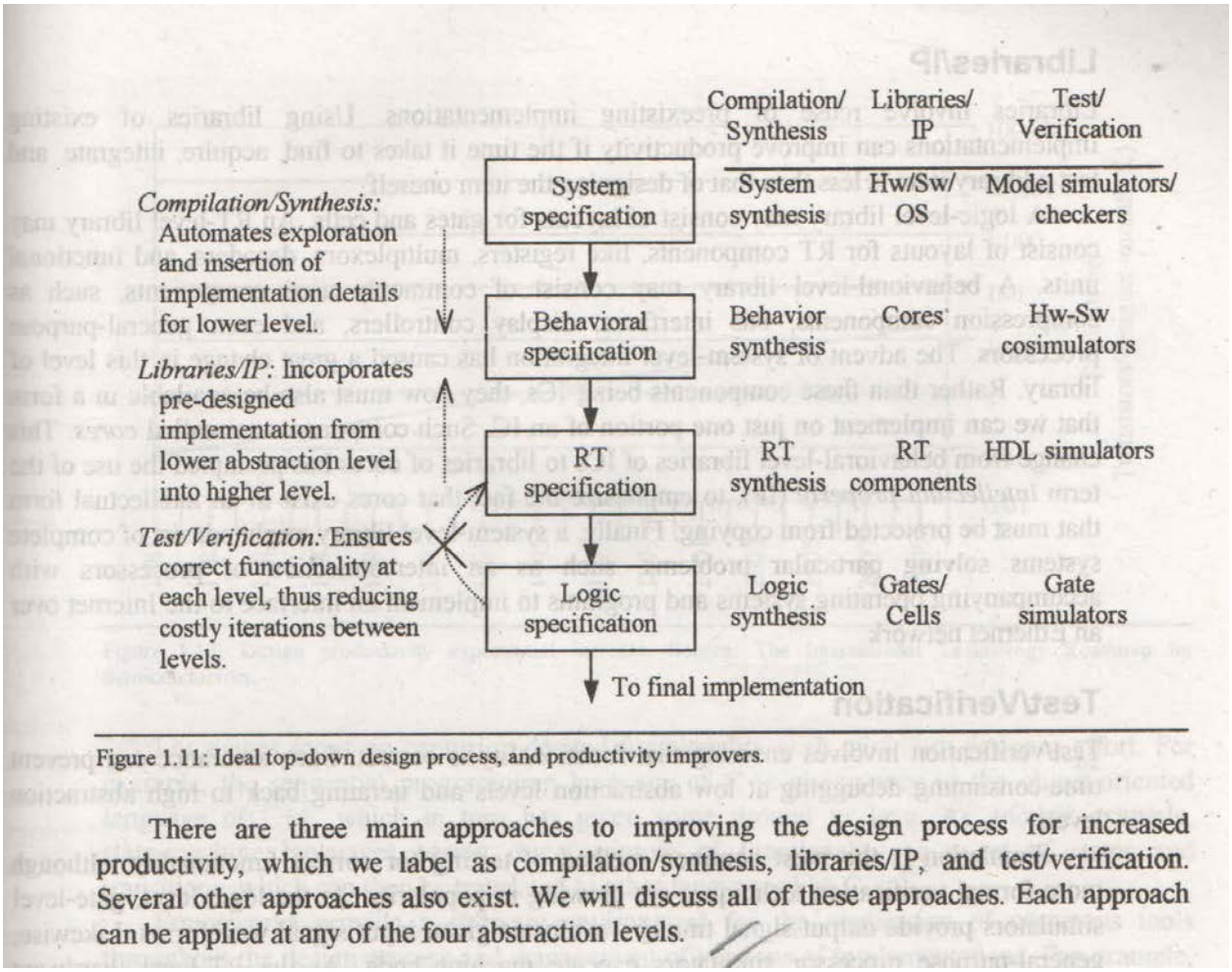


Figure 1.11: Ideal top-down design process, and productivity improvers.

There are three main approaches to improving the design process for increased productivity, which we label as compilation/synthesis, libraries/IP, and test/verification. Several other approaches also exist. We will discuss all of these approaches. Each approach can be applied at any of the four abstraction levels.

Q.3 a. Differentiate between timer, counter and watchdog Timer. (3)

Answer:

Timers and Counters

A *timer* is an extremely common peripheral device that can measure time intervals. Such a device can be used to either generate events at specific times, or to determine the duration between two external events. Example applications that require generating events include keeping a traffic light green for a specified duration, or communicating bits serially between devices at a specific rate. An example of an application that determines inter-event duration is that of computing a car's speed by measuring the time the car takes to pass over two separated sensors in a road.

A timer measures time by counting pulses that occur on an input clock signal having a known period. For example, if a particular clock's period is 1 microsecond, and we've counted 2,000 pulses on the clock signal, then we know that 2,000 microseconds have passed.

A *counter* is a more general version of a timer. Instead of counting clock pulses, a counter counts pulses on some other input signal. For example, a counter may be used to count the number of cars that pass over a road sensor, or the number of people that pass through a turnstile. We often combine counters and timers to measure rates, such as counting the number of times a car wheel rotates in one second, in order to determine a car's speed.

To use a timer, we must configure its inputs and monitor its outputs. Such use often requires or can be greatly aided by an understanding of the internal structure of the timer. The internal structure can vary greatly among manufacturers. We provide a few common features of such internal structures in Figure 4.1.

Watchdog Timers

A special type of timer is a watchdog timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, we must generate a signal for the timer every X time units. If we fail to generate this signal in time, then the timer "times out" and generates a signal indicating that we failed.

b. Given a timer structured with 16 bit up counter and a clock frequency of 10 MHz, determine its range and resolution. (4)

Answer:

Solution:

resolution = period = 1 / frequency = 1 / (10 MHz) = 1e-7 s
range = 2^16 * resolution = 65536 * 1e-7s = .0065536 s
= 0 to 6.5536 ms

c. Explain Pulse With Modulation Modulators with help of an example showing control of a DC motor using PWM technique. (9)

Answer:

Overview

A *pulse width modulator* (PWM) generates an output signal that repeatedly switches between high and low values. We control the duration of the high value and of the low value by indicating the desired period, and the desired *duty cycle*, which is the percentage of time the signal is high compared to the signal's period. A *square wave* has a duty cycle of 50%. The pulse's width corresponds to the pulse's time high, as shown in Figure 4.5.

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

A common use of a PWM is to generate a clock-like signal to another device. For example, a PWM can be used to blink a light at a specific rate.

Another common use of a PWM is to control the average current or voltage input to a device. For example, a DC (direct current) electric motor rotates when its input voltage is set high, with the rotation speed proportional to the input voltage level. Suppose the revolutions per minute (rpm) equals 10 times the input voltage. To achieve a desired rpm of 125, we would need to set the input voltage to 1.25 V, whereas achieving 250 rpm would require an input voltage of 2.50 V.

One approach to control the average input voltage to a DC motor uses a DC-to-DC converter circuit, which converts some reference voltage to a desired voltage. However, these circuits can be expensive. Another approach uses a digital-to-analog converter. A third approach, perhaps the simplest, uses a PWM. The PWM approach makes use of the fact that a DC motor does not come to an immediate stop when its input voltage is lowered to 0, but rather it coasts, much like a bicycle coasts when we stop pedaling. Thus, we need only set the

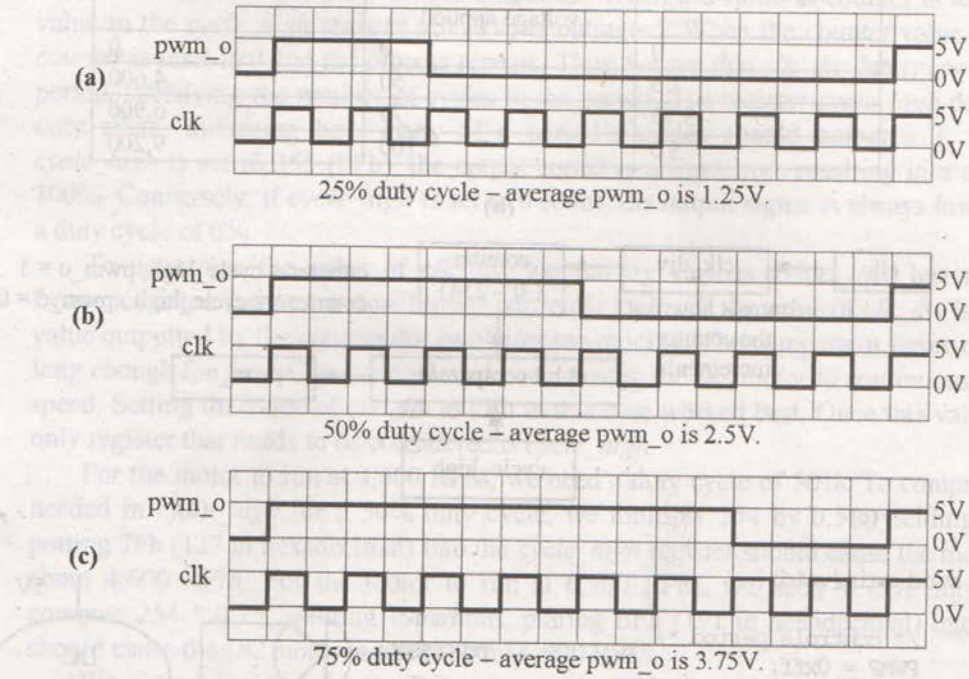


Figure 4.5: Operation of a PWM, (a) 25% duty cycle, (b) 50% duty cycle, (c) 75% duty cycle. In the diagrams, logic high is 5V, low is 0V.

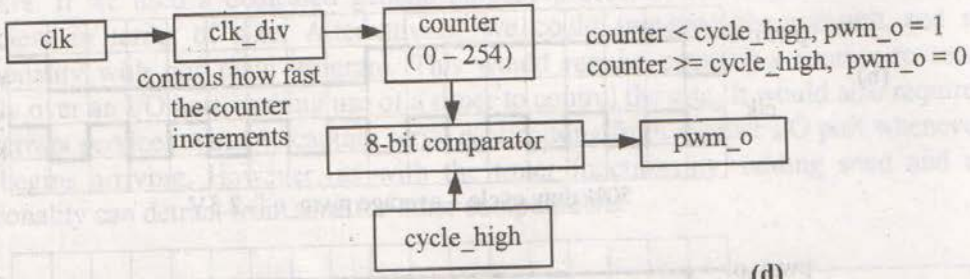
average input voltage appropriately to obtain the desired speed. Using a PWM, we set the duty cycle to achieve the appropriate average voltage, and we set the period small enough for smooth operation of the motor (i.e., so the motor does not noticeably speed up and slow down). Assuming the PWM's output is 5 V when high and 0 V when low, then we can obtain an average output of 1.25 V by setting the duty cycle to 25%, since 5 V * 25% = 1.25 V. This duty cycle is shown in Figure 4.5(a). Likewise, we can obtain an average output of 2.50 V by setting the duty cycle to 50%, as shown in Figure 4.5(b). A duty cycle of 75% would result in average output of 3.75 V, as shown in Figure 4.5(c). This duty cycle adjustment principle applies to the control of a wide variety of electric devices, such as dimmer lights.

Another use of a PWM is to encode control commands in a single signal for use by another device. For example, we may control a radio-controlled car by sending pulses of different widths. Perhaps a width of 1 ms corresponds to a turn left command, a 4-ms width to turn right, and an 8-ms width to forward. The receiver can use a timer to measure the pulse width, by starting a timer when the pulse starts and stopping the timer when the pulse ends, and thus determining how much time elapsed.

(a)

input voltage	% of maximum voltage applied	RPM of DC motor
0	0	0
2.5	50	4,600
3.75	75	6,900
5.0	100	9,200

(b)



(c)

```
void main(void){

  /* controls period */
  PWMP = 0xff;
  /* controls duty cycle */
  PWM1 = 0x7f;
  while(1){};
}
```

(d)

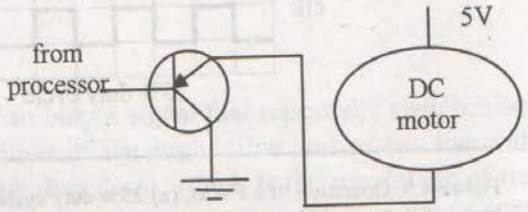


Figure 4.6: Controlling a DC motor with a PWM: (a) relationship between applied voltage and DC motor speed, (b) internal PWM structure, (c) pseudo-code, (d) connection to DC motor.

Example: Controlling a DC Motor Using a PWM

In this example, we wish to control the speed of a direct-current (DC) electric motor using a PWM. The speed of the DC motor is proportional to the voltage applied to the motor. Suppose that for a fixed load, the motor yields the revolutions per minute (rpm) shown in Figure 4.6(a) for the given input voltages. We must set the duty cycle of a PWM such that the average output voltage equals the desired voltage.

Suppose that we use a PWM as part of a system that includes two 8-bit registers called *clk_div* and *cycle_high*, an 8-bit counter, and an 8-bit comparator, as shown in Figure 4.6(b). The PWM works as follows. Initially, the value of *clk_div* is loaded into the register. The *clk_div* register works as a clock divider. After a specified amount of time has elapsed, a pulse is sent to the *counter* register. This causes the counter to increment itself. The comparator then

looks at the values in the *counter* register and the *cycle_high* register. When the counter value is less than *cycle_high*, a 1 (+5V) is outputted. When the value in counter is lower than the value in the *cycle_high* register a 0 (0V) is outputted. When the counter value reaches 254, *counter* is reset to 0 and the process repeats. Thus, we see that *clk_div* determines the PWM's period, specifying the number of cycles in the period. The register *cycle_high* determines the duty cycle, indicating how many of a period's cycles should output a 1. Note that if *cycle_high* is set to 255 (FFh), the output signal is always high resulting in a duty cycle of 100%. Conversely, if *cycle_high* is set to 0 (00h), the output signal is always low resulting in a duty cycle of 0%.

To determine the value of *clk_div*, we can try various values and test to see if the frequency is too fast or too slow for our particular motor. If the value of *clk_div* is too low, the value outputted by the comparator oscillates too quickly. The comparator never outputs zeros long enough for the DC motor to slow down, causing the DC motor to continuously run at full speed. Setting the value of *clk_div* to FFh in this case worked best. Once this value is set, the only register that needs to be considered is *cycle_high*.

For the motor to run at 4,600 RPM, we need a duty cycle of 50%. To compute the value needed in *cycle_high* for a 50% duty cycle, we multiply 254 by 0.50, yielding 127. Thus, putting 7Fh (127 in hexadecimal) into the *cycle_high* register should cause the motor to run at about 4,600 RPM. For the motor to run at 6,900 RPM, we need a 75% duty cycle. We compute 254 * 0.75, yielding 191. Thus, putting BFh (191 in hexadecimal) into *cycle_high* should cause the DC motor to run at about 6,900 RPM.

We cannot just connect the DC motor to the PWM because the PWM does not provide enough current to run the DC motor. To remedy this problem, we use an NPN transistor to drive the DC motor. The code and schematic used for this example are found in Figure 4.6(c) and (d). In the figure, the name of the *clk_div* register is *PWMP* and *cycle_high* is *PWM1*

Q.4 a. What benefits are derived if we choose to implement systems functionality on a general purpose processor? (4)

Answer:

Low unit cost of processor

Manufacturer can afford high NRE cost in processor design as it is amortized over large number of units .

Only software need be written by the embedded system designer.

b. Define (i) MIPS (ii) throughput (iii) benchmarks with examples (3)

Answer:

Standard definitions of (i) MIPS

(ii) throughput w.r.t . Pipelined circuits

(iii) Benchmarks with examples e.g Dhrystone benchmarks

c. What forms the programmers view of a processor? Give the fields that form part of an instruction. Explain with the help of an example. (3)

Answer:

Instruction Set view

Instruction parts

Opcode operands various combination

d. Discuss the various addressing modes possible in a processor. (6)

Answer

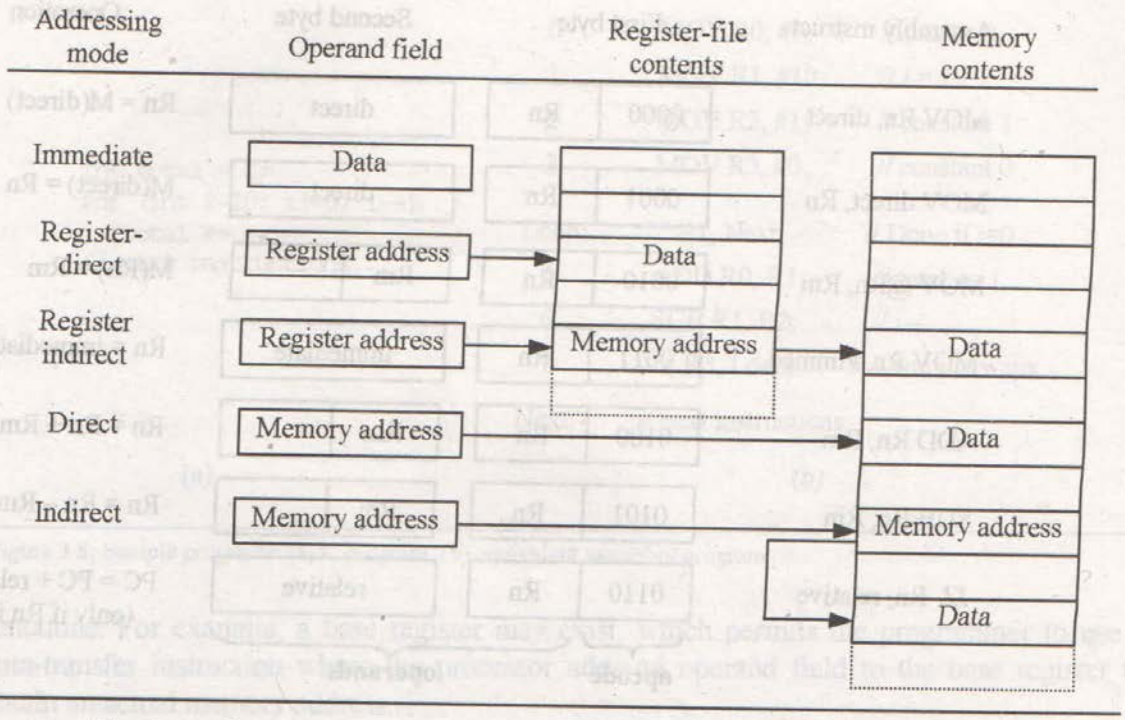


Figure 3.6: Addressing modes.

The operand field may indicate the data's location through one of several addressing modes, illustrated in Figure 3.6. In *immediate* addressing, the operand field contains the data itself. In *register* addressing, the operand field contains the address of a datapath register in which the data resides. In *register-indirect* addressing, the operand field contains the address of a register, which in turn contains the address of a memory location in which the data resides. In *direct* addressing, the operand field contains the address of a memory location in which the data resides. In *indirect* addressing, the operand field contains the address of a memory location, which in turn contains the address of a memory location in which the data resides. Those familiar with structured languages may note that direct addressing implements regular variables, and indirect addressing implements pointers. In *inherent* or *implicit* addressing, the particular register or memory location of the data is implicit in the opcode; for example, the data may reside in a register called the "accumulator." In *indexed* addressing, the direct or indirect operand must be added to a particular implicit register to obtain the actual operand address. Jump instructions may use *relative* addressing to reduce the number of bits needed to indicate the jump address. A relative address indicates how far to jump from the current address, rather than indicating the complete address. Such addressing is very common since most jumps are to nearby instructions.

Immediate
Register direct
Register indirect
Direct
Indirect

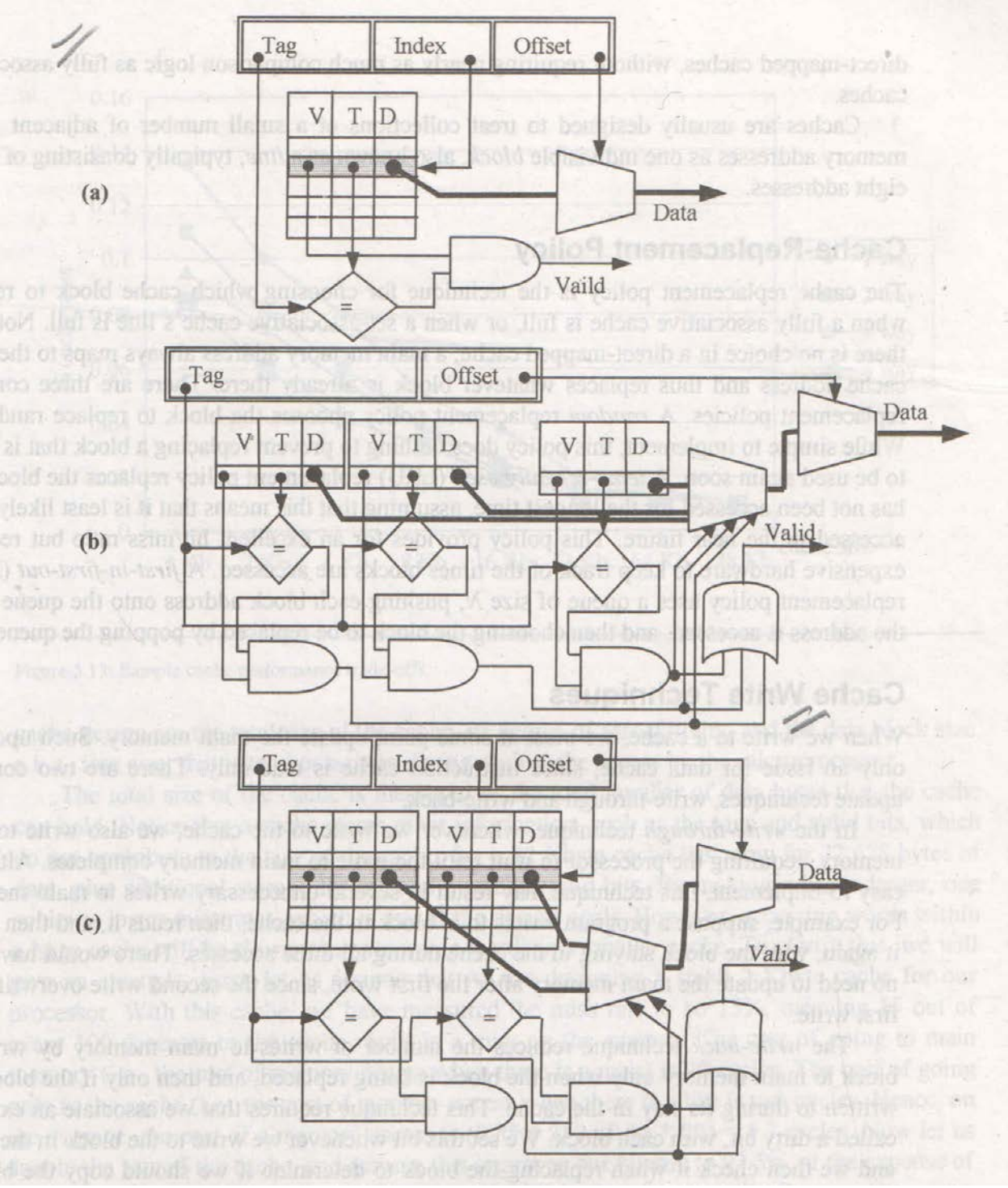
Q.5 a. Explain and Compare direct mapping and fully associative mapping for Cache mapping. (4)

Answer:

Cache Mapping Techniques

Cache mapping is the method for assigning main memory addresses to the far fewer number of available cache addresses, and for determining whether a particular main memory address's contents are in the cache. Cache mapping can be accomplished using one of three basic techniques (see Figure 5.12):

1. In *direct mapping*, illustrated in Figure 5.12(a), the main memory address is divided into two fields, the index and the tag. The *index* represents the cache address, and thus the number of index bits is determined by the cache size (i.e., index size = $\log_2$(cache size)). Note that many different main memory addresses will map to the same cache address. When we store the contents of a main memory address in the cache, we also store the *tag*. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and compare the tag there with the desired tag. If the tags match, then we check the valid bit. The *valid bit* indicates whether the data stored in that cache slot has previously been loaded into the cache from the main memory. We use the *offset* portion of the memory address to grab a particular word within the cache line. A cache *line*, also known as a cache *block*, is the number of (inseparable) adjacent memory addresses loaded from or stored into main memory at a time. A typical block size is four or eight addresses.
2. In *fully associative mapping*, illustrated in Figure 5.12(b), each cache address contains not only the contents of a main memory address, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.



Direct caches are easy to implement but may result in numerous misses if two or more words with same index are accessesd frequently

Fully associative caches are fasr but comparison logic is expensive to implement.

b. A given design with cache implemented has a main memory access cost of 20 cycles on a miss and two cycles on a hit. The same design without the cache has a main memory access cost of 16 cycles. Calculate the minimum hit rate of the cache to make the cache Implementation worthwhile. (8)

Answer:

H = hit rate
miss rate = 1-hit rate = 1-H
avg. memory access cost (cache) < memory access cost (no cache)
2 cycles * H + 20 cycles * (1-H) < 16 cycles
2H + 20 – 20H < 16
– 18H < – 4
H > (4/18) = .22 = 22% hit rate minimum

c. Define and discuss NVRAM. (4)

Answer:

NVRAM — Nonvolatile RAM

Nonvolatile RAM, or NVRAM, is a special RAM variation that is able to hold its data even after external power is removed. There are two common types of NVRAM.

One type, often called *battery-backed RAM*, contains a static RAM along with it own permanently connected battery. When external power is removed or drops below a certain threshold, the internal battery maintains power to the SRAM, and thus the memory continues to store its bits. Compared with other forms of nonvolatile memory, battery-backed RAM is far more writable, as illustrated in Figure 5.2. Since no special programming is necessary, writes are done in nanoseconds, just like reads. Furthermore, unlike ROM-based forms of nonvolatile memory, battery-backed RAM imposes no limits on the number of times it can be written to. Storage permanence is obviously better than SRAM or DRAM, with many NVRAMs having batteries that can last for 10 years. However, NVRAMs are more susceptible to having bits changed inadvertently due to noise than are EEPROM or flash.

A second type of NVRAM contains a static RAM as well as an EEPROM or flash having the same capacity as the static RAM. This type of NVRAM stores its complete RAM contents into the EEPROM just before power is turned off, or whenever instructed to store the data, and then reloads that data from EEPROM into RAM after power is turned back on. //

Q.6 a. Discuss the classification of port based and bus based I/O with further sub-classification details. (4)

Answer:

Port and Bus-Based I/O

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

In *port-based I/O*, also known as *parallel I/O*, a port can be directly read and written by processor instructions just like any other register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named $P0$. A C-language programmer may write to $P0$ using an instruction like: $P0 = 255$, which would set all eight pins to 1s. In this case, the C compiler manual would have defined $P0$ as a special variable that would automatically be mapped to the register $P0$ during compilation. Conversely, the programmer might read the value of a port $P1$ being written by some other device by typing something like $a = P1$. In some microprocessors, each bit of a port can be configured as input or output by writing to a configuration register for the port. For example, $P0$ might have an associated configuration register called $CP0$. To set the high-order four bits to input and the low-order four bits to output, we might say: $CP0 = 15$. This writes 00001111 to the $CP0$ register, where a 0 means input and a 1 means output. Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say: $x = P0.2$, giving x the value of the number 2 pin of port $P0$.

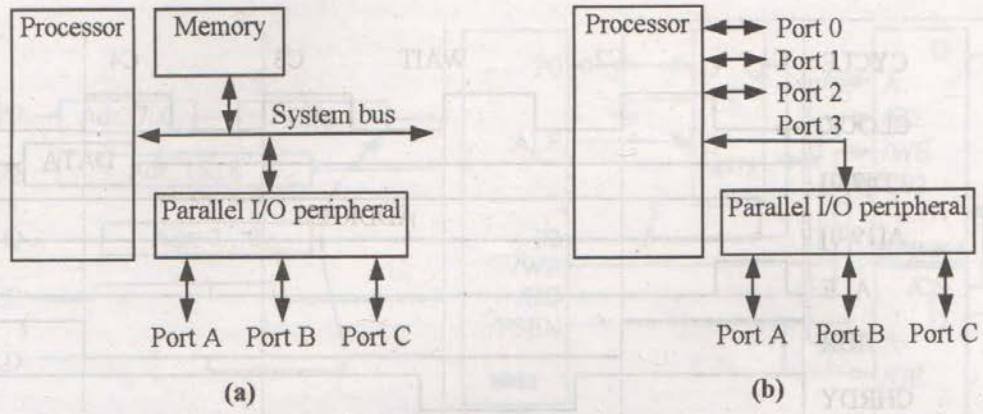


Figure 6.6: Parallel I/O: (a) adding parallel I/O to a bus-based I/O processor, (b) extended parallel I/O.

In *bus-based I/O*, the microprocessor has a set of address, data, and control ports corresponding to bus lines, and uses the bus to access memory as well as peripherals. The microprocessor has the bus protocol built in to its hardware. Specifically, the software does not implement the protocol but merely executes a single instruction that in turn causes the hardware to write or read data over the bus. We normally consider the access to the peripherals as I/O, but don't normally consider the access to memory as I/O, since the memory is considered more as a part of the microprocessor.

A system may require parallel I/O (port-based I/O), but a microprocessor may only support bus-based I/O. In this case, a parallel I/O peripheral may be used, as illustrated in Figure 6.6(a). The peripheral is connected to the system bus on one side, with corresponding address, data, and control lines, and has several ports on the other side, consisting just of a set of data lines. The ports are connected to registers inside the peripheral, and the microprocessor can read and write those registers in order to read and write the ports.

Even when a microprocessor supports port-based I/O, we may require more ports than are available. In this case, a parallel I/O peripheral can again be used, as illustrated in Figure 6.6(b). The microprocessor has four ports in this example, one of which is used to interface with a parallel I/O peripheral, which itself has three ports. Thus, we have extended the number of available ports from four to six. Using such a peripheral in this manner is often referred to as *extended parallel I/O*.

Memory-Mapped I/O and Standard I/O

In bus-based I/O, there are two methods for a microprocessor to communicate with peripherals, known as memory-mapped I/O and standard I/O.

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

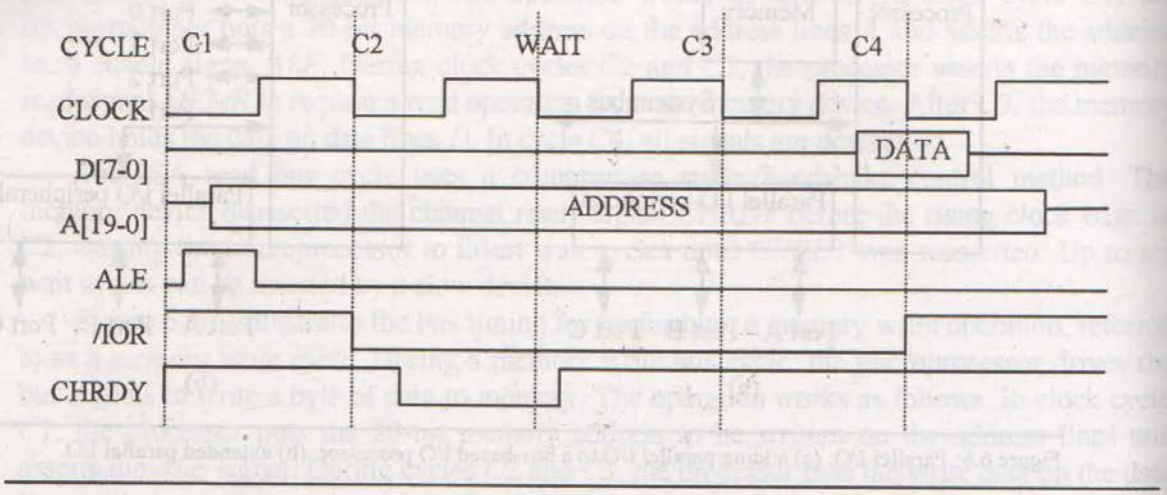


Figure 6.7: ISA bus protocol for standard I/O.

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

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

```
IN R0, A
IN R1, B
ADD R0, R1
OUT A, R0
```

Advantages of standard I/O include no loss of memory addresses to the use as I/O addresses, and potentially simpler address decoding logic in peripherals. Address decoding logic can be simplified with standard I/O if we know that there will only be a small number of peripherals, because the peripherals can then ignore high-order address bits. For example, a bus may have a 16-bit address, but we may know there will never be more than 256 I/O addresses required. The peripherals can thus safely ignore the high-order 8 address bits, resulting in smaller and/or faster address comparators in each peripheral. Note that we can build a system using both standard and memory-mapped I/O, since peripherals in the memory space act just like memory themselves.

b. Discuss multilevel bus architectures with the help of an industry standard multilevel bus. (4)

Answer:

Multilevel Bus Architectures

A microprocessor-based embedded system will have numerous types of communications that must take place, varying in their frequencies and speed requirements. The most frequent and high-speed communications will likely be between the microprocessor and its memories. Less frequent communications, requiring less speed, will be between the microprocessor and its peripherals, like a UART. We could try to implement a single high-speed bus for all the communications, but this approach has several disadvantages. First, it requires each peripheral

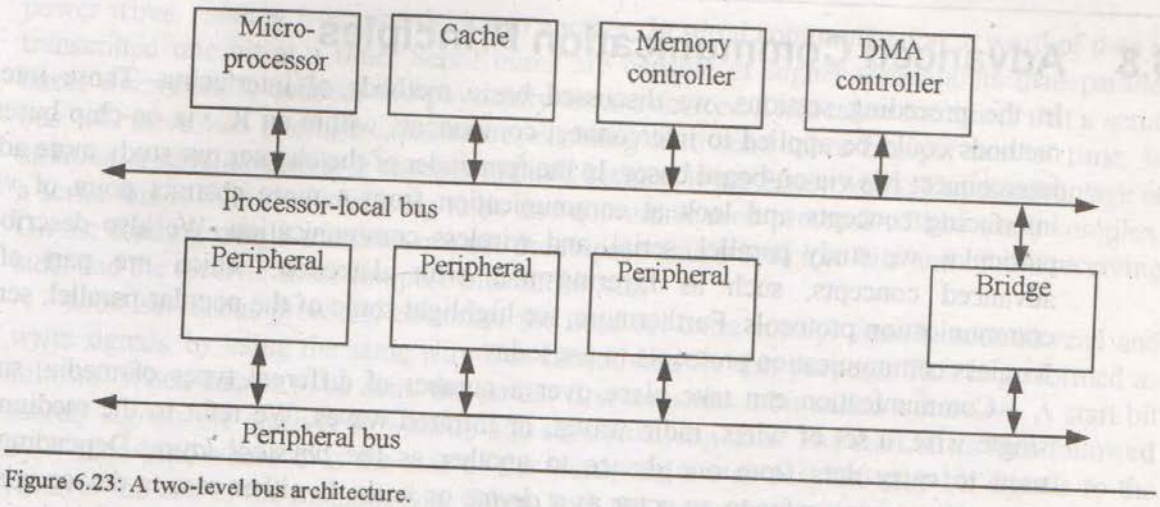


Figure 6.23: A two-level bus architecture.

to have a high-speed bus interface. Since a peripheral may not need such high-speed communication, having such an interface may result in extra gates, power consumption and cost. Second, since a high-speed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, having too many peripherals on the bus may result in a slower bus.

Therefore, we often design systems with two levels of buses: a high-speed processor local bus and a lower-speed peripheral bus, as illustrated in Figure 6.23. The *processor local bus* typically connects the microprocessor, cache, memory controllers, and certain high-speed coprocessors, and is processor specific. It is usually wide, as wide as a memory word.

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

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

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

c. Discuss the IEEE 802.11 protocol standard for bus communication. What are the protocols for wireless communication? (8)

Answer:

In this section, we briefly introduce three new and emerging wireless protocols, namely IrDA, Bluetooth, and the IEEE 802.11.

IrDA

The Infrared Data Association (IrDA) is an international organization that creates and promotes interoperable, low-cost, infrared data interconnection standards that support a walk-up, point-to-point user model. Their protocol suite, also commonly referred to as IrDA, is designed to support transmission of data between two devices over short-range point-to-point infrared at speeds between 9.6 kbps and 4 Mbps. IrDA is that small, semitransparent, red window that you may have wondered about on your notebook computer. Over the last several years, IrDA hardware has been deployed in notebook computers, printers, personal digital assistants, digital cameras, public phones, and even cell phones. One of the reasons for this has been the simplicity and low cost of IrDA hardware. Unfortunately, until recently, the hardware has not been available for applications programmers to use because of a lack of suitable protocol drivers.

Microsoft Windows CE 1.0 was the first Windows operating system to provide built-in IrDA support. Windows 2000 and Windows 98 now also include support for the same IrDA programming APIs that have enabled file sharing applications and games on Windows CE. IrDA implementations are becoming available on several popular embedded operating systems.

Bluetooth

Bluetooth is a new and global standard for wireless connectivity. This protocol is based on a low-cost, short-range radio link. The radio frequency used by Bluetooth is globally available. When two Bluetooth-equipped devices come within 10 meters of each other, they can establish a connection. Because Bluetooth uses a radio-based link, it doesn't require a line-of-sight connection in order to communicate. For example, your laptop could send information to a printer in the next room, or your microwave oven could send a message to your cordless phone telling you that your meal is ready. In the future, Bluetooth is likely to be standard in tens of millions of mobile phones, PCs, laptops and a whole range of other electronic devices.

IEEE 802.11

IEEE 802.11 is an IEEE-proposed standard for wireless local area networks (LANs). There are two different ways to configure a network: ad-hoc and infrastructure. In the ad-hoc network, computers are brought together to form a network on the fly, Here, there is no structure to the network, there are no fixed points, and usually every node is able to communicate with every other node. Although it seems that order would be difficult to maintain in this type of network, special algorithms have been designed to elect one machine as the master station of the network with the others being servants. Another algorithm in ad-

hoc network architectures uses a broadcast and flooding method to all other nodes to establish who's who. The second type of network structure used in wireless LANs is the infrastructure. This architecture uses fixed network access points with which mobile nodes can communicate. These network access points are sometime connected to landlines to widen the LAN's capability by bridging wireless nodes to other wired nodes. If service areas overlap, handoffs can occur. This structure is very similar to the present day cellular networks around the world.

The IEEE 802.11 protocol places specifications on the parameters of both the physical PHY and medium access control MAC layers of the network. The PHY layer, which actually handles the transmission of data between nodes, can use direct sequence spread spectrum, frequency-hopping spread spectrum, or infrared pulse position modulation. IEEE 802.11 makes provisions for data rates of either 1 Mbps or 2 Mbps, and calls for operation in the 2.4 to 2.4835 GHz frequency band, which is an unlicensed band for industrial, scientific, and medical applications, and 300 to 428,000 GHz for IR transmission. Infrared is generally considered to be more secure to eavesdropping, because IR transmissions require absolute line-of-sight links (no transmission is possible outside any simply connected space or around corners), as opposed to radio frequency transmissions, which can penetrate walls and be intercepted by third parties unknowingly. However, infrared transmissions can be adversely affected by sunlight, and the spread-spectrum protocol of IEEE 802.11 does provide some rudimentary security for typical data transfers.

The MAC layer is a set of protocols, which is responsible for maintaining order in the use of a shared medium. The IEEE 802.11 standard specifies a carrier sense multiple access with collision avoidance CSMA/CA protocol. In this protocol, when a node receives a packet to be transmitted, it first listens to ensure no other node is transmitting. If the channel is clear, it then transmits the packet. Otherwise, it chooses a random backoff-factor, which determines the amount of time the node must wait, until it is allowed to transmit its packet. During periods in which the channel is clear, the transmitting node decrements its backoff counter. When the backoff counter reaches zero, the node transmits the packet. Since the probability that two nodes will choose the same backoff factor is small, collisions between packets are minimized. Collision detection, as is employed in Ethernet, cannot be used for the radio frequency transmissions of IEEE 802.11. The reason for this is that when a node is transmitting it cannot hear any other node in the system, which may be transmitting, since its own signal will drown out any others arriving at the node.

Whenever a packet is to be transmitted, the transmitting node first sends out a short ready-to-send RTS packet containing information on the length of the packet. If the receiving node hears the RTS, it responds with a short clear-to-send CTS packet. After this exchange, the transmitting node sends its packet. When the packet is received successfully, as determined by a cyclic redundancy check, the receiving node transmits an acknowledgment ACK packet.

Q.7 a. Define Tasks and Task States. (4)

Answer:

Tasks and Task States

The basic building block of software written under an RTOS is the task. Tasks are very simple to write: under most RTOSs a task is simply a subroutine. At some point in your program, you make one or more calls to a function in the RTOS that starts tasks, telling it which subroutine is the starting point for each task and some other parameters that we'll discuss later, such as the task's priority, where the RTOS should find memory for the task's stack, and so on. Most RTOSs allow you to have as many tasks as you could reasonably want.

Each task in an RTOS is always in one of three states:

Running—which means that the microprocessor is executing the instructions that make up this task. Unless yours is a multiprocessor system, there is only one microprocessor, and hence only one task that is in the running state at any given time.

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

Blocked—which means that this task hasn't got anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example, a task that handles data coming in from a network will have nothing to do when there is no data. A task that responds to the user when he presses a button has nothing to do until the user presses the button. Any number of tasks can be in this state as well.

b. What is a re-entrant function? What are its characteristics? What are the grey areas in re-entrancy? Explain with example. (6)

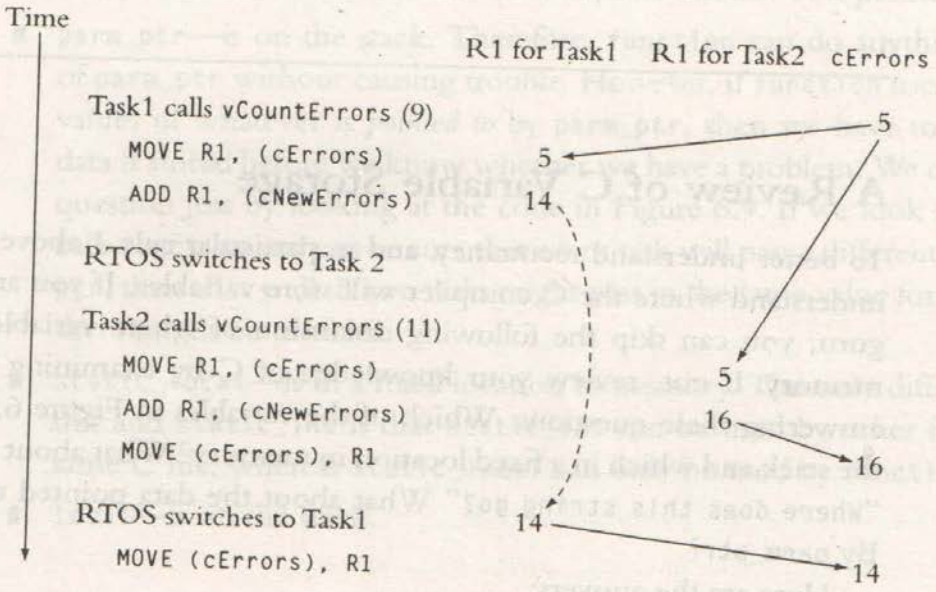
Answer:

Reentrancy

People sometimes characterize the problem in Figure 6.7 by saying that the shared function `vCountErrors` is not **reentrant.** Reentrant functions are functions that can be called by more than one task and that will always work correctly,

Figure 6.8 Why the Code in Figure 6.7 Fails

```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;     cErrors += cNewErrors;
      MOVE R1, (cErrors)
      ADD R1, (cNewErrors)
      Move (cErrors), R1
      RETURN
;}
```



even if the RTOS switches from one task to another in the middle of executing the function. The function `vCountErrors` does not qualify.

You apply three rules to decide if a function is reentrant:

1. A reentrant function may *not* use variables in a nonatomic way unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task.
2. A reentrant function may *not* call any other functions that are not themselves reentrant.
3. A reentrant function may *not* use the hardware in a nonatomic way.

Figure 6.9 Variable Storage

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   .
   .
   .
}
```

A Review of C Variable Storage

To better understand reentrancy, and in particular rule 1 above, you must first understand where the C compiler will store variables. If you are a C language guru, you can skip the following discussion of where variables are stored in memory. If not, review your knowledge of C by examining Figure 6.9 and answering these questions: Which of the variables in Figure 6.9 are stored on the stack and which in a fixed location in memory? What about the string literal "`Where does this string go?`" What about the data pointed to by `vPointer`? By `parm_ptr`?

Here are the answers:

- `static_int`—is in a fixed location in memory and is therefore shared by any task that happens to call `function`.
- `public_int`—Ditto. The only difference between `static_int` and `public_int` is that functions in other C files can access `public_int`, but they cannot access `static_int`. (This means, of course, that it is even harder to be sure that this variable is not used by multiple tasks, since it might be used by any function in any module anywhere in the system.)[6].

6. Of course, if you want, you *could* write code that passes the address of `static_int` to some function in another C file, and then that function could use `static_int`. After that, `static_int` would be as big a problem as `public_int`

- `initialized`—The same. The initial value makes no difference to where the variable is stored.
- `string`—The same.
- "`Where does this string go?`"—Also the same.
- `vPointer`—The pointer itself is in a fixed location in memory and is therefore a shared variable. If `function` uses or changes the data values *pointed to* by `vPointer`, then those data values are *also* shared among any tasks that happen to call `function`.
- `parm`—is on the stack.[7] If more than one task calls `function`, `parm` will be in a different location for each, because each task has its own stack. No matter how many tasks call `function`, the variable `parm` will not be a problem.
- `parm_ptr`—is on the stack. Therefore, `function` can do anything to the value of `parm_ptr` without causing trouble. However, if `function` uses or changes the values of whatever is *pointed to* by `parm_ptr`, then we have to ask where *that* data is stored before we know whether we have a problem. We can't answer that question just by looking at the code in Figure 6.9. If we look at the code that calls `function` and can be sure that every task will pass a different value for `parm_ptr`, then all is well. If two tasks might pass in the same value for `parm_ptr`, then there might be trouble.
- `static_local`—is in a fixed location in memory. The only difference between this and `static_int` is that `static_int` can be used by other functions in the same C file, whereas `static_local` can only be used by `function`.
- `local`—is on the stack.

Applying the Reentrancy Rules

Whether or not you are a C language guru, examine the function `display` in Figure 6.10 and decide if it is reentrant and why it is or isn't.

This function is not reentrant, for two reasons. First, the variable `fError` is in a fixed location in memory and is therefore shared by any task that calls `display`. The use of `fError` is not atomic, because the RTOS might switch

7. Be forewarned that there is at least one compiler out there that would put `parm`, `parm_ptr`, and `local` in fixed locations. This compiler is *not* in compliance with any C standard—but it produces code for an 8051, an 8-bit microcontroller. The ability to write in C for this tiny machine is worth some compromises.

Figure 6.10 Another Reentrancy Example

```
BOOL fError;   /* Someone else sets this */

void display (int j)
{
   if (!fError)
   {
      printf ("\nValue: %d", j);
      j = 0;
      fError = TRUE;
   }
   else
   {
      printf ("\nCould not display value");
      fError = FALSE;
   }
}
```

tasks between the time that it is tested and the time that it is set. This function therefore violates rule 1. Note that the variable `j` is no problem; it's on the stack.

The second problem is that this function may violate rule 2 as well. For this function to be reentrant, `printf` must also be reentrant. Is `printf` reentrant? Well, it might be, but don't count on it unless you have looked in the manual that comes with the compiler you are using and seen an explicit statement that it is.

Gray Areas of Reentrancy

There are some gray areas between reentrant and nonreentrant functions. The code here shows a very simple function in the gray area.

```
static int cErrors;

void vCountErrors (void)
{
   ++cErrors;
}
```

This function obviously modifies a nonstack variable, but rule 1 says that a reentrant function may not use nonstack variables *in a nonatomic way.* The question is: is incrementing `cErrors` atomic?

As with a number of the shared-data problems that we discussed in Chapter 4, we can answer this question only with a definite "maybe," because the answer depends upon the microprocessor and the compiler that you are using. If you're using an 8051, an 8-bit microcontroller, then ++cErrors is likely to compile into assembly code something like this:

```
        MOV     DPTR,#cErrors+01H
        MOVX    A,@DPTR
        INC     A
        MOVX    @DPTR,A
        JNZ     noCarry
        MOV     DPTR,# cErrors
        MOVX    A,@DPTR
        MOVX    @DPTR,A
noCarry:
        RET
```

which doesn't look very atomic and indeed isn't anywhere close to atomic, since it takes nine instructions to do the real work, and an interrupt (and consequent task switch) might occur anywhere among them.

But if you're using an Intel 80x86, you might get:

```
        INC     (cErrors)
        RET
```

which is atomic.

If you really need the performance of the one-instruction function and you're using an 80x86 and you put in lots of comments, perhaps you can get away with writing vCountErrors this way. However, there's no way to know that it will work with the next version of the compiler or with some other microprocessor to which you later have to port it. Writing vCountErrors this way is a way to put a little land mine in your system, just waiting to explode. Therefore, if you need vCountErrors to be reentrant, you should use one of the techniques discussed in the rest of this book.

c. When is a task blocked? What are the common issues of task state which are dealt by scheduler? (6)

Answer:

Blocked—which means that this task hasn't got anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example, a task that handles data coming in from a network will have nothing to do when there is no data. A task that responds to the user when he presses a button has nothing to do until the user presses the button. Any number of tasks can be in this state as well.

A part of the RTOS called the **scheduler** keeps track of the state of each task and decides which one task should go into the running state. Unlike the scheduler in Unix or Windows, the schedulers in most RTOSs are entirely simpleminded about which task should get the processor: they look at priorities you assign to the tasks, and among the tasks that are not in the blocked state, the one with the highest priority runs, and the rest of them wait in the ready state. The scheduler will not fiddle with task priorities: if a high-priority task hogs the microprocessor for a long time while lower-priority tasks are waiting in the ready state, that's too bad. The lower-priority tasks just have to wait; the scheduler assumes that you knew what you were doing when you set the task priorities.

Figure 6.1 shows the transitions among the three task states. In this book, we'll adopt the fairly common use of the verb **block** to mean "move into the blocked state," the verb **run** to mean "move into the running state" or "be in the running state," and the verb **switch** to mean "change which task is in the running state." The figure is self-explanatory, but there are a few consequences:

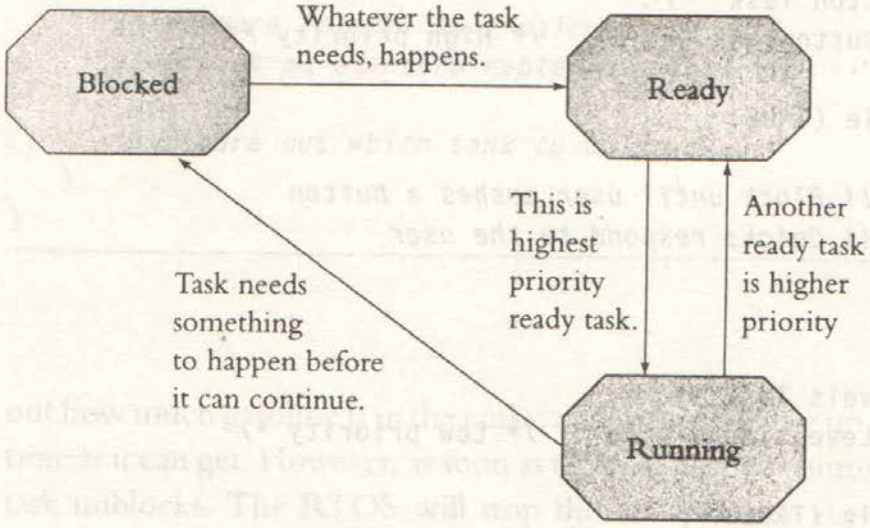
A task will only block because it decides for itself that it has run out of things to do. Other tasks in the system or the scheduler cannot decide for a task that it needs to wait for something. As a consequence of this, a task has to be running just before it is blocked: it has to execute the instructions that figure out that there's nothing more to do.

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

The shuffling of tasks between the ready and running states is entirely the work of the scheduler. Tasks can block themselves, and tasks and interrupt routines can

3. These distinctions among these other states are sometimes important to the engineers who wrote the RTOS (and perhaps to the marketers who are selling it, who want us to know how much we're getting for our money), but they are usually not important to the user.

Figure 6.1 Task States



move other tasks from the blocked state to the ready state, but the scheduler has control over the running state. (Of course, if a task is moved from the blocked to the ready state and has higher priority than the task that is running, the scheduler will move it to the running state immediately. We can argue about whether the task was ever really in the ready state at all, but this is a semantic argument. The reality is that some part of the application had to do something to the task—move it out of the blocked state—and then the scheduler had to make a decision.)

Q.8 a. What is the basic task of queues mailboxes and pipes? Explain with an example how they help to improve execution time? What decides the choice from amongst them? (6)

Answer:

Tasks must be able to communicate with one another to coordinate their activities or to share data. For example, in the underground tank monitoring system the task that calculates the amount of gas in the tanks must let other parts of the system know how much gasoline there is. In Telegraph, the system we discussed in Chapter 1 that connects a serial-port printer to a network, the tasks that receive data on the network must hand that data off to other tasks that pass the data on to the printer or that determine responses to send on the network.

In Chapter 6 we discussed using shared data and semaphores to allow tasks to communicate with one another. In this section we will discuss several other methods that most RTOSs offer: **queues, mailboxes,** and **pipes.**

Here's a very simple example. Suppose that we have two tasks, `Task1` and `Task2`, each of which has a number of high-priority, urgent things to do. Suppose also that from time to time these two tasks discover error conditions that must be reported on a network, a time-consuming process. In order not to delay `Task1` and `Task2`, it makes sense to have a separate task, `ErrorsTask`, that is responsible for reporting the error conditions on the network. Whenever `Task1` or `Task2` discovers an error, it reports that error to `ErrorsTask` and then goes on about its own business. The error reporting process undertaken by `ErrorsTask` does not delay the other tasks.

An RTOS queue is the way to implement this design. Figure 7.1 shows how it is done. In Figure 7.1, when `Task1` or `Task2` needs to log errors, it calls `vLogError`. The `vLogError` function puts the error on a queue of errors for `ErrorsTask` to deal with.

The `AddToQueue` function adds (many people use the term **posts**) the value of the integer parameter it is passed to a queue of integer values the RTOS maintains internally. The `ReadFromQueue` function reads the value at the head of the queue and returns it to the caller. If the queue is empty, `ReadFromQueue`

b. What issues are involved in using queues in an RTOS? (4)

Answer:

Most RTOSs require that you initialize your queues before you use them, by calling a function provided for this purpose. On some systems, it is also up to you to allocate the memory that the RTOS will manage as a queue. As with semaphores, it makes most sense to initialize queues in some code that is guaranteed to run before any task tries to use them.

- Since most RTOSs allow you to have as many queues as you want, you pass an additional parameter to every queue function: the identity of the queue to which you want to write or from which you want to read. Various systems do this in various ways.
- If your code tries to write to a queue when the queue is full, the RTOS must either return an error to let you know that the write operation failed (a more common RTOS behavior) or it must block the task until some other task reads data from the queue and thereby creates some space (a less common RTOS behavior). Your code must deal with whichever of these behaviors your RTOS exhibits.
- Many RTOSs include a function that will read from a queue if there is any data and will return an error code if not. This function is in addition to the one that will block your task if the queue is empty.
- The amount of data that the RTOS lets you write to the queue in one call may not be exactly the amount that you want to write. Many RTOSs are inflexible about this. One common RTOS characteristic is to allow you to write onto a queue in one call the number of bytes taken up by a void pointer.

c. How is INTERRUPT ROUTINE code different from Task Code? For an INTERRUPT ROUTINE explain with a suitable example as to how it should work in an RTOS? (6)

Answer:

Interrupt routines in most RTOS environments must follow two rules that do not apply to task code.

Rule 1. An interrupt routine must not call any RTOS function that might block the caller. Therefore, interrupt routines must not get semaphores, read from queues or mailboxes that might be empty, wait for events, and so on. If an interrupt routine calls an RTOS function and gets blocked, then, in addition to the interrupt routine, the task that was running when the interrupt occurred will be blocked, even if that task is the highest-priority task. Also, most interrupt routines must run to completion to reset the hardware to be ready for the next interrupt.

Rule 2. An interrupt routine may not call any RTOS function that might cause the RTOS to switch tasks unless the RTOS knows that an interrupt routine, and not a task, is executing. This means that interrupt routines may not write to mailboxes or queues on which tasks may be waiting, set events, release semaphores, and so on—unless the RTOS knows it is an interrupt routine that is doing these things. If an interrupt routine breaks this rule, the RTOS might switch control away from the interrupt routine (which the RTOS thinks is a task) to run another task, and the interrupt routine may not complete for a long time, blocking at least all lower-priority interrupts and possibly all interrupts.

To understand rule 2, examine Figure 7.14, a naive view of how an interrupt routine *should* work under an RTOS. The graph shows how the microprocessor's

Figure 7.13 Legal Uses of RTOS Functions in Interrupt Routines

```
/* Queue for temperatures. */
int iQueueTemp;

void interrupt vReadTemperatures (void)
{
    int aTemperatures[2];       /* 16-bit temperatures. */
    int iError;

    /* Get a new set of temperatures. */
    aTemperatures[0] = !! read in value from hardware;
    aTemperatures[1] = !! read in value from hardware;

    /* Add the temperatures to a queue. */
    sc_qpost (iQueueTemp,
       (char *) ((aTemperatures[0] << 16) | aTemperatures[1]),
       &iError);
}

void vMainTask (void)
{
    long int lTemps;    /* 32 bits; the same size as a pointer. */
    int aTemperatures[2];
    int iError;

    while (TRUE)
    {
       lTemps = (long) sc_qpend (iQueueTemp, WAIT_FOREVER,
          sizeof(int), &iError);
       aTemperatures[0] = (int) (lTemps >> 16);
       aTemperatures[1] = (int) (lTemps & 0x0000ffff);
       if (aTemperatures[0] != aTemperatures[1])
          !! Set off howling alarm;
    }
}
```

Figure 7.14 How Interrupt Routines *Should* Work

attention shifted from one part of the code to another over time. The interrupt routine interrupts the lower-priority task, and, among other things, calls the RTOS to write a message to a mailbox (legal under rule 1, assuming that function can't block). When the interrupt routine exits, the RTOS arranges for the microprocessor to execute either the original task, or, if a higher-priority task was waiting on the mailbox, that higher-priority task.

Figure 7.15 shows what really happens, at least in the worst case. If the higher-priority task is blocked on the mailbox, then as soon as the interrupt routine writes to the mailbox, the RTOS unblocks the higher-priority task. Then the RTOS (knowing nothing about the interrupt routine) notices

Figure 7.15 What Would Really Happen

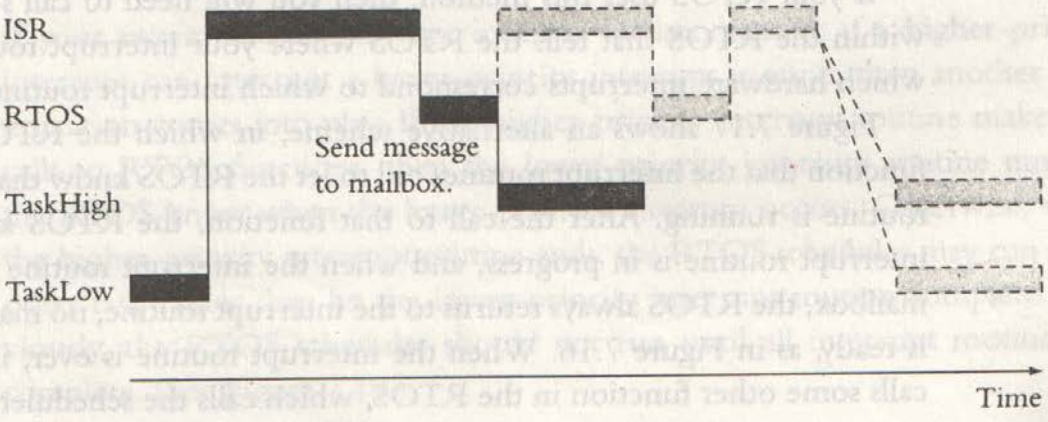
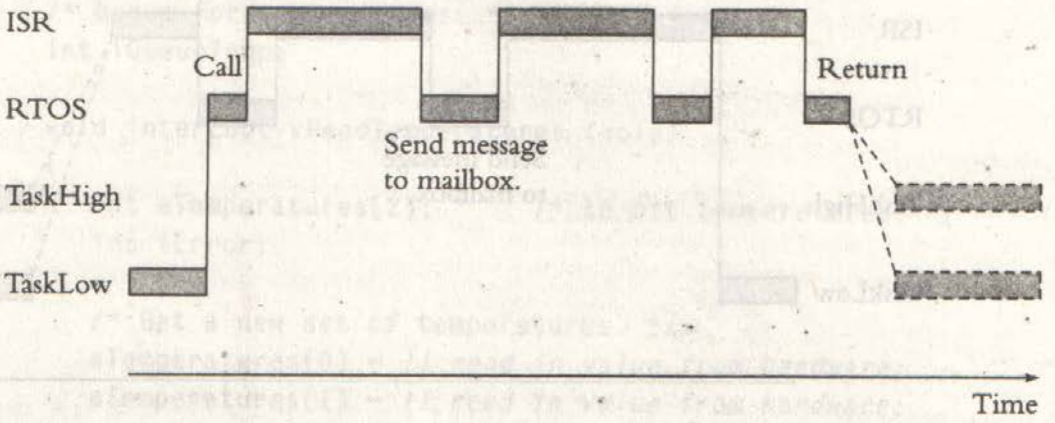


Figure 7.16 How Interrupt Routines Do Work



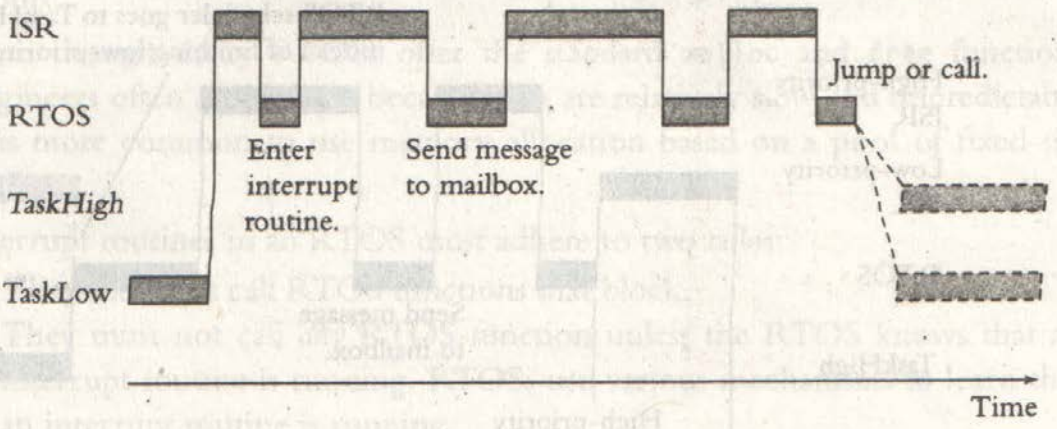
that the task that it thinks is running is no longer the highest-priority task that is ready to run. Therefore, instead of returning to the interrupt routine (which the RTOS thinks is part of the lower-priority task), the RTOS switches to the higher-priority task. The interrupt routine doesn't get to finish until later.

RTOSs use various methods for solving this problem, but all require your cooperation. Figure 7.16 shows the first scheme. In it, the RTOS intercepts all the interrupts and then calls your interrupt routine. By doing this, the RTOS finds out when an interrupt routine has started. When the interrupt routine later writes to the mailbox, the RTOS knows to return to the interrupt routine and not to switch tasks, no matter what task is unblocked by the write to the mailbox. When the interrupt routine is over, it returns, and the RTOS gets control again. The RTOS scheduler then figures out what task should now get the microprocessor.

If your RTOS uses this method, then you will need to call some function within the RTOS that tells the RTOS where your interrupt routines are and which hardware interrupts correspond to which interrupt routines.

Figure 7.17 shows an alternative scheme, in which the RTOS provides a function that the interrupt routines call to let the RTOS know that an interrupt routine is running. After the call to that function, the RTOS knows that an interrupt routine is in progress, and when the interrupt routine writes to the mailbox, the RTOS always returns to the interrupt routine, no matter what task is ready, as in Figure 7.16. When the interrupt routine is over, it jumps to or calls some other function in the RTOS, which calls the scheduler to figure out

Figure 7.17 How Interrupt Routines Do Work: Plan B



what task should now get the microprocessor. Essentially, this procedure disables the scheduler for the duration of the interrupt routine.

In this plan, your interrupt routines must call the appropriate RTOS functions at the right moments.

Some RTOSs use a third mechanism: they provide a separate set of functions especially for interrupt routines. So for example, in addition to `OSSemPost`, there might be `OSISRSemPost`, which is to be called from interrupt routines. `OSISRSemPost` is the same as `OSSemPost`, except that it always returns to the interrupt routine that calls it, never to some other task. In this method, the RTOS also has a function the interrupt routine calls when it is over, and that function calls the scheduler.

Q.9 a. How do we select the optimum number of tasks a system work should be divided into? (6)

Answer:

One of the first problems in an embedded-system design is to divide your system's work into RTOS tasks. An immediate, obvious question is "Am I better off with more tasks or with fewer tasks?" To answer that question, let's look at the advantages and disadvantages of using a larger number of tasks. First, the advantages:

- With more tasks you have better control of the relative response times of the different parts of your system's work. If you divide the work into eight tasks, for example, you can assign eight different priority levels. You'll get good response times for the work done in the higher-priority tasks (at the expense of the response time for the work done in the lower-priority tasks). If you put all that same work into one task, then you will get response more akin to that of the round-robin architecture discussed in Chapter 5. If you use a number of tasks somewhere in between one and eight, you'll get response somewhere in between.
- With more tasks your system can be somewhat more modular. If your system has a printer and a serial port and a network connection and a keyboard, and if you handle all of these devices in one task, then that task will of necessity be somewhat messy. Using a separate task for each device allows for cleaner code.
- With more tasks you can sometimes encapsulate data more effectively. If the network connection is handled by a separate task, only the code in that task needs access to the variables that indicate the status of the network interface.

Now for the disadvantages:

- With more tasks you are likely to have more data shared among two or more tasks. This may well translate into requirements for more semaphores, and hence into more microprocessor time lost handling the semaphores and into more semaphore-related bugs.
- With more tasks you are likely to have more requirements to pass messages from one task to another through pipes, mailboxes, queues, and so on. This will also translate into more microprocessor time and more chances for bugs.

Table 8.1 Timings of an RTOS on a 20 MHz Intel 80386

Service	Time
Get a semaphore	10 microseconds (μsec)
Release a semaphore	6–38 μsec
Switch tasks	17–35 μsec
Write to a queue	49–68 μsec
Read from a queue	12–38 μsec
Create a task	158 μsec
Destroy a task	36–57 μsec

- Each task requires a stack; therefore, with more tasks (and hence more stacks) you will probably need more memory, at least for stack space, and perhaps for intertask messages as well.
- Each time the RTOS switches tasks, a certain amount of microprocessor time evaporates saving the context of the task that is stopping and restoring the context of the task that is about to run. Other things being equal, a design with more tasks will probably lead to a system in which the RTOS switches tasks more often and therefore a system with less throughput.
- More tasks probably means more calls to the RTOS. RTOS vendors promote their products by telling you how fast they can switch tasks, put messages into mailboxes, set events, and so on. And the RTOS vendors have indeed made their systems fast. However, the RTOS functions don't do anything your customers care about. The typical laser printer customer is unimpressed by claims that a printer switches tasks 2000 times per second; his question is "How fast does it print?" Your system runs faster if it *avoids* calling the RTOS functions: the irony is that once you decide to use an RTOS, your best design is often the one that uses it least. Table 8.1 shows the timings from one RTOS running on a 20 MHz Intel 80386, a relatively fast processor. These times are short, certainly, but they aren't zero. Calling these functions frequently can add up to a lot of processing overhead.

b. Give the pseudo code for a Task structure. Explain the pros and cons of it. (5)

Answer:

Figure 8.5 shows pseudo-code for the task structure you should use most of the time.

The task in Figure 8.5 remains in an infinite loop, waiting for an RTOS signal that there is something for it to do. That signal is most commonly in the form of a message from a queue from which this task (and only this task) reads

Figure 8.5 Recommended Task Structure

```
vtaska.c

!! Private static data is declared here

void vTaskA (void)
{
   !! More private data declared here, either static
   !! or on the stack

   !! Initialization code, if needed.

   while (FOREVER)
   {
      !! Wait for a system signal (event, queue message, etc.)

      switch (!!type of signal)
      {
         case !! signal type 1:
            .
            .
            .
            break;

         case !! signal type 2:

            .
            .
            .
            break;
         .
         .
         .
      }
   }
}
```

anywhere, because all of its data is private, although that's a rule that often has to be broken.)

- When there is nothing for this task to do, its input queue will be empty, and the task will block and use up no microprocessor time.
- This task does not have public data that other tasks can share; other tasks that wish to see or change its private data write requests into the queue, and this task handles them. There is no concern that other tasks using the data use semaphores properly; there is no shared data, and there are no semaphores.

If you are familiar with Windows programming, you will see that this task structure is very similar to the structure of the window routine in Windows.

Tasks in an embedded system are often structured as state machines: the state is stored in private variables within the task; the messages that the task receives on its queue are the events. This construction is natural, because the RTOS ensures that the events will get queued neatly one after another, and the task will deal with them systematically one at a time.

Different task structures occasionally make sense. For example, the task in Figure 8.4 blocks in two places: on its input queue and during the delay. The alternate structure works for that task, because it can't do anything during the delay anyway. If messages are written to its input queue while the task is waiting for the flash memory to complete a write, those messages may as well stay on the queue. It is pointless to have the task read the messages out of the queue when it can't deal with them.

c. Saving memory and power are critical to an embedded system. Discuss methods for saving power in an embedded system. (5)

Answer:

Unlike desktop systems with their megabytes, embedded systems often have limited memory, as we discussed in Chapter 1. Conserving memory space is a subject that could take up several chapters; here we'll discuss just a few considerations specific to embedded systems.

In an embedded system, you may be short of code space, you may be short of data space, or you may be short of both. They are not interchangeable, since code must be stored in ROM and data in RAM. When you are working on saving memory, you must therefore make sure that you are saving the right sort. Packing data structures, for example, saves data space but is likely to cost code space, since your program must unpack the data to use it.

The methods for saving data space are the familiar ones of squeezing data into efficient structures. One special consideration if you use an RTOS is that each task needs memory space for its stack. Therefore, you should ensure that your system allocates only as much stack memory as is needed. The first method for determining how much stack space a task needs is to examine your code. Each function call, function parameter, and local variable takes up a certain number of bytes on the stack, depending upon your microprocessor and compiler, and you can search your code for the deepest combination of function nesting, parameters, and local variables. You must then add space for the worst-case nesting of interrupt routines, and you need to allow some amount of space for the RTOS itself, an amount you can usually find in the RTOS manual. The principle behind this method is simple; carrying out this method can prove surprisingly difficult, however. The second method is experimental. Fill each stack with some recognizable data pattern at startup, run the system for a period of time, stop it, and then examine how much of the data pattern was overwritten on each stack. This method may be easier to perform, but it is difficult to be sure that the worst case happened during the experiment.

Here are a few ways to save code space. Some of these techniques have obvious disadvantages; apply those only if they're needed to squeeze your code into your ROM.

- Make sure that you aren't using two functions to do the same thing. For example, if your code calls the standard C library `memcpy` function in 28 places and calls the standard (and very similar) `memmove` function once, check to see if you can't change that one call to `memmove` into a call to `memcpy` and get `memmove` out of your program. Alternatively, perhaps you can change the 28 calls to `memcpy` into calls to `memmove` and get rid of `memcpy`. Look at the listings from your linker/locator (discussed in Chapter 9) to see which functions are large enough to be worth trying to eliminate in this manner.
- Check that your development tools aren't sabotaging you. Calling `memcpy` might cause your tools to drag in `memmove`, `memset`, `memcmp`, `strcpy`, `strncpy`, `strset`, and who knows what else, even if you don't use those other functions. The manuals that come with your tools should indicate how to prevent this. Otherwise, consider writing your own function, perhaps `mymemcpy`, that will perform the same operation as `memcpy` but that won't be joined to all those other functions.
- Configure your RTOS to contain only those functions that you need. If your software does not use pipes, for example, leaving the RTOS pipe function in your system will certainly waste code space, and it may waste data space, too, if those functions need some space for static data.
- Look at the assembly language listings created by your cross-compiler (discussed in Chapter 9) to see if certain of your C statements translate into huge numbers of instructions. Surprising things often pop out of such an investigation. For example, the code below shows three methods of initializing `iMember` in the `a_sMyData` array of structures. Although all three do the same thing, the compiler may turn them into radically different amounts of code. Don't try to guess which method will be the best; compile them and look at the listings.

```
struct sMyStruct a_sMyData[3];
struct sMyStruct *p_sMyData;
int i;

/* Method 1 for initializing data */
a_sMyData[0].iMember = 0;
a_sMyData[1].iMember = 5;
a_sMyData[2].iMember = 10;

/* Method 2 */
for (i = 0; i < 3; ++i)
   a_sMyData[i].iMember = 5 * i;
```

```
/* Method 3 */
i = 0;
p_sMyData = a_sMyData;
do
{
   p_sMyData->iMember = i;
   i += 5;
   ++p_sMyData;
} while (i < 10);
```

- Consider using static variables instead of variables on the stack. Many microprocessors can read and write static variables using fewer instructions than they do for stack variables. If you are using one of these microprocessors, you will save space by declaring local variables to be static. If your code contains a function that accepts as a parameter a pointer to a structure that the function uses extensively, copying that structure into a static structure can also be a code space-saver. For example

```
void vFixStructureCompact (struct sMyStruct *p_sMyData)
{
   static struct sMyStruct sLocalData;
   static int i, j, k;

   /* Copy the struct in p_sMyData to sLocalData */
   memcpy (&sLocalData, p_sMyData, sizeof sLocalData);

   !! Do all sorts of work in structure sLocalData, using
   !! i, j, and k as scratch variables.

   /* Copy the data back to p_sMyData */
   memcpy (p_sMyData, &sLocalData, sizeof sLocalData);
}
```

may take up much less space than the more obvious

```
void vFixStructureLarge (struct sMyStruct *p_sMyData)
{
   int i, j, k;

   !! Do all sorts of work in structure pointed to by
   !! p_sMyData, using i, j, and k as scratch variables.
}
```

Of course, `vFixStructureCompact` is not reentrant, it may be slower than `vFixStructureLarge` (since `memcpy` takes some time to execute), and `sLocalData`

will use up additional data space, but if you can't fit your program into the ROM otherwise, this technique is worth pursuing. You can gauge whether this method is worthwhile by rewriting a few of your routines this way, compiling them, and examining the compiler listings.

- If you are using an 8-bit processor, consider using `char` variables instead of `int` variables. For example, the innocent-looking

```
int i;
struct sMyStruct sMyData[23];
.
.
.
for (i = 0; i < 23; ++i)
  sMyData[i].charStructMember = -1 * i;
```

can translate into a huge amount of code compared to

```
char ch;
struct sMyStruct sMyData[23];
.
.
.
for (ch = 0; ch < 23; ++ch)
  sMyData[ch].charStructMember = -1 * ch;
```

simply because arithmetic with `int` variables is so much more complex than arithmetic with `char` variables for an 8-bit processor. The `for` statement, the array reference, and of course the multiplication by −1 all require calculation.

- If all else fails, you can usually save a lot of space—at the cost of a lot of headaches—by writing your code in assembly language. Before doing this, try writing a few pieces of code in assembly to get a feel for how much space you might save (and how much work it will be to write and to maintain).

As we discussed in Chapter 1, some embedded systems run on battery power, and for these systems, battery life is often a big issue. The primary method for preserving battery power is to turn off parts or all of the system whenever possible. That includes the microprocessor. Specific methods for doing this vary considerably from one system to another; this section contains a few general notes on the subject.

Most embedded-system microprocessors have at least one power-saving mode; many have several. Software can typically put the microprocessor into

one of these modes with a special instruction or by writing a value to a control register within the microprocessor. The modes have names such as **sleep mode, low-power mode, idle mode, standby mode,** and so on. Each microprocessor is different, however; you have to read the manual about yours to know the characteristics of its particular power-saving modes.

A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback typically is that the only way to start the microprocessor up again is to reset it. This means that the hardware engineer must design some circuitry to do this at an appropriate moment. It also means that your program will start over from the beginning each time the microprocessor leaves its power-saving mode; your software must then figure out whether the system is coming up for the first time or whether it is just waking up after a short sleep. One simple way to do this is to write a recognizable signature into the RAM the first time the system starts, say by writing the value 0x9283ab3c at location 0x0100. Whenever the system starts, your program can check location 0x0100. If the system was turned off, location 0x0100 will contain garbage; if the system is waking up after a sleep, your program will find 0x9283ab3c. More sophisticated methods are also available. Static RAM uses very little power when the microprocessor isn't executing instructions, so it is common just to leave it on, even when software puts the microprocessor to sleep.

Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate. Any interrupt starts the microprocessor up again, and the microprocessor will execute the corresponding interrupt routine and then resume the task code from the instruction that follows the one that put the microprocessor to sleep. This mode saves less power than the one described above. However, no special hardware is required, and you don't have the hassle of having your software restart from the beginning. Further, you can use this power-saving mode even while other things are going on. For example, a built-in DMA channel can continue to send data to a UART, the timers will continue to run, interrupt, and awaken the microprocessor, and so on.

If you plan to have your software put your microprocessor into one of its power-saving modes, plan to write fast software. The faster your software finishes its work, the sooner it can put the microprocessor back into a power-saving mode and stop using up the battery.

Another common method for saving power is to turn off the entire system and have the user turn it back on when it is needed. The cordless bar-code

scanner is an example of such a system. It turns itself off until the user pulls the trigger to initiate another scan; the trigger-pull turns the entire system back on. If you plan to do this, then the hardware engineer must obviously provide a means for software to turn the system off and for the user to turn it back on. The method obviously reduces power consumption to zero; however, software must save in EEROM or flash any values it will need to know when the system starts again, since the RAM will forget its data when the power goes off.

If your system needs to turn off any part of itself other than the microprocessor, then the hardware engineer must provide mechanisms for software to do that. The data sheets for the parts in your system will tell you which draw enough power to be worthwhile turning off. In general, parts that have a lot of signals that change frequently from high to low and back draw most power.

Text Book

Embedded System Design, AUnified Hardware/ Software Introduction , Frank Vahid/ Tony Givargis, Third Edition ,2010 reprint , John Wiley Student Edition.

An Embedded Software Primer, David E. Simon, Tenth Impression 2011,Pearson Education .